Context Dynamics

The Physics of AI Agent Behavior

Shane

2026

Contents

Preface **1**

I The Measurement Problem **7**

1 Flatland **9**

2 The Partnership Problem **25**

3 The Embedding Problem **43**

II A New Coordinate System **57**

4 Frames, Not Turns **59**

5 Trajectories **75**

6 The Coordinate Space **89**

III Dynamics **107**

7 Inertia **109**

8 Entropy **129**

9 Space **147**

10 Conservation and Symmetry **163**

IV Composition and Prediction **179**

11 The Fused Signal **181**

12 Phases and Transitions **195**

13 Intervention Theory **209**

14 The Closed Loop **221**

V Architecture **235**

15 The Write-Back Architecture **237**

16 The Harness Dynamics Model **251**

17 Open Horizons **265**

Epilogue: Beyond Machines **279**

Notation **287**

Glossary **293**

Bibliography **301**

Preface

I built AI agents for a living before I understood what they were doing.

That sounds like a confession, but it is closer to a description of the industry. In 2024 and 2025, the people shipping agent systems—myself included—had good benchmarks, reasonable cost models, and almost no insight into what happened between the first prompt and the final output. We could tell you whether an agent passed a task. We could not tell you how it got there, why it chose the path it did, or what it was doing during the long stretches where it appeared to be thinking.

I do not think we were negligent. I think we lacked a framework. The measurements available to us—tokens consumed, turns taken, tools called, pass/fail—were all surface-level projections of a process we had no vocabulary to describe. We could see the shadow on the wall. We could not see the thing casting it.

This book is my attempt to describe the thing.

Why This Book Exists

The gap between agent capability and agent understanding is growing.

Every few months, a new generation of models arrives. They are better at coding, better at reasoning, better at multi-step planning. Benchmarks track these improvements with precision. SWE-bench scores go up. HumanEval pass rates climb. Cost per task falls.

But ask a different kind of question—*how* did the agent solve that problem? Where did it spend its attention? When did it commit to a strategy, and what would have happened if it had committed to a different one?—and the measurement infrastructure goes quiet. Not because nobody cares, but because we have no instruments for it.

This is the same gap that exists between a car's speedometer and an understanding of internal combustion. The speedometer is useful. It tells

you something real. But it cannot help you build a better engine, diagnose a misfire, or predict when the transmission will fail. For that, you need a model of what is happening inside.

Agents are being deployed in medicine, law, finance, software engineering, scientific research. The stakes are rising. The speedometer is not enough.

I started working on the ideas in this book because I needed them myself. I was building an agent system and I could not answer basic questions about its behavior. How much of the context window was the agent actually using at each step? When it made a decision, was that decision informed by the full state of the problem, or by a narrow slice of it? Were there moments where the agent's behavior was predictable from the structure of the task, and if so, could I use that predictability to intervene before things went wrong?

The answers required a framework that did not exist. So I started building one.

Who This Book Is For

Three groups of readers, with three different entry points.

If you want to understand the framework without implementing it, this book is written for you. Parts I through IV develop the core ideas from first principles. Every concept is grounded in concrete examples from real agent behavior. You do not need to write code to follow the argument. You need to be comfortable with quantitative reasoning—graphs, distributions, the occasional equation—but no specific technical background is assumed beyond that.

If you build agent systems and want actionable measurement tools, the framework here is designed to be implemented. The quantities I define—inertia, entropy rate, trajectory curvature, phase transitions—are all computable from data you already have or can collect with modest instrumentation. Part V walks through one complete implementation. The formal definitions in the notation appendix are precise enough to code against.

If you are a researcher looking for open problems, nearly every chapter ends with questions I cannot answer. The framework is young. It has

structure and internal consistency, but it has not been tested at the scale or breadth it deserves. Several chapters identify specific conjectures that could be verified or falsified with the right experiments. I would be glad to be proven wrong about any of them, because that would mean someone found a better model.

How to Read This Book

The book has five parts and an epilogue.

Parts I through IV develop the theory. They are architecture-agnostic—the framework applies to any agent system, regardless of which model it runs, which tools it uses, or how its orchestration is designed. Part I establishes the measurement problem. Part II introduces the coordinate space. Part III defines the dynamical quantities. Part IV builds the governance model. Each part depends on the ones before it, but each is also somewhat self-contained: if you understand the coordinate space (Part II), you can read the governance chapters (Part IV) without internalizing every detail of the dynamics (Part III).

Part V is one implementation. It describes a real system built around the principles developed in the first four parts. This section is specific where the theory is general. It names particular technologies, describes particular trade-offs, and shows what the abstract quantities look like when they are computed from actual agent traces. If you only care about the framework and not the implementation, you can stop after Part IV and miss nothing essential.

The Epilogue steps back. Context dynamics, as a way of thinking about information flow in situated reasoning systems, is not limited to AI. The same patterns show up wherever an agent—human, institutional, biological—makes sequential decisions under uncertainty with finite memory. I do not develop that argument fully. I gesture at it, because I think it is true and I think other people will do more with it than I can.

Chapters build on each other, but I have tried to make the dependencies explicit. If a chapter requires a concept from an earlier one, it says so. If you are the kind of reader who skips around, the glossary and notation appendix are designed to support you.

What This Book Is Not

It is not a programming manual. There is code in Part V, but the book's contribution is the framework, not the implementation.

It is not a product pitch. The system described in Part V is one system. The framework in Parts I through IV is the point. If you build a better system using these ideas, the book has succeeded.

It is not a survey paper. I cite related work where it is relevant, but I have not attempted to comprehensively catalog the agent evaluation literature. That literature is growing fast and any survey would be stale before publication.

It is not a prompt engineering guide. Context dynamics operates at a layer below prompt construction. It is about the shape of information as it flows through a reasoning system, not about how to phrase instructions to get better outputs.

What it is: a framework for thinking about agent behavior as a measurable, governable phenomenon. A set of instruments for seeing the sphere instead of the shadow.

Acknowledgments

This book grew out of building software, not out of writing papers. The ideas were sharpened by the daily experience of watching agents do things I did not expect and not having the vocabulary to describe what I was seeing.

I owe debts to the open-source agent community, whose shared traces and benchmarks made it possible to test these ideas against real data. To the researchers working on mechanistic interpretability, who demonstrated that the internal states of neural systems are not opaque if you build the right instruments. And to the small group of people who read early drafts of these chapters and told me, with varying degrees of politeness, where the reasoning was sloppy.

The remaining sloppiness is mine.

This framework is incomplete. Writing a book requires drawing a line and saying: this is what I know now. I have drawn that line. What lies beyond

it is more interesting than what lies within it, and I look forward to being surprised by what other people find there.

Part I

The Measurement Problem

Chapter 1

Flatland

Here is a telemetry report from an AI coding agent:

> **Task:** Fix failing test in `user_auth_test.py` **Turns:** 12 **Tokens consumed:** 48,291 **Tool calls:** 8 **Result:** Pass ✓

The test was red. The agent ran for a while. The test turned green. The report is tidy and complete. It has numbers and a checkmark.

Pause on this for a moment.

What actually happened during those twelve turns? Did the agent understand the failure immediately and spend most of its time on a tricky implementation detail? Did it misdiagnose the problem twice before stumbling onto the fix? Did it read the right file first, or did it wander through six irrelevant modules before finding the one that mattered?

None of those questions can be answered from the report.

The telemetry captured something. But the something it captured is much smaller than the thing we are trying to understand. The report tells us what the agent *consumed* and whether it *succeeded*. It tells us nothing about how it *thought*.

And yet, across the industry, reports like this one form the backbone of how we evaluate AI agent performance. Leaderboards rank agents by pass rate. Cost analyses compare token consumption. Speed benchmarks measure turns to completion. All of these metrics are drawn from the same flat surface of numbers. All of them are silent about what happened between the measurements.

The way we measure AI agents is wrong.

Not slightly miscalibrated. Not missing a few edge cases. Wrong in the way that counting sunrises is wrong if you are trying to understand orbital

mechanics. The individual observations are accurate. The framework that connects them is missing.

This book is about building that framework.

The Flatland Problem

In 1884, Edwin Abbott published a novella called *Flatland: A Romance of Many Dimensions*. The story describes a world of two-dimensional beings who live on a flat plane. They can perceive length and width but have no concept of height. When a three-dimensional sphere passes through their world, they see it as a circle that appears from nothing, grows, shrinks, and vanishes. The sphere is real. What the Flatlanders observe is a projection of the sphere onto their two-dimensional world. The projection is not wrong, exactly. The circle does exist at each moment. But it misses the object's actual shape, its actual motion, its actual nature.

Agent telemetry, as practiced today, lives in Flatland.

We record turns, tokens, tool calls, pass rates, completion times. Each measurement is accurate. Each captures a real quantity. But these quantities are projections of a higher-dimensional process onto a flat surface of numbers. The agent's reasoning has shape and trajectory and momentum. Our instruments see a circle that grows and shrinks.

The Flatlander is not stupid. The Flatlander measures precisely. The Flatlander's mistake is believing that the projection is the object. That the circle *is* the sphere. That nothing has been lost by collapsing three dimensions into two.

We make the same mistake with agents every day.

To understand what agents actually do when they work, we need to leave Flatland. We need instruments that can perceive the shape of the sphere.

That is what this book builds, piece by piece.

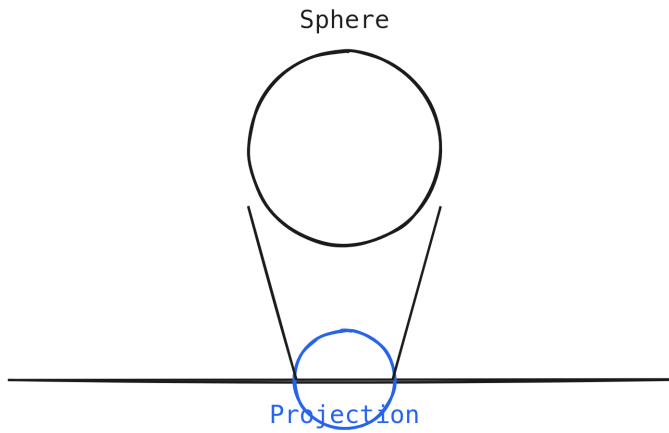

Figure 1.1: A sphere passing through Flatland appears as a growing and shrinking circle

What an AI Agent Actually Does

Before we can discuss measuring agents well, we should be precise about what an agent is, because the word has been stretched to mean almost anything.

An AI agent, for our purposes, is a program that uses a large language model (LLM) to accomplish tasks through an iterative loop. The loop works like this:

1. **Observe.** The agent receives information about its current situation. This might be a task description, the contents of a file, the output of a command it just ran, or an error message from a failed attempt.
2. **Decide.** The LLM processes all the information it has accumulated so far and produces a response. That response might include reasoning about what to do next, or it might be a direct action.
3. **Act.** The agent executes an action in the world. It might read a file, run a test, edit source code, search a codebase, or execute a shell

command. These actions are called *tool calls*, because the agent uses external tools to interact with its environment.

4. **Observe again.** The results of the action come back. A file's contents. A test's output. An error trace. This new information gets added to the agent's *context*: the running accumulation of everything it has seen and done during this session.

Then the loop repeats. Observe, decide, act, observe. Each full cycle is called a *turn*.

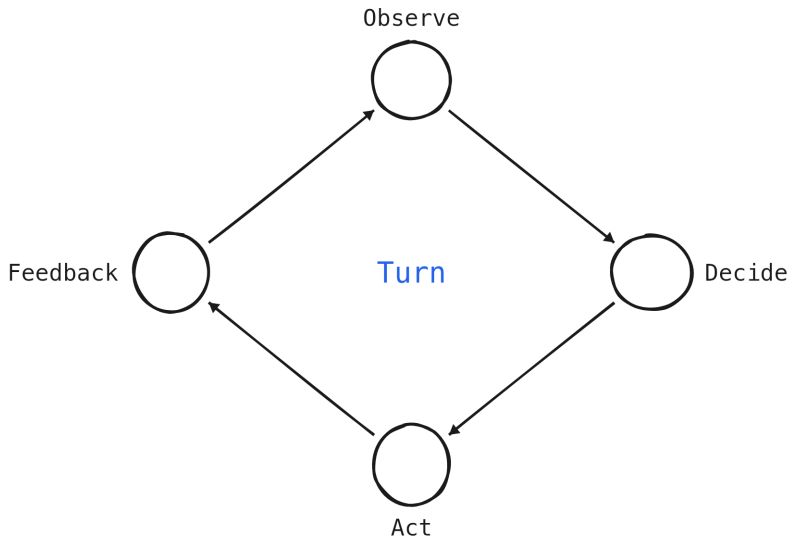

Figure 1.2: The agent loop: observe, decide, act, observe again

A session is a sequence of turns, starting when the agent receives a task and ending when it reports a result (or gives up, or crashes, or is terminated). During a session, the agent's context grows with each turn. It accumulates observations, actions taken, results received, and whatever reasoning the model has produced along the way.

The context is the agent's working memory. Everything it knows about the task lives there.

That's the mechanical picture. An agent loops, accumulates context, and acts on the world until it arrives at a result. Simple enough to describe.

The trouble starts when you try to understand what happens *during* the loop.

What Telemetry Sees vs. What Happened

Return to our failing test. Here is what the telemetry system recorded:

Telemetry Report

Metric	Value
Task	Fix failing test `test_session_expiry`
Turns	12
Tokens (input)	31,407
Tokens (output)	16,884
Tool calls	8
Unique files read	5
Files modified	1
Time elapsed	47s
Result	Pass

Clean. Quantified. And almost completely uninformative about what the agent did.

Here is what the agent actually did during those twelve turns, reconstructed from a full session trace:

Session Trace

Phase 1 — Orientation (Turns 1–2) The agent reads the failing test. It reads the error output. It identifies the assertion that failed: `session.expires_at` is `None` when the test expects a timestamp. It forms an initial hypothesis: the `create_session` function is not setting the expiry field.

Phase 2 — Model Formation (Turns 3–5) The agent reads `session.py`, the module that contains `create_session`. It discovers the function *does* set `expires_at`, but only when a `ttl` parameter is provided. The test does not pass `ttl`. The agent reads the test more carefully and finds a config fixture that should be providing a default TTL. It reads the config fixture. The fixture is correct.

The agent has now formed a mental model of the problem: the config fixture provides a default TTL, but `create_session` is not reading from the config when no explicit `ttl` argument is given.

Phase 3 — Commitment (Turn 6) The agent decides to modify `create_session` to fall back to the configured default TTL when no explicit value is passed. It edits the function.

Phase 4 — Failure (Turn 7) The agent runs the test. It fails again, with a different error. The config lookup returns a string (`"3600"`), but the function expects an integer. The agent did not anticipate this type mismatch.

Phase 5 — Branching (Turns 8–10) The agent considers two options. It could add `int()` conversion in `create_session`, or it could fix the config fixture to return an integer. It reads two other functions that consume config values. Both do their own type conversion. The agent decides to follow the existing pattern and add conversion in `create_session`.

Phase 6 — Convergence (Turns 11–12) The agent adds the type conversion, runs the test, and the test passes. It reports success.

Compare these two views. The telemetry report is a shadow on the wall. The session trace reveals an agent that oriented quickly, built a correct model of the codebase, committed to a reasonable fix, encountered an unexpected obstacle, investigated the codebase's conventions before choosing a response, and converged on a solution consistent with existing patterns.

That is a meaningfully different story from "twelve turns, forty-eight thousand tokens, pass." It would also be a meaningfully different story from an agent that tried four wrong fixes before landing on the right one by trial and error. Both agents could produce the same telemetry report. They could not produce the same session trace.

The telemetry report is not wrong. Every number in it is accurate. But it has collapsed a six-phase cognitive process into five scalars and a boolean. The information that distinguishes good problem-solving from bad problem-solving has been projected away.

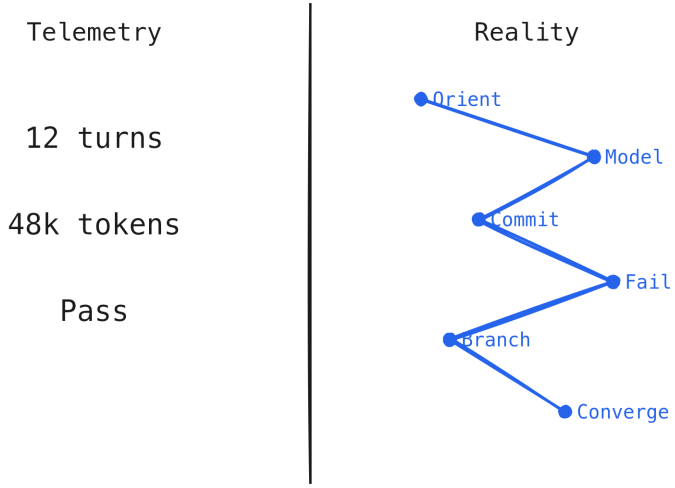

Figure 1.3: Telemetry metrics on the left versus the winding path of a real session trace on the right

Turns Are Frames

A subtle but critical mistake underlies most current evaluation systems: the assumption that a turn represents a unit of reasoning. It does not.

A turn is a frame.

Consider how film works. A movie runs at twenty-four frames per second. Each frame is a still photograph. If you examine a single frame, you can identify the objects in the scene, measure their positions, catalog their colors. If you examine a sequence of frames, you can do the same for each one, and you can count how many frames the sequence contains.

What you cannot do, from individual frames alone, is understand the *motion*.

A character reaches for a door handle. That reaching is not contained in any single frame. It exists in the transition *between* frames. The motion is reconstructed by a viewer who perceives the continuity across the sequence. The individual frames are measurements. The motion is the thing being measured.

Agent turns work the same way. Each turn captures a snapshot: here is what the agent saw, here is what it decided, here is what it did. But the agent's reasoning does not start and stop at turn boundaries. A hypothesis formed in Turn 3 shapes every action through Turn 10. A failed attempt in Turn 7 causes a reassessment that restructures the entire approach. The cognitive process is continuous. The turns sample it at discrete intervals.

When we count turns, we are counting frames. When we average token usage per turn, we are measuring the brightness of individual photographs. When we catalog tool calls, we are listing the objects visible in each still image.

None of this tells us about the motion.

The motion is where meaning lives. Go back to the session trace from earlier. The agent's decision in Turn 6 did not appear from nowhere. It was the product of a model built across the five preceding turns. The *decision* happened at Turn 6. The *reasoning that produced the decision* spanned the entire sequence before it. No individual frame contains it.

An agent that reaches a correct fix by building a coherent mental model has different motion than one that reaches the same fix by trying random modifications until one works. The frames might look similar. The motion between them is nothing alike.

Most current agent telemetry systems observe sessions the way early astronomy observed planetary motion: recording positions without understanding the dynamics that produced them. Tycho Brahe spent decades cataloging the positions of Mars with extraordinary precision. Those positions were accurate. They were also, by themselves, insufficient to explain *why* Mars moved the way it did. That required Kepler, who looked at the same data and found the shape of the orbit. The positions were projections. The orbit was the object.

We are in the Tycho Brahe phase of agent evaluation. We have positions. We need orbits.

Sessions Are Trajectories

If turns are frames and the interesting thing is the motion between them, then a session is not really a sequence of turns. A session is a *path*.

The agent starts somewhere: a task description, an initial set of observations. It moves through a space of possible actions and states, reading files, forming hypotheses, testing modifications, revising plans. Each action changes what the agent knows, which changes what it will do next, which changes where it ends up. The session is the path it traces through this process.

That path is what we will call a *trajectory*.

The word is borrowed from physics, where a trajectory is the path an object follows through space over time. A cannonball has a trajectory. A planet has a trajectory. The trajectory is shaped by forces: gravity, air resistance, initial velocity. If you know the forces, you can predict the path. If you can observe the path, you can infer the forces.

Agent sessions have trajectories too. And those trajectories are shaped by forces. Different forces than gravity, but forces nonetheless: properties of the task, the codebase, the agent's accumulated context, the model's tendencies. We will define these forces precisely in later chapters. For now, the claim is just this: what we are observing, when we watch an agent work, is not a sequence of isolated decisions. It is a continuous path through a problem space, shaped by identifiable forces.

The telemetry report records five points on that path and a label for the endpoint. The trajectory is the path itself.

This distinction has practical consequences. Two trajectories can arrive at the same endpoint (task completed, test passing) while following completely different paths. One path might be efficient and direct. The other might be wasteful and meandering. One might demonstrate genuine understanding. The other might demonstrate persistence plus luck. The endpoint is the same. The trajectories are different. And if we want to build better agents, the trajectories are what we need to understand.

We will formalize the concept of a trajectory in Chapter 5, defining it as a mathematical object with measurable properties. But the intuition should already be forming: the unit of analysis for understanding agent behavior is not the turn, not the session summary, not the pass/fail outcome. It is the shape of the path the agent took to get there.

The Benchmark Problem

This brings us to benchmarks.

The AI field relies heavily on benchmarks to evaluate agent capability. SWE-bench, for instance, presents agents with real software engineering tasks drawn from open-source repositories. Each task requires the agent to diagnose and fix an actual bug. The agent either produces a correct fix or it does not. The benchmark score is the percentage of tasks an agent solves correctly.

SWE-bench and benchmarks like it have been enormously useful. They replaced vague claims about agent ability with concrete, reproducible measurements. That was genuine progress.

But benchmark scores inherit the Flatland problem. They are projections.

Consider two agents evaluated on the same set of one hundred tasks. Both solve sixty-three. Their benchmark scores are identical: 63%. Based on this number, they appear equivalent.

Here are the trajectories for one of those tasks, a bug where a CLI tool silently ignores invalid configuration keys:

Agent A — Trajectory

Turn 1: Read the failing test. Identified the expected behavior: invalid keys should produce a warning.

Turn 2: Read the CLI config parser. Found the `load_config()` function. Noticed it uses `dict.update()`, which silently overwrites without validation.

Turn 3: Read the config schema definition. Identified the set of valid keys.

Turn 4: Formed a hypothesis: `load_config()` needs a validation step between reading the config file and applying values. Wrote the validation function. Added a call to it in `load_config()`.

Turn 5: Ran the test. Passed.

Total: 5 turns, 19K tokens, 4 tool calls.

Agent B — Trajectory

Turn 1: Read the failing test.

Turns 2–3: Searched the codebase for "config" and "invalid key." Neither search found the relevant file.

Turn 4: Found the config parser through the CLI entry point. Read it.

Turn 5: Added a warning for invalid keys using a hardcoded list of valid ones.

Turn 6: Ran the test. Failed. The hardcoded list was incomplete and the warning format was wrong.

Turns 7–8: Read the test more carefully, found the config schema, and rewrote the validation to use both.

Turn 9: Ran the test. Failed. A missing import.

Turns 10–11: Fixed the import. Test passed.

Total: 11 turns, 43K tokens, 9 tool calls.

Same task. Same outcome. Same contribution to the benchmark score. Completely different behavior.

Agent A diagnosed the problem in three turns, formed a coherent plan, and executed it. Agent B wandered through the codebase, attempted a brittle fix based on incomplete information, failed, read the things it should have read initially, and recovered. Agent A's trajectory is direct and efficient. Agent B's trajectory is exploratory and reactive, with a false start that cost six turns.

On a benchmark, these agents score the same on this task. If this pattern holds across tasks, they score the same overall. A benchmark consumer comparing them would have no reason to prefer one over the other.

But any engineering team that has worked with AI agents knows these are not equivalent. Agent A is more likely to handle novel tasks well, because its behavior suggests it builds models before acting. Agent B is more likely to fail on harder tasks, because its strategy of "try something and see" scales poorly with problem complexity.

The benchmark score tells you *where the agent ended up*. It does not tell you *how it got there*. And the how is where the useful information lives.

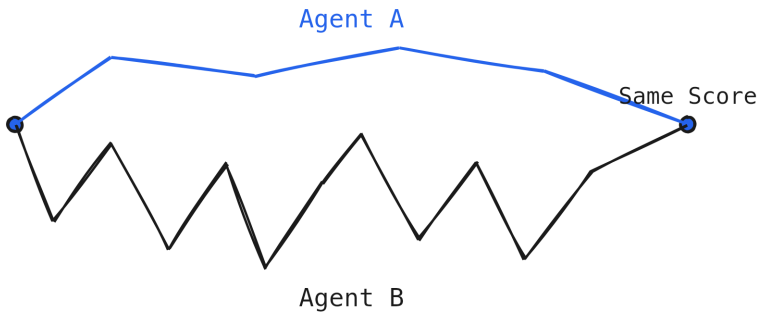

Figure 1.4: Agent A takes a direct trajectory while Agent B wanders before arriving at the same outcome

This is not an argument against benchmarks. Benchmarks do what they were designed to do: measure outcomes at scale. The argument is that outcomes are not enough. An evaluation system that cannot distinguish Agent A from Agent B is an evaluation system that cannot tell you which agent to deploy, which architecture to invest in, or what went wrong when a session fails. It can tell you *that* something failed. It cannot tell you *why*.

Three Forces

If agent sessions are trajectories, and trajectories have shape, then something must be giving them that shape. A cannonball follows a parabola because gravity pulls it downward while its initial velocity carries it forward. The shape of the path reveals the forces acting on the object.

Agent trajectories have forces too. We will develop these rigorously throughout the book, but three of them deserve a brief introduction here, because they will reappear constantly.

Inertia. When an agent has spent six turns reading files, building a model of a subsystem, and committing to a particular diagnosis, switching to a different diagnosis is expensive. Not because the agent cannot physically do it. The next turn can contain anything. But the accumulated context pulls the agent forward along its current path. All those observations, all that reasoning, all that built-up model: abandoning it means the agent wasted that work. The deeper the agent is into a particular line of investigation, the more context weight is pulling it forward, and the harder it becomes to change direction.

This is not a metaphor. It is measurable. Agents demonstrably commit to failing strategies longer than they should, and the likelihood of persisting increases with the amount of context invested in the current approach. The context creates inertia. Go back to Agent B from the benchmark example. By Turn 5, it had already written a hardcoded validation list. Even after that approach failed, it did not discard the idea of inline validation. It refined the existing approach rather than reconsidering from scratch. The context weight of Turns 2 through 5 kept it on the same path.

Entropy. An agent starts a session with a clear task and (usually) a clear initial plan. As turns accumulate, things drift. The agent encounters unexpected complications. It makes small detours. Each individual deviation is minor. But they accumulate. After thirty turns, the agent's current activity may have drifted substantially from its original plan. The coherence between intent and action degrades over time.

This is a form of entropy. The agent's trajectory becomes less ordered as the session progresses. Early turns tend to be purposeful and directed. Late turns tend to be reactive and patchwork. The rate at which this degra-

dation occurs varies by task, by agent, and by something we have not named yet but will: the structure of the space the agent is working in.

Space. The "space" an agent navigates is not physical space. It is the structure of the problem and its environment. A well-organized codebase with clear module boundaries, consistent naming, and good test coverage creates an easy space to navigate. Files are where you expect them. Functions do what their names suggest. Tests tell you what broke and why.

A tangled codebase with circular dependencies, inconsistent conventions, and spotty test coverage creates a difficult space. The agent has to work harder to find relevant information. Its hypotheses are more likely to be wrong, because the codebase's structure is misleading. The topology of the space shapes the trajectory.

Two agents with identical capabilities will produce different trajectories in different spaces. The same agent will look competent in a well-structured codebase and confused in a poorly-structured one. The agent did not change. The space did.

These three forces act on every agent session. Inertia pulls the agent along its current path. Entropy degrades coherence over time. Space determines how easy or hard it is to navigate toward a solution. The interplay of these forces produces the trajectory's shape.

(And notice: none of these forces can be derived from turn counts, token totals, or pass/fail outcomes. They are properties of motion, not properties of individual frames.)

What This Book Builds

The field of AI agents is moving fast. New models, new benchmarks, new scaffolding frameworks appear weekly. Most of this work operates in Flatland: better prompts that produce higher pass rates, faster execution that reduces token costs, improved tool designs that require fewer turns. This work is valuable. Agents have gotten meaningfully more capable.

But the instruments have not kept up with the agents. We are building increasingly sophisticated systems and evaluating them with the measurement equivalent of a bathroom scale. The scale tells you a number. It

does not tell you whether that number is muscle or fat, whether it is going up or down, or what you should do about it.

This book develops a framework for measuring agent behavior in its full dimensionality. We call it Context Dynamics: the study of how AI agents move through context space under the influence of identifiable forces.

But before we can build better instruments, we need to understand both sides of the measurement. Agents are not black boxes that appeared overnight. They emerged from specific technical lineages, carry specific tendencies, and fail in specific patterns. If we want to measure their motion, we need to know what kind of objects they are.

And there is a second complication. We are not neutral observers. The humans who deploy agents, evaluate them, and try to collaborate with them bring their own perceptual habits to the task. We anthropomorphize. We project intent where there is pattern-matching. We mistake fluency for understanding. Our own cognition is part of the measurement apparatus, and it introduces distortions we rarely account for.

So the next step is to look at both partners in this collaboration: what agents actually are, how they got to where they are now, and what we bring to the encounter when we sit down to work with them.

We refer to this domain as Context Dynamics. Not because we invented the phenomena it describes. Agents have always had trajectories. Context has always exerted forces. Evaluation has always been incomplete. The dynamics have been operating since the first agent loop executed its first turn.

We are giving them a name.

Chapter 2

The Partnership Problem

In January 2023, a friend of mine asked me to watch him use ChatGPT. He had been playing with it for a few weeks. He wanted to show me something. He typed a question about a tricky database migration he was working on. The response came back in seconds: clear, structured, confident. He leaned back and said, “It’s like having a senior engineer on call twenty-four hours a day.”

I looked at the screen. The answer was wrong. Not obviously wrong. Not gibberish. It had the shape of a correct answer. The SQL syntax was valid. The migration strategy was plausible. But it would have silently dropped a column constraint that his application depended on.

He didn’t notice. Why would he? The answer read like something a competent person would write. It had the texture of expertise. The confidence in the prose matched the confidence he was placing in it.

That moment has stayed with me, not because my friend was careless or naive. He’s a good engineer. It stayed with me because the gap between what he experienced and what actually happened was enormous, and neither of us had vocabulary for it. He experienced a conversation with a knowledgeable colleague. What actually happened was that a statistical model produced a sequence of tokens that maximized plausibility given the input. Those two descriptions refer to the same event. The distance between them is the territory this chapter explores.

Chapter 1 named Context Dynamics and gestured toward understanding both sides of the human-agent partnership. Now we need to actually look at those sides. How did agents get to where they are? What do humans bring to the encounter? The answers turn out to be tangled together, because every advance in agent capability has been accompanied by a corresponding shift in human perception. The two have co-evolved, and the

co-evolution has a pattern: at each stage, the gap between what agents appear to do and what they actually do has grown wider.

This is not a complaint. The gap is not a bug to be fixed. It is a structural feature of the partnership, and understanding it is a prerequisite for measuring anything useful about agent behavior.

So let's trace the history.

Ghosts in the Machine

Joseph Weizenbaum built ELIZA in 1966 at MIT. The program was a parlor trick, and Weizenbaum knew it. ELIZA matched patterns in user input and reflected them back as questions. You type "I'm worried about my mother." ELIZA responds, "Tell me more about your mother." No understanding. No model of you, your mother, or worry. Just string manipulation with therapeutic phrasing.

What disturbed Weizenbaum was not the program. It was the people.

His secretary asked him to leave the room so she could have a private conversation with ELIZA. Students spent hours talking to it, pouring out personal problems, despite knowing how it worked. Weizenbaum had explained the mechanism. They had seen the code. They talked to it anyway, and they reported feeling understood.

The gap between perception and reality existed from the very first human-computer conversation. ELIZA was a lookup table with personality. Its users experienced therapy. The distance was small, because ELIZA's responses were obviously formulaic if you paid attention. But the fact that the distance existed at all, that people projected understanding onto string matching, told us something important about the human side of this partnership. We are not neutral observers of agent behavior. We arrive with expectations, and those expectations do real work. For the next fifty years, the gap stayed roughly the same size. Chatbots got more sophisticated, adding more patterns, more rules, more branching logic. But the basic architecture remained: input processing, rule application, output generation. No learning. No persistent context. No model of the world. And humans continued to project more onto these systems than was there, in

rough proportion to how convincing the output looked. Then the proportion changed.

The Inflection

On November 30, 2022, OpenAI released ChatGPT. Within five days, a million people were using it. Within two months, a hundred million. No technology in history had been adopted that fast.

The speed mattered. Previous AI systems had been introduced to researchers, then developers, then early adopters, then the public. Each group developed intuitions about what the technology could and could not do before passing it along. ChatGPT skipped the line. A hundred million people encountered a language model with no preparation, no context, no framework for understanding what they were interacting with.

And what they encountered felt like a conversation with someone intelligent.

The system could write essays, debug code, explain quantum mechanics, compose poetry, translate between languages, and do all of it in a conversational style that felt natural. People described it as "thinking," "understanding," "knowing." These words felt accurate because the outputs matched what those words would predict. If something understands quantum mechanics, it can explain quantum mechanics clearly. ChatGPT explained quantum mechanics clearly. The inference felt airtight.

On the agent side, nothing had changed architecturally. ChatGPT was a transformer predicting the next token, the same fundamental operation that GPT-2 had performed in 2019. The difference was scale: more parameters, more training data, more compute, plus reinforcement learning from human feedback to make the outputs more conversational. The mechanism was the same. The results were qualitatively different.

This is the moment the gap blew open. Previous chatbots had been obviously limited. You could find the edges of ELIZA in thirty seconds. Even commercial chatbots from the 2010s would break down quickly under unexpected input. ChatGPT's edges were harder to find. You could use it for hours and never hit an obvious failure. When it was wrong, it was wrong in the way a smart person might be wrong: plausibly, confidently, with

good reasoning that happened to start from a flawed premise or end at an incorrect conclusion.

The hundred million people using it were not equipped to distinguish between "this system understands my question" and "this system is producing statistically likely continuations of my question." There was no reason they should have been. The distinction is counterintuitive. If the output looks like understanding, smells like understanding, and functions like understanding, calling it "statistically likely continuations" feels like pedantry.

But it is not pedantry. It is the difference between a system that knows when it is wrong and one that does not. A colleague who understands your database migration will notice the dropped constraint. A system that is predicting plausible next tokens has no mechanism for noticing, because "noticing" requires a model of your application's semantics that no amount of next-token prediction produces.

My friend's experience in January 2023 was not unusual. It was universal. Millions of people were sitting down with ChatGPT, having the experience of collaborating with an intelligent partner, and developing intuitions calibrated to that experience. Those intuitions would carry forward into every subsequent interaction with AI systems, including the ones where the stakes were higher and the failures less forgiving.

Into the Workflow

Before ChatGPT, GitHub had already changed the relationship between developers and AI with a quieter release. GitHub Copilot launched in June 2022, five months before ChatGPT. It was not a chatbot. It was a code completion engine, trained on billions of lines of public code, that sat inside your editor and suggested what to type next.

The shift was significant. ELIZA and ChatGPT were conversation partners. You talked to them. Copilot was a collaborator. You worked with it. The agent had entered the workflow.

This changed the failure mode. With a chatbot, a wrong answer is a wrong answer. You read it, evaluate it, accept or reject it. With Copilot, a wrong suggestion could slip directly into your code. You're typing a function, the

suggestion appears, you hit Tab, you keep typing. The cognitive cost of evaluating each suggestion is low but nonzero, and humans are optimizers. When the suggestions are right ninety percent of the time, evaluating the other ten percent starts to feel like overhead.

Developers reported a new experience: code they didn't fully understand appearing in their own files. Not because Copilot forced it there, but because they had accepted suggestions without reading them carefully. The code looked right. It followed the patterns of the surrounding code. It used the correct API calls. It handled the common cases. But sometimes it missed an edge case, or used a deprecated method, or introduced a subtle type error that the compiler would not catch.

"Looks right" and "is right" diverged. This distinction had always existed in programming. Experienced developers know that code which appears correct on first reading can harbor bugs that take hours to find. But Copilot accelerated the phenomenon. It produced code that looked right at a pace that exceeded most developers' ability to verify. The bottleneck shifted from writing code to evaluating code, and most workflows were not designed for that shift.

On the agent side, Copilot operated within a context window. It saw the current file, maybe some neighboring files, and pattern-matched against its training data to predict likely continuations. It had no model of the codebase as a whole. It did not understand what the code was supposed to do. It could not reason about whether a function's behavior was correct relative to its callers, its tests, or its specification. It predicted plausible code given local context. Sometimes local context was sufficient. Sometimes it was not.

The distinction between plausible and correct would become the central tension of the next era.

Agents That Act

In 2023 and 2024, something shifted again. Language models gained the ability to use tools.

The pattern was called ReAct, for Reason and Act, and it changed what agents could do. Instead of just generating text, an agent could now read

files, run shell commands, search the web, call APIs, and modify code. The model would reason about what to do, issue a tool call, receive the result, reason about what to do next, issue another tool call. A loop.

This was the moment when four separate intellectual traditions converged, though few people recognized it at the time.

Reinforcement learning had spent decades studying agents that observe states, take actions, and receive rewards. The RL community had the mathematics of sequential decision-making: policies, value functions, the exploration-exploitation tradeoff. They understood what it meant for an agent to learn from feedback.

The BDI tradition from cognitive science and philosophy of mind had studied how rational agents form beliefs about the world, commit to goals, and persist in plans until new evidence forces revision. They understood intention persistence, the tendency of a committed agent to refine its current approach rather than start over.

Software engineering had been building autonomous agents for decades: daemons, monitors, crawlers, orchestrators. Systems that observe their environment, apply rules, and take action, running continuously without human intervention. They understood what it meant for an agent to operate within a larger system.

And the tool-use community, the newest of the four, had figured out how to give language models hands. How to connect the reasoning engine to the physical world of files, processes, and network requests.

These traditions had developed independently, with different vocabularies, different formalisms, different conferences. But the tool-use agent that emerged in 2023 embodied all of them. It observed its environment (RL). It formed beliefs and committed to plans (BDI). It operated within a larger system of files, tests, and version control (software agents). And it reasoned in natural language while calling tools (tool-use loop).

Strip away the different vocabularies, and all four traditions describe the same structure: observe, decide, act, receive feedback. Then do it again. The loop is the agent. The traditions differ in how they formalize the components, what they emphasize, what they measure. But the loop is universal.

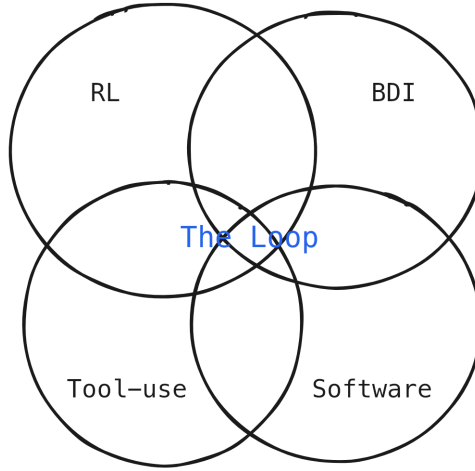

Figure 2.1: Four agent traditions converge on the same core loop

And once agents could act, the human experience changed profoundly. Excitement and unease arrived together. An agent that can read your files and run commands is not a conversation partner or a suggestion engine. It is a collaborator with the ability to change things. When you ask it to fix a bug, it reads code, forms a hypothesis, implements a fix, runs the tests, and iterates. When it works, the experience is exhilarating. When it breaks something, the experience is visceral in a way that a wrong chatbot answer never was.

The stakes rose because the actions were real. A chatbot's mistake is text on a screen. A tool-use agent's mistake is a modified file, a failed build, a corrupted state.

The Widening Gap

By 2025 and 2026, tool-use agents had matured into sustained collaborative systems. Claude Code, Codex, and their peers could maintain sessions lasting hours, executing multi-step plans, managing complex modi-

fications across large codebases, and adapting when their initial approach failed. The capabilities were real and growing.

But so was the gap.

The most capable agents still operate by the same mechanism: given accumulated context, predict a useful next action. They have gotten dramatically better at this prediction. Their context windows are larger, allowing them to hold more of a problem in view at once. Their planning horizons are longer, enabling multi-step strategies that earlier agents could not sustain. Their memory within a session is richer, carrying forward the results of earlier investigations. Their observability of the environment is deeper, reading not just files but project structure, test results, and runtime state. Their coupling to their environment is tighter, with more tools and more sophisticated tool use. Their autonomy is higher, running for longer stretches without human intervention.

Each of these dimensions represents genuine progress. Agents have moved along every axis. An agent from 2024 and an agent from 2026 are meaningfully different in what they can accomplish. The improvement is not illusory.

What is illusory is the model of the agent that humans construct from the experience of using it. A system that maintains a coherent plan across fifty tool calls feels like it understands the plan. A system that corrects its own mistakes feels like it has judgment. A system that asks clarifying questions feels like it is thinking about your problem. These experiences generate a mental model of the agent as something like a junior developer: not as experienced as you, but competent, trainable, and getting better.

That mental model is useful. It generates mostly correct predictions about what the agent will do next. But it is wrong in the places where it matters most. A junior developer knows when they are out of their depth. A language model does not. A junior developer can recognize that a task requires knowledge they lack and ask for help. A language model will attempt the task with equal confidence whether it has the relevant training data or not. A junior developer's mistakes correlate with difficulty. A language model's mistakes correlate with training distribution, which is not the same thing.

The gap between the junior-developer mental model and the predict-next-action reality is where the most consequential failures occur. Not the obvious failures, where the agent produces gibberish or crashes. The subtle failures, where the agent produces something that looks correct, passes cursory review, and breaks in production three weeks later.

We have traced the arc from ELIZA to the present. At each stage, the agents got more capable and the human perception of those agents got more sophisticated. But perception and reality did not converge. They diverged. The more capable the agent, the more convincing its outputs, and the harder it became to distinguish between what the agent appeared to be doing and what it was actually doing.

This is one half of the partnership problem. The agent side. The other half is us.

Because humans do not arrive at this collaboration as blank slates. We bring specific cognitive habits, specific biases, specific failure modes that interact with agent capabilities in predictable and measurable ways. Understanding those failure modes is not a philosophical exercise. It is a prerequisite for building any measurement framework that accounts for the full system: agent, human, and the space between them.

The Intelligence We Project

Start with a confession. I do it too. I say the model "thinks" about a problem. I say it "understands" my codebase. I say it "wants" to use a particular approach. Every one of those verbs is wrong, and I use them anyway, because the alternative is clumsy. "The model produced token sequences statistically consistent with understanding my codebase" is accurate and unusable.

The language would be harmless if it stayed as shorthand. It does not. The verbs leak into expectations. When I say the model understands my codebase, some part of me starts expecting the behaviors that understanding would produce. I give vaguer instructions because a colleague who understands the codebase can fill in the gaps. I get frustrated when the agent misses something "obvious" that was obvious to me only because I carry context the agent never had. The projection of intelligence

creates a set of expectations that the architecture cannot satisfy, and then I blame the architecture for failing to meet expectations I manufactured.

This happens at every level of experience. Novice users do it explicitly: they believe the agent is reasoning. Expert users do it implicitly: they know the mechanism is token prediction, but their interaction patterns assume reasoning anyway. The knowledge does not fix the behavior. I have explained transformer architecture to engineers who nodded along, understood every word, and then in the same session said "the model figured out what I meant." The anthropomorphic frame is stickier than the technical one.

The projection scales beyond individual interaction. Watch what happens when people build multi-agent systems. A pattern has emerged that I call the agent org chart. Someone decides their problem needs multiple agents, so they create a CEO agent, a CTO agent, a CMO agent, a marketing lead. They assign each agent a role description and wire them together in an organizational hierarchy.

The structure feels rigorous. It mirrors how human organizations work. There are reporting lines, areas of responsibility, handoffs between departments. The person who built it presents the architecture with genuine pride, because it looks like a well-designed org.

But calling one language model instance "CEO" does not give it strategic vision. Calling another "CTO" does not give it technical depth. They are the same model, with the same weights, running the same prediction algorithm. The role description in the system prompt is just another input to token prediction. The CEO agent does not have different competencies than the CTO agent. It has a different preamble.

The org chart is cargo-culting organizational design. Human organizations benefit from hierarchy because different people bring different knowledge, different perspectives, different cognitive styles. The VP of engineering and the VP of marketing genuinely see the product differently, because they are different people with different training and different incentive structures. Putting a different title on the same language model does not reproduce this divergence. It reproduces the org chart's shape while missing the property that makes org charts useful.

People who build these systems often cannot evaluate whether the outputs are actually better than what a single agent would produce, because the same cognitive bias that made the structure feel rigorous makes the outputs feel trustworthy. The feeling of rigor substitutes for the measurement of rigor.

This is a preview of a problem we will see again and again: the human side of the partnership introduces distortions that compound the agent side's limitations.

Confidence Without Calibration

We are bad at judging agent output. This is not a moral failing. It is a predictable consequence of how the output is shaped.

A language model optimizes for plausibility. Its outputs read like something a competent person would produce. When you review generated code, it has the right structure, the right variable names, the right patterns. It follows the conventions of the surrounding codebase. It handles the common cases. It looks like code you would write yourself, or code a colleague would submit in a pull request.

The failure mode is not loud wrongness. It is quiet wrongness. Obviously broken code is easy to catch. Code that is almost right, that works for every case except one, that handles the common path but misses the edge condition your application actually hits in production, requires deep evaluation to identify. And deep evaluation takes time and expertise.

Here is where the Dunning-Kruger problem enters. Your ability to evaluate generated code correlates with your own expertise in the relevant domain. A senior engineer reviewing generated database code will catch the dropped constraint. A junior engineer, or a senior engineer working outside their specialty, will see code that looks correct and approve it. Most people overestimate their ability to evaluate code in domains they are less familiar with. This is not unique to agent-generated code, but agent-generated code triggers it more reliably because the code always looks competent. Human-written bad code often has tells: inconsistent style, uncertain comments, visible hesitation. Generated code has no uncertainty. Every suggestion arrives with the same polish.

The result is a systematic bias toward accepting generated output. When code looks competent, the default is trust. Overriding that default requires effort, and effort is a finite resource that depletes over a long session of reviewing suggestions. By the fortieth suggestion, your review threshold has dropped. By the hundredth, you are skimming. The agent's output quality has not changed. Your evaluation quality has collapsed.

This creates a paradox for telemetry. If we use human evaluation as the ground truth for agent output quality, we are measuring with a ruler that bends depending on who holds it and how tired they are. Any measurement framework that relies on human judgment without accounting for the systematic biases in that judgment is measuring the wrong thing.

What We Bring to the Prompt

Now turn the lens on the input side. The quality of agent output depends on the quality of the prompt, and our prompts are terrible.

Not always. Not everyone's. But in the aggregate, human-to-agent communication is lossy, ambiguous, and contradictory in ways that human-to-human communication manages to survive only because humans share enormous amounts of implicit context. You can tell a colleague "fix the auth bug" and they will draw on shared knowledge of the codebase, the recent deploy, the customer complaint from last week, and the team's conventions about how auth works. A language model receives the string "fix the auth bug" and nothing else, unless you provide the rest.

We leave critical context unstated because it is obvious to us. We change direction mid-conversation without signaling. We ask for X when we mean Y, because we have not fully articulated what we want even to ourselves. We send three contradictory messages in a row as we think through the problem in real time, and then blame the agent for picking the wrong interpretation.

I have watched myself do this. I gave an agent a task, it went in a direction I did not like, and my first reaction was that the agent was wrong. When I went back and reread my prompt, the direction it chose was a reasonable interpretation of what I had written. The problem was not the agent's comprehension. The problem was my communication.

The prompt is a bottleneck, and we are the bottleneck's bottleneck. Every ambiguity in the input becomes noise in the output. The agent has no effective mechanism for pushing back. It can ask clarifying questions, but it does not know what it does not know. It cannot detect the unstated assumption that would change everything. It receives our worst communication habits and does its best with them, which means its output reflects our input fidelity more than most people realize.

There is an irony here. We hold agents to a standard of communication that we do not meet ourselves. We expect the agent to infer our intent from incomplete instructions, and we call it a failure when it infers wrong. But we are the ones who provided the incomplete instructions. The degraded output follows from degraded input, and the degradation starts with us.

The Ego in the Loop

There is an emotional dimension to this partnership that rarely gets discussed in technical contexts.

We take agent failures personally. When the agent breaks our code, the frustration is disproportionate to the damage, because the failure feels like a betrayal of trust. We gave it access. It was supposed to help. We take agent successes personally too. When it produces something elegant, we feel a glow of satisfaction, as though we wrote it. We did not.

Confirmation bias runs deep in agent interactions. When the agent agrees with our approach, we feel validated. When it suggests a different approach, our first instinct is resistance, not curiosity. We are easy targets for flattery. A model trained on human feedback has learned that agreeable responses get higher ratings. Our emotional responses to agent output are data about us, not about the agent.

This emotional entanglement is not trivial. It shapes which outputs we accept and which we reject. It determines whether we investigate a suspicious result or wave it through. It influences whether we give the agent a second chance after a failure or switch to doing the work ourselves out of spite. The ego in the loop is a measurement problem, because the human evaluator's judgment is contaminated by their emotional state, and the contamination is invisible to them. We believe we are making technical

assessments. We are making emotional ones and rationalizing them after the fact.

The Paradox of Imperfection

So we anthropomorphize, we misjudge quality, we communicate badly, and we let our egos interfere with evaluation. These are the failure modes of the human partner.

And they are also, in a different light, the capabilities of the human partner.

The same intuition that leads us to project intelligence onto language models also lets us sense when something is wrong before we can articulate why. A senior developer looks at generated code and feels uneasy. They cannot point to the bug yet. But the feeling is real, and it is grounded in years of pattern recognition that no prompt can transfer.

The same ego that makes us defensive about our approach also makes us care about the outcome. An agent optimizes a loss function. A human cares whether the software works for the people who use it. That caring is not computable, and it drives decisions that pure optimization would never reach. The developer who stays late to fix a bug that affects three users is not optimizing. They are exercising judgment shaped by empathy, by professional pride, by a sense of responsibility that no reward signal captures.

Creativity, ethical judgment, aesthetic sense, domain knowledge built over years of practice, the ability to hold a user's needs in mind while making technical tradeoffs. These are the qualities the human side brings to the partnership, and the current structure of that partnership wastes most of them. We spend our intuition on guessing whether generated code is correct instead of directing it toward design decisions. We spend our judgment on evaluating plausible-looking output instead of shaping strategy. We spend our creativity on crafting better prompts instead of crafting better systems. The partnership, as currently practiced, amplifies the human's worst qualities and underutilizes the best ones.

The measurement gap from Chapter 1 is a direct contributor to this waste. Without seeing the real shape of agent behavior, without observing the actual sequence of what the agent did and why, humans fall back on intuition

and bias as their primary evaluation tools. Intuition and bias are poor substitutes for observation. But when observation is unavailable, they are all we have.

Which suggests a way forward. If our intuitions about agent behavior are unreliable, and they are, then any serious attempt to understand the partnership needs to start somewhere other than intuition. We need a unit of observation that does not depend on how we feel about what the agent did. We need to record what actually happened.

The Frame

Every interaction between an agent and its environment has a structure, whether anyone records it or not. The agent receives information. It makes a decision. The decision produces an effect. The environment responds to that effect.

We can write this as a tuple: (o, d, a, f).

The variable o is the observation: the information available to the agent at the moment of decision. What files it has read, what errors it has seen, what the user has said. The variable d is the decision: what the agent chose to do, given that observation. The variable a is the action: the concrete effect on the environment. A file written, a command executed, a message sent. And f is the feedback: what the environment returned. A test result, a compiler error, a user response.

This is a *frame*. Chapter 1 used the word to mean a snapshot, a still photograph from a film reel. The meaning here is the same, made precise: one tick of the observe-decide-act-feedback loop, with its components recorded.

A frame is not a summary of what happened. It is a record of what happened. It captures the agent's information state at the moment of decision, the decision itself, the resulting action, and the environment's response. No interpretation. No narrative. Just the structure of a single step.

A trajectory, the concept Chapter 1 introduced, is a sequence of frames. Each frame follows from the one before it: the feedback from frame n becomes part of the observation in frame $n + 1$. The trajectory is the

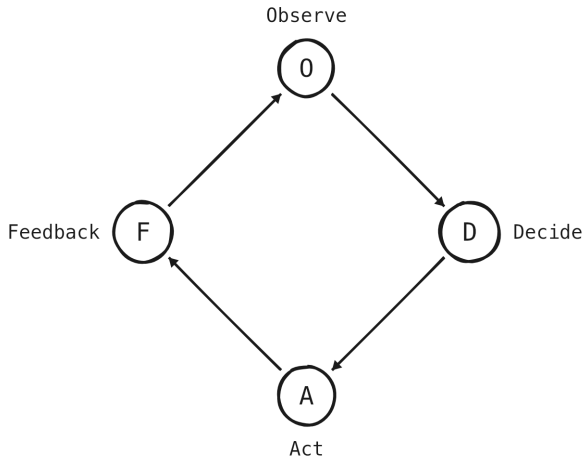

Figure 2.2: The frame: observe, decide, act, feedback

complete record of an agent's behavior across a session, decomposed into its atomic units.

Notice what the frame does not contain. No assessment of whether the decision was good. No label for whether the action succeeded. No interpretation of why the agent chose this path over another. The frame is pre-evaluative. It records structure, not quality. Judgment comes later, and it comes from outside the frame.

This separation is deliberate. The human failure modes we have been cataloguing all involve premature evaluation: judging output before understanding it, projecting intent where there is only prediction, letting emotional responses masquerade as technical assessments. The frame resists this by refusing to carry evaluation in its structure. It records what happened. What it means is a separate question.

Definition precedes measurement. Before you can measure anything about agent behavior, you need to define what you are observing. The frame gives us that definition. It takes the messy, heterogeneous reality of agent interaction and imposes a structure that makes comparison possible. Two

frames from different agents, different tasks, different domains can be compared because they share the same shape.

But the components of a frame are wildly different in kind. An observation might be a thousand lines of source code. A decision might be a natural language plan. An action might be a shell command. Feedback might be a stack trace. Shared structure is necessary but not sufficient. To make frames measurable, to place them in a space where distance and direction mean something, we need a way to represent these heterogeneous components in a common coordinate system.

Chapter 3

The Embedding Problem

An agent reads a Python file. Then it writes a SQL query. Then it runs a shell command and reads the output.

Three actions. Three completely different kinds of thing.

The Python file is structured text with syntax and semantics. The SQL query is a declarative statement in a formal language. The shell output is a raw dump: maybe a stack trace, maybe a list of files, maybe a wall of JSON. These objects live in different representation spaces. They obey different rules. They have different grammars, different structures, different relationships to the world.

A human watching this session feels like it hangs together. You follow the agent's reasoning. You see why it read that file, why it wrote that query, why it ran that command. The session has a coherence that you recognize immediately.

But that feeling is exactly the kind of intuition Chapter 2 warned us about. We project understanding onto sequences of plausible output. We fill gaps with our own context. The coherence we perceive may be real, or it may be a story we are telling ourselves. If we want to know whether this session is actually coherent, whether the agent is making progress or drifting, we cannot rely on how it feels to watch. We need a way to represent coherence mathematically.

Chapter 2 ended with a structure: the frame, (o, d, a, f). Observation, decision, action, feedback. A trajectory is a sequence of these frames. But as Chapter 2 noted, the components of each frame are wildly heterogeneous. They live in different representation spaces. To trace a trajectory through them, to measure distance and direction, we need a shared coordinate system.

That is what this chapter is about. We need coordinates that do not depend on human intuition. Coordinates that make the relationships between frames computable, not felt.

The good news: this problem has been solved. Not by agent researchers. By linguists, by graph theorists, by the machine learning community working on representation learning. The technique is called embedding, and it is the single most important enabling technology for everything that follows in this book.

Maps and Coordinates

Start with something physical.

A city is a discrete entity. It has a name. You can list cities: Paris, Tokyo, Denver, Nairobi. You can talk about them, compare them, say one is larger than another. But you cannot, from the names alone, answer the question: which two cities are closest to each other?

Names do not carry distance information. “Paris” and “Lyon” are both six characters long. “Paris” and “Perth” differ by two characters. None of this tells you anything about geography.

Latitude and longitude change everything. Once every city has coordinates, the question “which two are closest?” is computable. You can calculate distance, draw paths, and compare any pair, even pairs nobody has thought to compare, because the coordinate system gives every city a position in a shared continuous space.

The key move is the transition from *discrete names* to *continuous coordinates*. Before coordinates, each city is an island. You can compare two cities only if someone has already measured the distance between that specific pair. After coordinates, every city is a point on a map. Distance is available for free, between any pair, including pairs nobody has thought to compare.

Embeddings do the same thing for non-physical objects.

An embedding is a function that takes a discrete thing and assigns it a position in continuous space. The position is represented as a vector: a list of numbers. Each number is a coordinate along one dimension, just

like latitude and longitude are coordinates along two dimensions. Except embeddings typically use many more dimensions. A hundred. Three hundred. A thousand. The principle is the same.

Once you have assigned continuous coordinates to discrete objects, distance becomes meaningful. Two objects that are "close" in the embedding space are similar in some way. Two objects that are far apart are dissimilar. The exact nature of the similarity depends on how the embedding was constructed. But the operation is always the same: turn discrete things into points, so that geometry becomes available.

This is interesting precisely because it captures relationships that humans already sense but cannot formalize. You know that "Paris" and "Lyon" are related. You know that a Python sort function and a Rust sort function are doing the same thing. You know that two files in the same subsystem are connected. You have intuitions about all of these relationships. Embeddings turn those intuitions into geometry. Measurable geometry. Geometry you do not have to trust your gut about.

This is the technology that lets you put a Python file and a SQL query on the same map.

Words as Points

The idea of embedding discrete symbols into continuous space has a specific intellectual history, and it starts with words.

In 1957, the British linguist J.R. Firth wrote a line that would become one of the most cited sentences in computational linguistics: "You shall know a word by the company it keeps."

This is the distributional hypothesis. Words that appear in similar contexts tend to have similar meanings. "Dog" and "cat" appear near words like "pet," "feed," "veterinarian." "Download" and "install" appear near words like "software," "file," "computer." You never need to define what "dog" means. You just observe where it shows up. The pattern of co-occurrence *is* the meaning, or at least a useful proxy for it.

Notice the dual nature of this idea. The distributional hypothesis is formalizing something humans already do. You know what "dog" means because of the contexts where you have encountered it. Firth's insight was

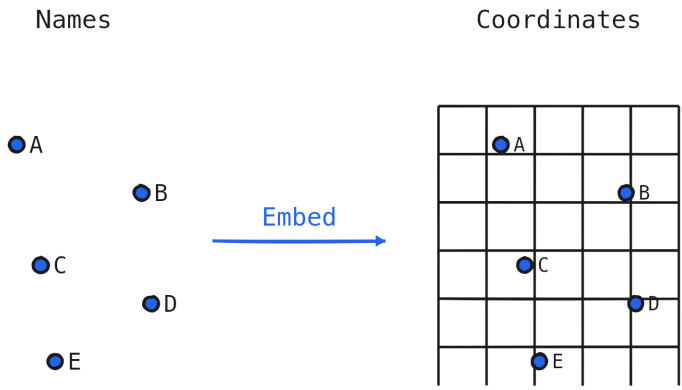

Figure 3.1: Discrete cities become points in continuous coordinate space

that this intuitive process can be made statistical. The gap between what we sense and what we can compute starts to close here.

For decades, the distributional hypothesis was an interesting linguistic observation without a practical computational form. Then, in 2013, a team at Google led by Tomas Mikolov published Word2Vec, and the observation became a technology.

Word2Vec is a neural network trained on a simple task: given a word, predict the words that tend to appear nearby. The training data is just text. Lots of text. Wikipedia articles, news stories, books, web pages. The network reads through this text and tries to learn: when I see “bank,” which words are likely to be within five positions of it? When I see “river,” which words? When I see “deposit”?

After training, you throw away the prediction machinery. What you keep is the internal representation the network learned for each word. That representation is a vector: a list of numbers. If the representation has 300 dimensions, each word gets a 300-dimensional vector.

These vectors are the word embeddings. And they have remarkable properties.

Words with similar meanings end up with similar vectors. "King" and "queen" are close together. "Dog" and "cat" are close together. This follows from the distributional hypothesis: similar contexts produce similar representations.

What is less expected is the geometry. Take the vector for "king," subtract "man," add "woman." The resulting vector is closest to "queen." Nobody programmed this. The relationship emerged from patterns of co-occurrence in billions of words of text.

The vector space encodes not just similarity but *relational structure*. Countries have a consistent direction. Verb tenses have a direction. These directions are not axes of the space, but they are stable: once you find a relational direction, you can apply it to other pairs and get sensible results.

This is where something shifts. Human intuition tells us that kings and queens are related, that Paris is to France as Rome is to Italy. We sense these relationships. But sensing them and computing them are different things. Word2Vec did not encode human intuition. It discovered structure in statistical patterns that *aligns* with human intuition. The embedding space is not a map of what we think. It is a map of what the data contains. That the two often agree is reassuring. That they sometimes diverge is informative.

From Words to Everything Else

Word2Vec embeds individual words. But agents do not work with individual words. They work with sentences, paragraphs, files, functions, error messages, test outputs. Can the same idea scale?

Yes. And it has, in several directions at once.

Sentence embeddings. Instead of assigning a vector to each word, assign a vector to each sentence. The training task changes, but the output is the same: a fixed-length vector for any sentence you feed in. Similar sentences end up near each other.

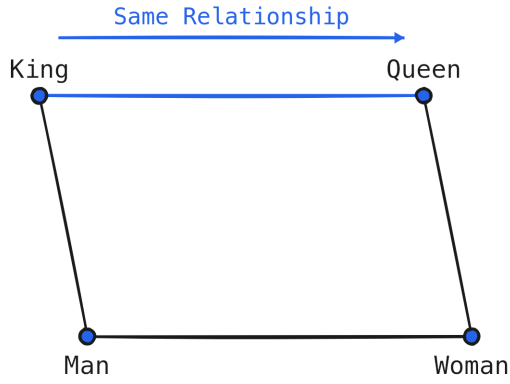

Figure 3.2: The king-queen-man-woman parallelogram in word embedding space

"The server crashed because of a null pointer" and "A null reference caused the server to fail" will have similar vectors, even though they share only a few words. The embedding captures the meaning, not the surface form.

Modern sentence embedding models, built on transformer architectures and trained on large datasets, are good enough to power practical applications. Semantic search is the canonical example: embed every document in a collection, embed a query, find the documents whose vectors are closest to the query vector. The query does not need to share keywords with the document. It just needs to mean something similar.

Code embeddings. The same principle works for source code. Train a model on large quantities of code, and it learns to assign vectors that capture functional similarity. Two functions that sort a list will have similar vectors even if one is written in Python and the other in Rust. Two functions that parse JSON will cluster together regardless of language.

This is a stronger claim than it might seem. Python and Rust share almost no syntax. Variable names differ. Control flow constructs differ. The surface form of the code is completely different. But the *meaning*, what

it computes, what inputs it takes, what outputs it produces, is the same. Code embedding models learn to see through the syntax to the semantics.

Multi-modal embeddings. Models that can embed text, code, images, and other modalities into the same vector space. A diagram of a neural network architecture and a paragraph describing that architecture end up near each other, even though one is pixels and the other is words.

The pattern is consistent. The embedding idea that started with individual words has scaled to sentences, paragraphs, documents, code, images, and combinations thereof. The dimensionality varies. The training tasks vary. The models vary. But the result is always the same: discrete objects get continuous coordinates. Distance becomes meaningful. The map exists.

Code Has Structure That Text Does Not

There is one more kind of embedding that matters for agent trajectories, and it requires a detour through graphs.

When you read a novel, the text is a sequence. Word follows word, sentence follows sentence. There is narrative structure, but the physical representation is linear. Code is different. Code has structure that goes beyond the sequence of characters.

Consider a Python file called `api.py`. It imports `flask`. It imports a function called `authenticate` from `auth.py`. It defines a function `get_user` that calls `db.query` from `database.py`. This file exists in a web of relationships:

It depends on an external framework, two local modules, and calls functions from a database layer. Every file in a codebase participates in a web like this. Imports, function calls, class inheritance, and module boundaries create a network of relationships. The full picture is a *graph*: a set of nodes (files, functions, classes) connected by edges (imports, calls, inheritance, containment).

This graph carries information that the text alone does not. Two files might share zero lines of code but be tightly coupled because they both depend on the same internal module. A function might be trivially short but structurally critical because it sits on the path between two large subsystems.

The dependency graph reveals relationships that you cannot see by reading files in isolation.

Text embeddings capture what a file *says*. Graph embeddings capture where a file *sits* in the larger structure.

Graphs as Geometry

Graphs are combinatorial objects. A node is either connected to another node or it is not. There is no notion of "slightly connected." The structure is discrete.

But the same move that worked for words works for graphs. You can embed nodes into continuous space so that the geometry of the space reflects the structure of the graph.

The intuition: two nodes that have similar neighborhoods in the graph should end up near each other in the embedding. If `api.py` and `routes.py` both import `flask`, both import `auth.py`, and both are imported by `app.py`, they have similar structural roles. They should have similar vectors.

One family of techniques for this, called Node2Vec, works by analogy to Word2Vec. Word2Vec learns word embeddings by observing which words appear near each other in text. Node2Vec learns node embeddings by observing which nodes appear near each other in random walks on the graph.

A random walk is what it sounds like. Start at a node. Pick a random neighbor. Move to it. Pick a random neighbor of the new node. Move. Keep going for some number of steps. The result is a sequence of nodes, much like a sentence is a sequence of words.

Generate thousands of these random walks from every node in the graph. Now you have a corpus of "sentences" where the "words" are nodes. Feed this corpus to the same kind of training procedure that Word2Vec uses. The result: each node gets a vector. Nodes that frequently appear near each other in random walks end up with similar vectors. And nodes that appear near each other in random walks are exactly the nodes that are structurally close in the graph.

A more recent family of techniques, Graph Neural Networks (GNNs), takes a different approach. Instead of random walks, a GNN propagates infor-

mation along edges. Each node starts with some initial feature (maybe a text embedding of the file's contents). Then, in a series of rounds, each node aggregates information from its neighbors. After one round, each node knows about its immediate neighbors. After two rounds, it knows about neighbors of neighbors. After k rounds, each node's representation reflects the structure of its k-hop neighborhood.

The details differ, but the result is the same as Node2Vec: each node in the graph gets a vector, and the vector captures the node's structural context. Files that play similar roles in the codebase, that sit in similar positions in the dependency web, end up near each other.

The graph embedding does not replace the text embedding. They capture different things. The text embedding of `api.py` reflects what the file contains: HTTP route handlers, request parsing, response formatting. The graph embedding of `api.py` reflects where the file sits: between the web framework and the database layer, adjacent to authentication, central to the request-handling subsystem. An agent that reads `api.py` is not just reading text. It is reading text that occupies a specific position in the dependency structure of a codebase. The text tells you what the agent is looking at. The graph tells you where that thing sits in the larger architecture.

We now have three kinds of embeddings on the table.

Text embeddings capture what something *says*: the content of a passage, the meaning of a sentence.

Code embeddings capture what something *computes*: the function of a piece of code, regardless of language or syntax.

Graph embeddings capture where something *sits*: the structural position of a node in a network.

An agent navigating a codebase encounters all three kinds of information simultaneously. It reads file contents (text/code). It follows imports and function calls (graph structure). It observes the output of commands (text). A complete picture of "where the agent is" at any given moment needs to account for all three.

Agent State as Coordinates

Now the payoff. Everything in Part I has been building to this.

Chapter 1 said that an agent session is a trajectory, a path through some space, and that we are looking at projections of that path. We see logs, we see outputs, but not the path itself. Chapter 2 said that our intuitions about those projections are unreliable. We anthropomorphize. We misjudge quality. We project understanding onto statistical prediction. And then Chapter 2 gave us the frame: (o, d, a, f), a pre-evaluative record of a single step in the loop.

The frame solves the recording problem. What it does not solve is the measurement problem. The components of a frame live in different representation spaces. You cannot compute the distance between an observation that is a thousand lines of Python and a decision that is a natural language plan. Not without first placing them in a shared space.

Embeddings are that shared space.

Each component of a frame can be embedded. The observation o, whatever the agent saw, gets a vector. If the agent read a file, that file has both a text embedding (what it contains) and a graph embedding (where it sits in the codebase). The decision d, what the agent chose to do, gets a vector. The action a, the concrete effect on the environment, gets a vector. The feedback f, what came back, gets a vector.

The simplest approach to combining these is concatenation: stack the text embedding and the graph embedding into a single, longer vector. The first coordinates encode what the agent is looking at; the rest encode where that thing sits in the codebase. More sophisticated combination strategies exist, but the principle is always the same: multiple embedding types merge into a single representation in a single space where every agent state has a position and distance is defined.

At the next tick, the agent does something new. Reads another file. Runs a command. Makes an edit. A new frame. A new set of embeddings. A new point in the same space.

Connect the points, and you have a trajectory. Not the metaphorical trajectory of Chapter 1. Not a narrative about phases and pivots. A literal

path through a coordinate space, defined by a sequence of points, each point computed from the agent's state at that moment.

Two sessions that read the same files in the same order will trace similar paths. Two sessions that take different approaches will trace different paths, even if they end at the same place. The geometry of the path captures the behavior: the orientation phase, the investigation, the commitment, the correction, the convergence. All the structure that Chapter 1 described in narrative form now has coordinates.

The three-part problem is solved, at least in principle. We know what to record (frames). We know not to trust our feelings about what the recording shows (the partnership problem). And now we have a coordinate system that makes measurement possible without relying on either projection or intuition.

What Distance Means

Once agent states have coordinates, distance becomes meaningful. But what does distance *mean* in this space?

Two states are close if they involve similar content, similar structural context, and similar accumulated knowledge. An agent reading `auth.py` after having read `session.py` and `user.py` is in a different position than an agent reading `auth.py` as its first action with an empty context. The file is the same. The state is different, because the accumulated context is different. The embedding of the full state captures this.

Distance between consecutive frames in a trajectory measures how much the agent's situation changed in one step. Small distance: the agent did something incremental, read a closely related file, made a minor edit. Large distance: the agent pivoted, switched from backend to frontend, or abandoned one hypothesis for another.

The sequence of these distances, step sizes along the trajectory, already tells you something about the session's dynamics. A trajectory with small, consistent step sizes suggests methodical exploration. A trajectory with sudden large jumps suggests pivots or confusion. A trajectory where step sizes increase over time suggests the agent is casting an increasingly wide net, possibly lost.

None of this requires understanding what the agent was thinking. It falls directly out of the geometry.

Here is where the three forces from Chapter 1 start to become measurable.

Inertia shows up as small, consistent distances in the same direction. The agent is making steady progress along a coherent path. Each frame is close to the last one, and the direction of movement is stable. The trajectory is smooth.

Entropy shows up as distances becoming erratic. The trajectory loses coherence. Step sizes vary wildly. Direction shifts unpredictably. The agent is thrashing, or exploring without a plan, or losing its grip on the problem.

Space is the structure of the embedding itself. The shape of the coordinate system determines which paths are natural and which require sharp turns. Some regions of the space are dense with related concepts. Others are sparse. The geometry of the space constrains what trajectories look like, just as the topology of a mountain range constrains which paths a hiker can take.

These three forces were metaphors in Chapter 1. They are about to become mathematics. Not here. Parts II and III will develop the full formalism. But the connection should be clear: the measurable properties of trajectories in embedding space correspond to the dynamics we described intuitively.

These embedding spaces are high-dimensional. Not two dimensions like a map. Not three like physical space. Hundreds of dimensions. Sometimes thousands. Human spatial intuition does not survive past three dimensions. You cannot visualize a path through 896-dimensional space.

But high dimensionality is what makes embeddings work. A two-dimensional map can represent cities by location, but it cannot simultaneously encode their populations, industries, languages, and climates. It does not have enough dimensions. A 768-dimensional space can encode all of these properties simultaneously, in a single set of coordinates. With enough dimensions, you can arrange objects so that every relevant relationship between them is captured by some direction in the space. Semantic similarity occupies some dimensions. Structural role occupies others. Syn-

tactic patterns occupy still others. These different kinds of information coexist without interfering.

For agent trajectories, this is an asset. An agent's state involves many simultaneous factors: what it is reading, what it has already read, where it is in the codebase, what hypothesis it is pursuing, how much context it has accumulated. A low-dimensional space would force tradeoffs. A high-dimensional space can represent them all.

We lose the ability to visualize. We gain the ability to represent. Techniques like t-SNE, UMAP, and PCA can project high-dimensional trajectories into two or three dimensions for visualization, but the projections lose information. The full-dimensional representation is where real analysis happens.

Part II formalizes what we have assembled here, giving inertia, entropy, and space precise definitions and measurable consequences in this coordinate system.

Part II

A New Coordinate System

Chapter 4

Frames, Not Turns

A chess game is recorded in algebraic notation. 1. e4 e5 2. Nf3 Nc6 3. Bb5 a6. Each entry is a move: what changed. White's pawn went to e4. Black's knight went to c6. The notation tells you what each player did, and from the sequence of moves you can reconstruct what happened.

But a move is not a position.

A photograph of a chessboard captures something different. Not what changed, but everything that is true at a single instant. All thirty-two pieces (or however many remain), their locations, the pattern they form, the tension between them. A photograph includes the move that just happened, but it also includes the entire context surrounding that move. The pawn structure. The open files. The king's exposure. The bishop aimed at the corner that nobody has mentioned yet but that will matter in six moves.

A move is an action. A photograph is a state.

Chapter 1 argued that agent turns are frames — samples of continuous motion, like individual photographs in a film reel. This chapter makes that idea concrete. What does a frame actually contain? What are the dimensions of that photograph? And can those dimensions be formalized into something we can compute with?

The answer to the last question is yes. The formalization produces a coordinate system: a set of independent dimensions that together specify an agent's position in context space at any given moment. Once we have coordinates, we can define distance. Once we have distance, we can define paths. Once we have paths, we can measure motion.

That is where we are headed. But first we need to understand what goes into a single frame.

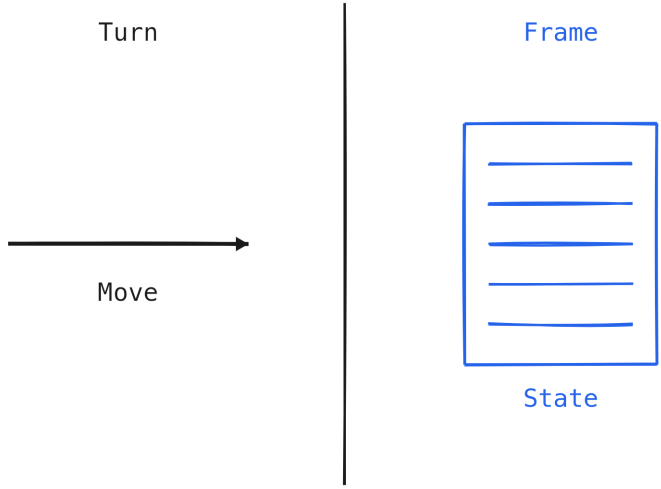

Figure 4.1: A chess move records what changed; a photograph captures the full board state

Anatomy of a Frame

Return to the example that has been running since Chapter 1. An agent is tasked with fixing `test_session_expiry`. The test expects `session.expires_at` to be a timestamp, but the value is `None`. The agent has been working for six turns.

At Turn 6, the agent decides to modify `create_session` to fall back to the configured default TTL. It writes the edit. That is the move. That is what the telemetry log records: `edit_file("session.py", ...)`.

The frame at Turn 6 contains far more than the edit.

What was read. By Turn 6, the agent has consumed the failing test, the error output, `session.py`, the config fixture, and a second look at the test. These files are in the agent's context. Their content informs the decision. The frame carries all of them, along with metadata about when each was read and in what order.

What was written. The edit itself: the specific lines added to `create_session`, the function signature it modified, the fallback logic it introduced. Also any

reasoning the agent produced (chain-of-thought text, plan statements, commentary on why it chose this approach over alternatives).

Position in the codebase. The agent is currently operating in a specific region of the project. It has touched `session.py`, `user_auth_test.py`, and `config.yaml`. These files have relationships: imports, shared data structures, test-to-source mappings. The agent sits at a particular node in the codebase's dependency graph.

Semantic state. What is the agent thinking about? Not in the interpretability sense of "what are the attention heads doing," but in the practical sense of "what topic is active in the context?" At Turn 6, the agent is thinking about session expiry, default configuration values, and the `create_session` function signature. That semantic content could be captured as an embedding of the recent context.

Resources consumed. The agent has used a certain number of input tokens, output tokens, and wall-clock seconds. It has a budget (explicit or implicit), and some fraction of that budget is gone. The remaining runway shapes what the agent can still afford to do.

Each of these components is present at every turn. Together, they define the agent's full state at a moment in time. A turn log captures one component (what was written). A frame captures all five.

The difference matters for the same reason that a chess photograph matters more than a chess move: the move tells you what changed, but the photograph tells you what is possible next. Understanding an agent's behavior requires knowing not just what it did, but the full context in which it chose to do it.

Five Coordinate Dimensions

The components of a frame can be organized into five independent dimensions. Independent in the sense that knowing one does not tell you the others. An agent can be deep in one part of the codebase (topology) while thinking about an unrelated concept (semantics), burning tokens fast (budget), mostly reading (behavior), and visiting new files for the first time (repetition). Each dimension varies on its own axis.

These five dimensions form the coordinate system for context space.

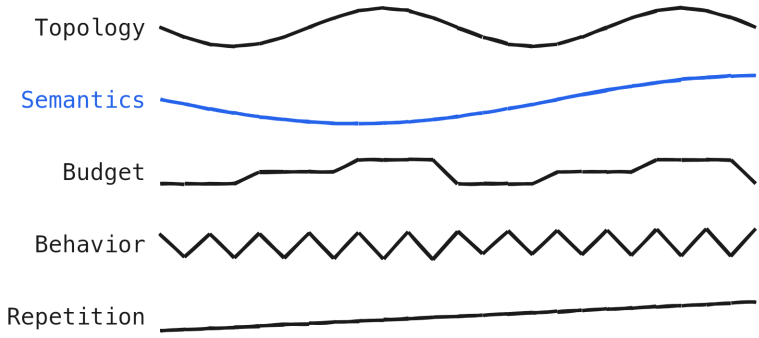

Figure 4.2: Five coordinate dimensions as parallel signal lines: topology, semantics, budget, behavior, repetition

Topology

Where is the agent in the structure of the project?

A codebase is a graph. Files import other files. Functions call other functions. Tests reference source modules. Configuration files are read by initialization routines. This graph has topology: clusters of closely related files, distant modules that rarely interact, central nodes that everything depends on, peripheral files that nothing references.

At any given turn, the agent occupies a position in this graph. The files it has recently read or modified form a neighborhood. That neighborhood might be tight (three files in the same module, all importing from each other) or scattered (five files spread across four different packages with no shared dependencies).

Consider two agents, both six turns in. One has read three files in the same database module, all importing from each other — a tight cluster with high internal connectivity. The other has read five files spread across

five unrelated packages. The first agent's topological position is compact. The second's is scattered.

The topology coordinate does not judge which is better. Sometimes a problem requires reading across many modules. Sometimes it requires deep focus on one. The coordinate records the fact: here is where the agent is in the structure. The interpretation comes later.

Topological position can be quantified. One straightforward measure: compute the average pairwise distance (in the dependency graph) between all files the agent has touched in the last N turns. Small average distance means a compact position. Large average distance means a scattered one. More sophisticated measures can account for the directionality of dependencies, the centrality of the files involved, or the depth of the agent's position in the module hierarchy (editing a leaf function versus editing a core utility that forty other modules import).

Semantics

What is the agent thinking about?

Topology tells you which files the agent is touching. Semantics tells you what those files mean, and more precisely, what the agent's current context means when considered as a whole.

Two agents can occupy the same topological position (same files read, same region of the codebase) while operating in entirely different semantic spaces. One is focused on the authentication logic in those files. The other is focused on the error-handling patterns. They are reading the same code, but attending to different aspects of it, and their subsequent actions will diverge based on that difference.

Semantic position can be estimated by embedding the agent's recent context (the last few turns of observations, actions, and reasoning) into a vector space. Language model embeddings are designed to capture meaning. A context window full of authentication-related discussion will embed near other authentication discussions and far from, say, database query optimization discussions.

Micro-example. An agent has spent four turns reading files related to user session management. Its recent context includes the words "session,"

"expiry," "token refresh," "TTL," and "cookie." The embedding of this context lands in a particular region of the semantic space. If the agent suddenly starts reading files about database indexing — different vocabulary, different concepts, different purpose, its semantic position shifts. That shift is measurable as the distance between successive context embeddings.

Semantic stability (low movement in the embedding space between turns) correlates with focused, purposeful work. Semantic drift (high movement) can indicate that the agent is exploring broadly or, less charitably, losing the thread.

Budget

What has the agent spent, and what remains?

Every agent session has resource constraints. Tokens cost money. Time costs patience. Context windows have fixed sizes. These constraints are not abstract. They shape behavior.

The budget dimension tracks the consumption curve: how much of each resource has been used, at what rate, and how much runway remains. This is the most directly measurable of the five dimensions. Token counts are exact. Wall-clock time is exact. Context window utilization (how much of the maximum context has been filled) is exact.

What makes budget a coordinate dimension rather than just a metric is that it interacts with the other dimensions in non-obvious ways. An agent at 20% of its token budget behaves differently from the same agent at 80% of its budget, even if everything else is identical. The remaining runway constrains what strategies are feasible. An agent with ample budget can afford to explore broadly (high-entropy behavior, scattered topology). An agent near its limit must be efficient (low-entropy behavior, targeted topology) or risk exhausting resources before completing the task.

Micro-example. Two agents, identical in every way, start the same task. Agent A has a 100,000-token budget. Agent B has a 20,000-token budget. Agent A reads six files before committing to a plan. Agent B reads two. Agent A's exploration phase is broader. Agent B's is more targeted (or more desperate). The budget coordinate explains the difference in behavior that the other four coordinates cannot.

The shape of the consumption curve also carries information. Linear consumption (steady tokens per turn) suggests a consistent strategy. Accelerating consumption (more tokens per turn as the session progresses) suggests the agent is reading larger files or producing longer outputs, possibly because the problem is getting harder or the context is getting more complex. Decelerating consumption (fewer tokens per turn) can indicate convergence, that the agent is narrowing its actions, producing smaller and more precise edits.

Behavior

What is the agent doing?

This is the action distribution. At any given turn, the agent takes an action: read a file, edit a file, run a test, search the codebase, execute a shell command. Over a window of recent turns, the distribution over these action types characterizes the agent's behavioral mode.

The behavior dimension captures not just which actions the agent takes, but their proportions and transitions. An agent that alternates between reading and editing in a steady rhythm has a different behavioral signature than an agent that reads ten files in succession, then edits five in succession, then runs three tests. The former is interleaving investigation and implementation. The latter is batching phases.

Micro-example. Over the last five turns, an agent's actions were: read, read, edit, test, edit. The action distribution is: read 40%, edit 40%, test 20%. This is a balanced distribution with moderate entropy, suggesting the agent is actively implementing and verifying.

A different agent over the same window: read, read, read, read, read. Distribution: read 100%, everything else 0%. Very low entropy. The agent is in pure investigation mode (or stuck in a read loop; the behavior coordinate alone cannot distinguish these, which is why we need all five dimensions).

Behavioral transitions carry information that the static distribution misses. Read-then-edit is a qualitatively different pattern from edit-then-read. The first suggests the agent is learning before acting. The second suggests the agent is acting before checking. Transition probabilities (given the agent just read a file, what is the probability the next action is an edit versus another read?) form a compact summary of the agent's operational style.

Repetition

Has the agent been here before?

The repetition dimension tracks novelty versus revisitation. At each turn, is the agent encountering new material or returning to material it has already seen?

This is distinct from topology (where the agent is) and semantics (what the agent is thinking about). An agent can be in the same topological region and the same semantic space on Turn 20 as it was on Turn 5, but the repetition coordinate differs: on Turn 5, those files were new. On Turn 20, they are being revisited.

From an information-theoretic perspective, re-reading a file carries less information than reading it for the first time, unless the agent is reading with a different question. The repetition coordinate captures the raw fact of how much of the agent's current activity involves previously-encountered material.

Micro-example. An agent reads `auth.py` at Turn 2 and again at Turn 14. The second read is a revisit. If between those turns the agent also read twelve other files, each for the first time, then the repetition dimension has been low (mostly novel material) and just spiked briefly. If instead the agent has been cycling through the same three files for twelve turns, the repetition dimension has been high throughout.

Quantifying repetition is straightforward. For each file read or edited in a turn, check whether the agent has previously encountered that file during this session. Compute the fraction of current-turn activity that involves previously-seen files. This fraction ranges from 0 (all new material) to 1 (all revisits).

Windowed repetition (what fraction of files in the last N turns are revisits) smooths out individual-turn noise and gives a stable measure of whether the agent is in exploration mode (low repetition, encountering new material) or consolidation mode (high repetition, working with known material).

High repetition is not inherently bad. A carpenter measures twice and cuts once. An agent that re-reads a complex function before editing it may be exercising appropriate caution. But high repetition combined with low behavioral diversity (reading the same files without transitioning to edits)

and low mutual information (the re-reads are not followed by informed actions) is a strong signal of unproductive looping. The coordinate records the fact. The diagnosis requires combining it with the other dimensions.

Putting the Coordinates Together

Each dimension captures something the others miss. Topology without semantics tells you where the agent is but not what it is thinking. Semantics without behavior tells you what the agent is thinking but not what it is doing about it. Behavior without budget tells you what the agent is doing but not whether it can afford to keep doing it. Repetition without everything else tells you whether the agent has been here before but not whether coming back was worthwhile.

Together, the five dimensions define a point in context space. A single frame, one turn of agent execution, maps to a single point in five-dimensional space:

(topology, semantics, budget, behavior, repetition)

This is the coordinate system. Each turn produces a new point. A session produces a sequence of points. That sequence is a path through context space. Chapter 5 will formalize that path as a trajectory.

Before getting there, we need to address a question that might be nagging: are these dimensions actually useful? Can they be computed from real agent sessions? Do they tell us things that simpler metrics do not?

The answer to all three is yes. And the way to demonstrate it is to move from abstract dimensions to concrete, computable features.

Computable Features

The five coordinate dimensions are not philosophical categories. They decompose into specific numerical features that can be extracted from standard agent logs. No model internals required. No attention-head inspection. Just the observable record of what the agent read, wrote, and did.

Here are four features, one drawn from each of several dimensions, that illustrate the kind of measurement this coordinate system enables.

Read Fingerprint Stability

Dimension: topology + repetition.

At each turn, the agent has a "read fingerprint" — the set of files it has read in the last N turns. Between successive turns, this fingerprint changes: a new file enters, an old one drops out of the window, or the set stays the same.

Read fingerprint stability is the Jaccard similarity between the read sets of consecutive windows. Jaccard similarity is the size of the intersection divided by the size of the union, a number between 0 (completely different sets) and 1 (identical sets).

High stability means the agent is working with the same files turn after turn. The fingerprint is not changing. The agent is anchored.

Low stability means the read set is churning. Different files every few turns. The agent is moving through the codebase.

Moderate stability with periodic drops corresponds to the natural rhythm of productive work: the agent settles into a region, works there for a while (high stability), then moves to a new region (stability drops), settles in again (stability rises). The pattern looks like a plateau interrupted by step changes.

An agent that never achieves high stability — perpetual churn, new files every turn, never settling — is exhibiting a specific pathology: inability to commit to a region of the codebase. An agent whose stability is high from Turn 1 and never drops may be exhibiting a different pathology: tunnel vision, inability to explore beyond the first files it encountered.

The shape of the stability curve over a session is a diagnostic. Not the absolute value at any moment, but the temporal pattern.

File Scatter

Dimension: topology.

File scatter measures how spread out the agent's activity is across the codebase's structure. Take all files touched (read or written) in the last N turns and compute a dispersion metric over the dependency graph.

One approach: map each file to its directory path and compute the depth of the least common ancestor across all active files. If all files are in `src/auth/`, the least common ancestor is `src/auth/` and the scatter is low. If files span `src/auth/`, `src/db/`, `tests/integration/`, and `config/`, the least common ancestor is the project root and the scatter is high.

A graph-based approach goes further. If the codebase's dependency graph is available (imports, function calls, type references), file scatter can be computed as the average shortest-path distance between all pairs of active files. Two files in the same module that import each other have distance 1. Two files in unrelated packages with no shared dependencies might have distance 5 or more.

File scatter correlates with task complexity. Simple bugs in a single module produce low scatter: the agent reads and edits files in a tight cluster. Architectural issues that cut across module boundaries produce high scatter: the agent must touch files in many different regions.

File scatter also correlates with agent confusion. An agent that should be working on a localized bug but exhibits high scatter may be searching too broadly — reading files that are structurally distant from the problem. The topology is telling you something about the agent's strategy (or lack thereof).

Tool Diversity

Dimension: behavior.

Tool diversity is the entropy of the action-type distribution over a sliding window. The interpretation in this coordinate framework is specific: tool diversity is the behavioral coordinate's measure of how many different types of actions the agent is engaging in.

High tool diversity (high entropy) means the agent is spreading its actions across reading, editing, searching, testing, and executing. It is engaging with the problem through multiple modes of interaction.

Low tool diversity (low entropy) means the agent is concentrated on one or two action types. It is in a specialized mode: pure investigation (all reads), pure implementation (all edits), or pure verification (all test runs).

The expected pattern in a healthy session: tool diversity starts moderate (the agent reads and searches during orientation), rises slightly during model formation (adding test runs and targeted searches), drops during committed editing (concentrated on edits and tests), and may rise again if a problem is encountered (re-exploration).

Flat tool diversity throughout a session — the same mix of actions from beginning to end, never narrowing, never concentrating, suggests the agent is not transitioning between cognitive phases. It is not moving from exploration to commitment. It is stuck in a single mode, recycling the same proportions of reads and edits without strategic evolution.

Budget Curve Shape

Dimension: budget.

Plot cumulative token consumption against turn number. The resulting curve has a shape, and that shape is informative.

A straight line (linear consumption) means the agent is using roughly the same number of tokens per turn. Each turn involves about the same amount of reading, thinking, and writing. This is a steady, undifferentiated pace.

A concave curve (decelerating consumption) means earlier turns consumed more tokens than later turns. The agent front-loaded its reading — big observations early, smaller targeted actions later. This is the shape of an agent that explored first, then committed to efficient execution. It is generally a healthy pattern.

A convex curve (accelerating consumption) means later turns consume more tokens than earlier turns. The agent's appetite is growing. This can happen when the problem turns out to be harder than expected (the agent reads more files, generates longer plans, produces longer edits as it grapples with complexity). It can also happen when the context is growing unwieldy and the model needs more tokens to process the accumulated information.

An S-curve (slow start, acceleration, deceleration) describes a three-phase session: orientation (slow), active investigation and implementation (fast), convergence (slow again). This is the most common shape in successful sessions that involve non-trivial problem-solving.

The budget curve shape can be quantified by fitting a polynomial to the cumulative consumption data and examining the coefficients, or more simply by computing the ratio of tokens consumed in the first half of the session to tokens consumed in the second half. A ratio greater than 1 indicates front-loading (concave). A ratio less than 1 indicates back-loading (convex). A ratio near 1 indicates linearity.

These four features are samples. The coordinate system admits many others: semantic drift rate (average distance between consecutive context embeddings), revisitation depth (how many times the most-visited file has been read), action transition entropy (how predictable is the sequence of action types), budget efficiency (task-relevant actions per token spent). Each feature is a projection of the five-dimensional coordinate onto a specific measurement axis.

The point is not to enumerate every possible feature. It is to demonstrate that the five dimensions are not abstract. They decompose into numbers. Those numbers can be computed from logs that already exist. And those numbers tell you things about agent behavior that turn counts and token totals cannot.

What a Single Frame Reveals

Consider a single frame from an agent session. Turn 14 of a 30-turn session. The frame's coordinates:

- **Topology:** compact. Three files in the same module, average graph distance 1.2.
- **Semantics:** stable. The context embedding has moved less than 0.05 (cosine distance) from the previous turn. The agent is thinking about the same thing it was thinking about at Turn 13.
- **Budget:** 55% consumed. Token consumption has been roughly linear. Midpoint of the session.
- **Behavior:** edit-dominant. 60% edits, 30% tests, 10% reads over the last five turns. Low entropy.
- **Repetition:** high. All three active files have been previously read. No new material in the last four turns.

From these coordinates alone — without reading the agent's reasoning, without seeing the actual code, without knowing the task — a picture

emerges. The agent is in a focused implementation phase. It has settled into a tight region of the codebase (compact topology), is working on a consistent problem (stable semantics), is actively building something (edit-dominant behavior), is verifying as it goes (tests in the mix), and is working with material it already understands (high repetition). It is midway through its budget with a steady consumption rate (no acceleration, no signs of distress).

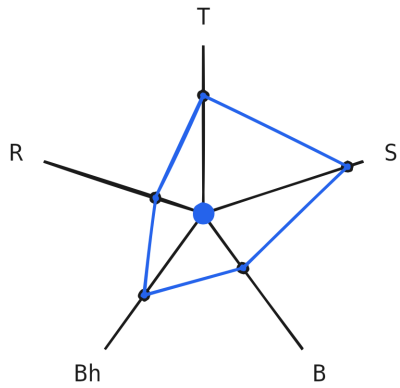

Figure 4.3: A single frame rendered as a radar chart across five dimensions

This is a coordinate. One point in five-dimensional space. On its own, it tells a story. In sequence with the coordinates from Turns 1 through 13, it tells a richer story — one about how the agent arrived at this focused state, how long it took to get here, and (given the budget trajectory) whether it is likely to complete the task with its remaining resources.

Change one coordinate and the picture changes. Same topology, same semantics, same budget, same behavior, but repetition is low — the agent is editing files it has never read. That is a different situation entirely. The agent is writing code in unfamiliar territory. Whether that is bold or reckless depends on what happens next.

Same everything, but the budget is at 90% consumed instead of 55%. The agent is doing the same focused work, but the urgency is different. It has very little runway left. If the current approach fails, there is not enough budget for a strategic pivot. The coordinates are almost the same. The situation is not.

What a Sequence of Frames Reveals

A frame is a coordinate. A sequence of coordinates is a path. That path — the trajectory through context space — is what we actually want to measure.

The next chapter defines trajectories formally: as mathematical objects with computable properties like length, curvature, velocity, and shape. But even informally, the progression is visible.

Frames 1 through 5: scattered topology, drifting semantics, high tool diversity, low repetition. The agent is exploring. It is moving quickly through the codebase, reading broadly, thinking about several different aspects of the problem, and encountering new material at every turn.

Frames 6 through 8: topology compacting, semantics stabilizing, tool diversity falling, repetition rising. The agent is committing. It has found the region of the codebase that matters and is settling in.

Frames 9 through 14: compact topology, stable semantics, edit-dominant behavior, high repetition. The agent is executing. Focused, efficient, working with known material.

Frame 15: sudden spike in tool diversity, topology scatters, semantics shift, repetition drops. Something went wrong. The agent encountered a failure and is re-exploring. A test failed. An import was missing. A type mismatch appeared that the agent did not anticipate.

Frames 16 through 20: gradual recompaction. Topology narrows again. Semantics stabilize in a slightly different location (the agent is now thinking about type conversion, not just session expiry). Behavior returns to edit-dominant. The agent has recovered.

This is a shape. It has rhythm, structure, identifiable phases. And it is entirely defined by the five coordinate dimensions tracked across time.

No access to the model's internal state. No interpretability tools. Just the observable record, organized into a coordinate system that makes the shape visible.

Two sessions with identical turn counts and token totals can have completely different shapes. One shows the clean explore-commit-execute pattern. The other shows perpetual scatter, never compacting, never stabilizing — thirty turns of aimless wandering that happens to arrive at the right answer by chance. The coordinates distinguish them. The telemetry report does not.

A frame is a coordinate. A sequence of coordinates is a path. That path is what we measure. The next chapter defines that path formally.

Chapter 5

Trajectories

A sequence of frames is a trajectory.

Chapter 1 introduced this idea in passing. Chapter 2 defined the agent loop. Chapter 3 showed how embeddings assign coordinates to agent states. Chapter 4 formalized the frame: the snapshot of an agent's state at a single moment.

This chapter is about what happens when you line up the snapshots.

A GPS receiver records your position once per second. Each recording is a point: latitude, longitude, maybe altitude. A single point tells you where you were at that instant. A sequence of points, connected in order, tells you something richer. It tells you where you went. The shape of the path through a city reveals whether you were commuting, lost, or exploring. The path tells a story that no individual point contains.

Agent frames work the same way. Each frame is a point in the coordinate space established in Chapter 3. The sequence of points, connected in order, is a path through that space. That path is the trajectory.

The distinction between "a sequence of frames" and "a trajectory" may seem like word games. It is not. The distinction is between a log and a geometric object. A log is a list. A geometric object has properties: length, direction, curvature, velocity. These properties are computable. They are comparable across sessions. They give us a language for describing what an agent did that goes beyond narrating the events in order.

This chapter defines that language.

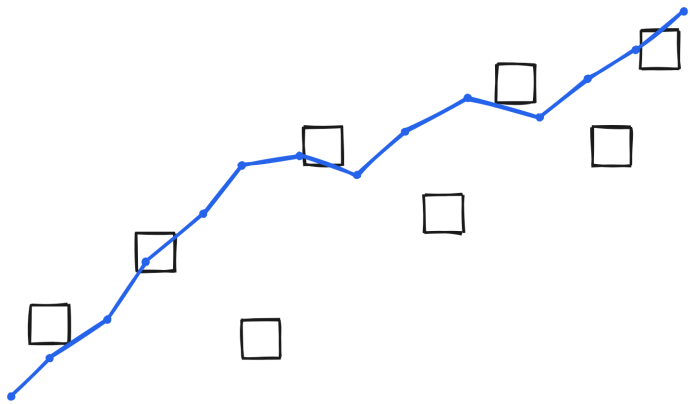

Figure 5.1: A GPS track through a city: individual position recordings connected into a continuous path

Paths, Not Logs

The standard representation of an agent session is a log. Turn 1: read `auth.py`. Turn 2: read `session.py`. Turn 3: edit `session.py`. Turn 4: run tests. Turn 5: read error output. And so on.

A log is a list of events indexed by time. It preserves sequence. It records what happened and when. If you want to know what the agent did at Turn 7, you look up Turn 7 in the log.

A trajectory is a different kind of object. It is a path through continuous space. Each event in the log has been embedded (Chapter 3) into a high-dimensional vector. The sequence of vectors traces a curve. The curve has geometric properties that the log does not.

Consider an analogy. A pilot's radio transcript logs every communication with air traffic control. "Cleared for takeoff." "Turning left heading 270." "Descending to flight level 350." That transcript is a log. The flight path recorded by the aircraft's transponder is a trajectory. Both describe the same flight. The transcript is a sequence of discrete events. The trajectory

is a continuous curve through three-dimensional space. The transcript can tell you what the pilot said. The trajectory can tell you the shape of the approach, the rate of descent, the radius of the turns.

Agent trajectories are analogous. The log tells you what happened. The trajectory tells you the shape of what happened. And shape, it turns out, carries information that the sequence of events does not.

Direction

The first geometric property of a trajectory is direction.

At any point along a path, you can ask: which way is it going? The answer is a vector, computed from the difference between the current position and the recent positions that preceded it. In calculus terms, direction is the first derivative of position with respect to time.

For agent trajectories, direction tells you what the agent is heading toward. Not in a goal-setting sense. In a geometric sense. If the last three frames involved reading files in the authentication subsystem, the trajectory's direction vector points "toward authentication." If the agent then starts reading database configuration files, the direction changes. The trajectory bends.

Direction is computed from the frame embeddings. Take the agent's position vector at frame t and subtract its position vector at frame $t - k$ (where k is a small window, maybe two or three frames). The resulting difference vector is the direction. It points from where the agent was toward where it is.

This is not a metaphor. The embedding space has coordinates. Subtraction is defined. The result is a vector with magnitude and orientation in that space. You can compare direction vectors between two different sessions to ask: are these agents heading the same way? You can track direction over time within a single session to ask: did the agent change course?

The direction vector encodes something that the log alone cannot express concisely. The log says "the agent read a sequence of configuration files." The direction vector says "the agent is moving through the configuration subsystem." The direction vector generalizes across specific file names

to capture the semantic trajectory. Two agents reading different configuration files in different codebases can have the same direction, because the embedding space places configuration-related content in a consistent region.

Velocity

The second property is velocity.

Velocity measures how fast the agent is moving through the space. More precisely: how much does the agent's position change between consecutive frames?

Compute the distance between the embedding at frame *t* and the embedding at frame *t + 1*. That distance is the step size. The sequence of step sizes across the session is the velocity profile.

A large step means the agent's state changed substantially in one turn. It jumped from one region of the space to a distant one. Maybe it switched from reading backend code to examining frontend templates. Maybe it abandoned one hypothesis entirely and started investigating a different module.

A small step means the agent's state changed incrementally. It read a file closely related to the one it just read. It made a minor edit to the file it was already working on. The agent stayed in the same neighborhood of the space.

The velocity profile of a session has a characteristic shape for different kinds of behavior.

Methodical investigation produces moderate, consistent velocity. The agent moves steadily through a region, reading related files, building understanding. Each step covers new ground but not distant ground. The velocity profile is a relatively flat line at a moderate level.

Rapid scanning produces high velocity. The agent is jumping across the codebase, reading files in different subsystems, searching broadly. Each step covers a large distance. The velocity profile spikes.

Deep focus produces low velocity. The agent is working intensively in one small area, re-reading the same few files, making incremental edits. It barely moves in the space. The velocity profile drops near zero.

Thrashing produces erratic velocity. Large jumps, then small steps, then large jumps again. No consistent pattern. The velocity profile oscillates wildly.

These shapes are visible in the data. They do not require interpretation or judgment calls. Compute the step sizes. Plot them. The pattern is there or it is not.

Curvature

The third property is curvature.

Direction tells you where the agent is heading. Velocity tells you how fast it is moving. Curvature tells you how much the direction is changing. It is the second derivative of position: the rate of change of the rate of change.

A straight path has zero curvature. The agent is heading in one direction and continuing that way. In practice, this means the agent has committed to a line of investigation and is following it without deviation. Each frame takes it further along the same heading.

A gently curving path has low curvature. The agent is gradually shifting its focus. Maybe it started in the authentication module and is drifting toward the session management module, following the dependency chain. The direction is changing, but slowly.

A sharp bend has high curvature. The agent's direction changed abruptly. It was heading one way and suddenly went another. In the session trace from Chapter 1, the moment the agent's fix failed and it had to reconsider (Phase 4, Turn 7) would produce a spike in curvature. The agent was heading toward "implement the fix" and suddenly pivoted to "diagnose the new error."

A path that loops back on itself has sustained curvature. The agent is returning to a region it already visited. It read the authentication code, went off to investigate something else, and is now coming back. The path curves around and intersects its earlier segment. This is visible in the geometry even if the log merely shows the same file being read again.

Curvature, like velocity, can be tracked over time to produce a profile. The curvature profile of a session tells you when the agent changed direction and how sharply. Peaks in curvature correspond to pivots, surprises, plan changes. Flat regions correspond to sustained commitment. The profile is a compressed representation of the agent's strategic evolution.

Together, direction, velocity, and curvature form a complete first-order description of the trajectory's local geometry. At any point along the path, these three quantities tell you: where is the agent heading, how fast is it moving, and how much is it turning?

These are not the only geometric properties we could define. Torsion (the rate of change of curvature) and higher derivatives exist and can be computed. But for the purposes of this book, direction, velocity, and curvature are sufficient. They capture the dynamics that matter for understanding agent behavior.

Why the Derivative Matters

Two agents are at the same point in the space. They have just read the same file. Their current frame embeddings are identical (or close enough to be indistinguishable).

Are they in the same state?

No.

Agent A arrived at this file after a systematic traversal of the dependency chain. It read the calling function, followed the import, read the imported module, found a reference to a configuration file, and read that. Its trajectory has been a steady march in one direction, with low curvature and moderate velocity. It is building a model.

Agent B arrived at the same file after a confused search through six unrelated modules. It searched for a keyword, read a false positive, searched again, read another false positive, tried a different search term, and finally stumbled onto this file. Its trajectory has been a zigzag through distant regions of the space, with high curvature and erratic velocity. It is guessing.

Same position. Different trajectories. Different derivatives. Different prognoses.

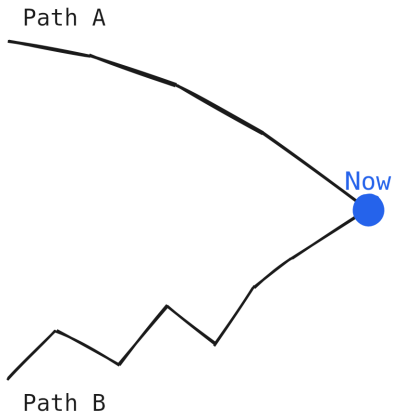

Figure 5.2: Two trajectories arriving at the same point from different directions

Agent A is likely to make productive use of this file, because it arrived with context that makes the file's content meaningful. Agent B is likely to extract less from the file, because it arrived without the surrounding context that gives the file's content significance.

Position alone is insufficient. You need the derivative.

This is the difference between a log and a trajectory. A log records positions. A trajectory records positions *and* their relationships over time. The relationships are the derivatives: direction, velocity, curvature. They encode how the agent arrived at its current state, and that history shapes what happens next.

In physics, this distinction is fundamental. The position of a ball at one instant does not determine its future. The position plus the velocity does (given the forces acting on it). A ball at the top of a hill, stationary, will roll down. The same ball at the same position, moving upward with enough speed, will continue over the hill. Same position. Different velocity. Different outcome.

Agent trajectories obey the same logic. The agent's current frame does not determine what happens next. The current frame plus the trajectory's recent direction and velocity do (given the forces acting on it, which later chapters will define). An agent converging toward a solution and an agent circling in confusion can occupy the same point. Their futures diverge because their derivatives diverge.

The Visual Language

This chapter establishes the conventions for how trajectories will be drawn throughout the rest of the book.

A trajectory is rendered as a path through abstract two-dimensional space. The space is a projection of the high-dimensional embedding space (via PCA, t-SNE, or UMAP), chosen to preserve the gross structure of the path. The projection loses information, as all projections do. It preserves enough for visual intuition.

The path is a continuous curve connecting the frame positions in sequence. Early frames are at one end. Late frames are at the other. Time flows along the path.

Labels. Key moments along the trajectory are annotated with small labels. "Read auth.py." "Test failed." "Hypothesis formed." These labels are sparse. Not every frame gets one. The labels mark the events that matter for understanding the path's shape.

Branching. When an agent considers alternatives (explicitly or implicitly), the trajectory forks. The path splits into two or more branches. The branch the agent actually follows continues as a solid line. Branches not taken are drawn as dashed lines that fade. The fork point is a decision. The visual shows the decision and its alternatives simultaneously.

Degradation. When an agent's behavior becomes incoherent, the path's visual character changes. A coherent trajectory is a clean curve. A degrading trajectory frays. The line becomes thicker, fuzzier, or scatters into a cloud of loosely connected points. This represents the loss of geometric structure: the direction becomes unstable, the velocity becomes erratic, the curvature spikes and oscillates. Instead of a path through space, the agent is producing a random walk.

Phase character. The path's visual appearance changes with the agent's behavioral phase.

During *exploration*, the path is broad and gently curving. The agent covers ground, sweeping through regions of the space. The velocity is moderate to high. The curvature is moderate.

During *commitment*, the path straightens and narrows. The agent has picked a direction and is following it. The velocity is steady. The curvature drops toward zero.

During *failure*, the path bends sharply. A sudden change of direction, visible as a kink or a hairpin turn. The curvature spikes. The velocity may spike too (the agent jumps to a new region) or may drop (the agent pauses to reassess).

During *branching*, the path forks. Multiple possible continuations radiate from the branch point. The visual emphasis is on the moment of divergence.

During *convergence*, the path narrows and decelerates. The steps get shorter. The direction stabilizes. The agent is homing in on a solution. The visual is a path that tapers to a point.

These are conventions, not laws. They describe typical patterns. Any given session may deviate. But the conventions provide a shared visual vocabulary for discussing trajectory shapes across chapters.

The Auth Test, Drawn

Return to the failing test from Chapter 1. The agent was given a task: fix `test_session_expiry`. Over twelve turns, it oriented, built a model, committed to a fix, hit a failure, branched, and converged. We described those phases in narrative form. Here is what the trajectory looks like when drawn.

Phase 1: Orientation (Turns 1-2). The path begins at the origin and moves outward in a broad sweep. The agent reads the failing test, reads the error output. Each frame is in a different part of the space (test code is structurally distant from error output in the embedding). The steps are large. Velocity is high. The agent is getting its bearings. The path covers ground quickly, like a surveyor walking the perimeter of a site.

Phase 2: Model Formation (Turns 3-5). The path changes character. It slows and curves into a tighter region. The agent reads `session.py`, the config fixture, and the test again more carefully. These files are structurally related (connected by imports and call chains), so their embeddings sit in a neighborhood. The path traces a loop through this neighborhood, visiting each file's region and returning. Velocity decreases. Curvature is moderate and steady. The agent is orbiting a cluster of related code, extracting information from multiple angles.

The direction vector during this phase points consistently toward the "session management" region of the space. The agent is not wandering. It is circling with purpose.

Phase 3: Commitment (Turn 6). The path straightens. A single long step from the reading region to the editing region. The agent has decided to modify `create_session`. In the embedding space, the transition from "reading and understanding" to "producing an edit" is a distinct directional shift. The velocity is moderate. The curvature drops sharply. The path narrows to a line. One direction. No branching.

Phase 4: Failure (Turn 7). The sharpest bend in the entire trajectory. The agent runs the test. It fails. The test output (a type error, a new failure mode) places the new frame in a region of the space distant from the agent's recent positions. The step is large. The curvature spikes. The path makes a hairpin turn.

In the visual, this looks like a path that was heading confidently in one direction and suddenly reversed. The kink is unmistakable. It marks the moment the agent's model of the problem changed.

Phase 5: Branching (Turns 8-10). The path forks. The agent considers two options: add type conversion in `create_session`, or fix the config fixture to return an integer. In the embedding space, these options point in different directions. One leads toward the session module. The other leads toward the configuration files.

The agent reads two other functions that consume config values (Turn 8-9). This is information gathering at the branch point, and in the trajectory it looks like a short exploratory loop near the fork. The agent is not committing to either branch yet. It is probing the space around both options.

At Turn 10, the agent chooses. The path commits to the "add conversion in `create_session`" branch. The other branch, the one leading toward config fixtures, appears as a dashed line that fades. The fork point is labeled "convention check."

Phase 6: Convergence (Turns 11-12). The path narrows and decelerates. Turn 11: the agent edits the function. Turn 12: it runs the test and the test passes. The last two frames are close together in the space, and close to the intended solution region. The step sizes shrink. The direction stabilizes. The path tapers to a final point.

The full trajectory, drawn on a page, looks like this: a broad initial sweep, a tightening orbit, a straight committed line, a sharp kink, a brief branching exploration, and a tapering convergence. The six phases of the narrative are visible as six distinct geometric signatures.

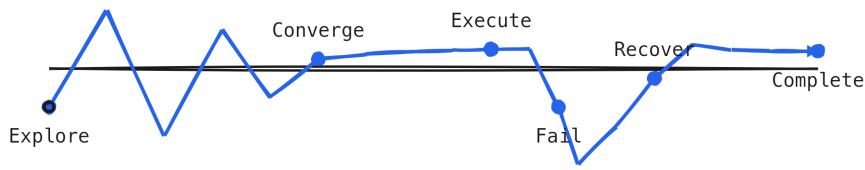

Figure 5.3: The canonical trajectory: broad sweep, tightening orbit, committed line, sharp kink, branching, convergence

Two things are visible in the drawing that are not visible in the log.

First, the failure at Turn 7 is geometrically dramatic. In the log, it is one line among twelve. In the trajectory, it is the sharpest feature of the entire

path. The curvature spike makes it visually prominent in proportion to its actual impact on the agent's behavior. The log treats all turns equally. The geometry does not.

Second, the relationship between the branching phase and the commitment phase is visible as a spatial relationship. The branch point at Phase 5 sits near the commitment path from Phase 3. The agent is revisiting a decision it already partially made. In the log, you have to read the events and infer this. In the trajectory, you can see it: the path loops back toward its own earlier segment.

What Trajectories Reveal

Given the geometric vocabulary of this chapter, consider what becomes computable.

Session efficiency. The ratio of the direct distance from start to end (the "as the crow flies" distance in embedding space) to the total path length (the sum of all step sizes). A ratio near 1 means the agent moved directly from start to finish. A ratio near 0 means the agent wandered extensively relative to how far it actually got. This is a single number that captures the shape of the entire session.

Pivot detection. Peaks in the curvature profile correspond to moments when the agent changed direction. These can be detected automatically and flagged. Each pivot is a candidate for investigation: why did the agent change course? Was it a planned re-evaluation, or a reaction to unexpected information? The curvature tells you *that* a pivot occurred and *when*. The frames surrounding the pivot tell you *why*.

Phase segmentation. The trajectory naturally segments into phases based on its geometric properties. Regions of high velocity and moderate curvature are exploration. Regions of low curvature and steady velocity are commitment. Curvature spikes are pivots. Velocity decreases near the end are convergence. These segments can be identified algorithmically, without any semantic analysis of what the agent was reading or writing.

Trajectory comparison. Two sessions working on the same task can be compared geometrically. Are they following similar paths? Do they diverge at the same point? Does one arrive at the solution with fewer pivots? The

comparison does not require aligning the sessions turn by turn (which is difficult when they have different numbers of turns). It operates on the path shapes directly.

Degradation detection. When the trajectory's geometric regularity breaks down (direction becomes random, velocity becomes erratic, curvature oscillates without pattern), the agent is likely degrading. This is detectable from the trajectory before the agent produces a wrong answer. The geometry degrades before the output does.

Each of these measurements falls out of the trajectory's geometry without requiring access to the model's internal state, without requiring semantic analysis of the agent's reasoning, without requiring a human to read the session trace and make judgments. The trajectory is a mathematical object. These are its measurable properties. The measurements are automated.

What Gets Thrown Away

Every agent session produces a trajectory. The twelve-turn test fix from Chapter 1. The hundred-turn refactoring marathon. The three-turn trivial bug fix. The fifty-turn dead end that produces nothing. Each of them traces a path through the embedding space.

Most systems throw this information away.

The standard pipeline captures the log, extracts summary statistics (turns, tokens, pass/fail), and discards everything else. The trajectory, the geometric object with all its measurable properties, is never constructed. The path is never drawn. The direction, velocity, and curvature are never computed.

This is like recording that a pilot flew from San Francisco to New York and arrived on time, without recording the flight path. You know the outcome. You do not know the shape of the journey. You cannot tell whether the pilot diverted around a storm, whether the approach was smooth or required three attempts, whether the route was efficient or circuitous. The outcome is identical for all of these scenarios. The trajectories are not.

The information exists. The logs contain the data needed to construct the trajectory. The embeddings can be computed. The geometry can be

measured. The pipeline simply does not do it. The data flows through and the shape is lost.

This book is about not losing the shape.

The chapters that follow will define the forces that act on trajectories (inertia, entropy, the structure of the space), develop the measurement theory for extracting signals from trajectory data, and build the systems that turn measurement into improvement. All of it depends on the object defined in this chapter.

The trajectory is the fundamental unit of analysis for understanding agent behavior. Not the turn. Not the session summary. Not the benchmark score. The path.

Chapter 6

The Coordinate Space

Five dimensions per frame. Many frames per session. The result is a space.

Chapter 4 defined the frame: a five-dimensional snapshot of an agent's state at a single moment. Semantic position, topological position, budget remaining, plan coherence, behavioral mode. Each frame is a point with five coordinates.

Chapter 5 defined the trajectory: the sequence of frames a session traces over time. Each trajectory is a path, a curve drawn through some space by the movement from one frame to the next.

But what is the space itself?

A trajectory implies a space the same way a hiking trail implies terrain. The trail is the thing you walk. The terrain is the thing you walk *through*. The terrain exists whether or not anyone walks on it. It has ridges and valleys and flat stretches and impassable cliffs. The trail follows the contours of the ground, shaped by them, constrained by them. Two hikers on the same ridge will take similar paths not because they coordinated, but because the terrain gives them limited options.

Agent sessions work the same way. Two agents solving similar problems trace similar trajectories not because they share a strategy, but because the space they move through has a shape. That shape constrains movement. It channels some paths and blocks others. Understanding that shape is the project of this chapter.

We are going to build the coordinate space: the mathematical object that contains all possible frames and all possible trajectories. Once we have it, we can talk about distance, neighborhoods, regions, and clusters. We can ask what it means for two agents to be "near" each other. We can project the space down for visualization and know what we lose in the projection.

The coordinate space is the map. Everything else in this book is about the paths agents draw on it, and the forces that push them along those paths.

How Dimensions Compose

Start with what a frame actually contains.

Semantic position: a vector capturing what the agent currently understands. This comes from embeddings, the machinery developed in Chapter 3. The agent has read certain files, encountered certain error messages, formed certain hypotheses. All of that gets embedded into a continuous vector. Two agents that understand similar things about their respective tasks have nearby semantic vectors, even if they are working on different codebases.

Topological position: where the agent sits in the structure of its environment. Which files has it visited? Which modules has it touched? How deep is it into the dependency graph? Two agents reading the same file are at the same topological position. Two agents that have traversed the same sequence of files have followed the same topological path.

Budget: how much resource remains. Tokens in the context window. API calls remaining. Wall-clock time before a timeout. Budget is monotonic. It only goes down. (Or, in the case of context windows with eviction strategies, it goes down and then jumps back up when old context is discarded, then goes down again.)

Plan coherence: how aligned the agent's current actions are with its stated or inferred plan. High coherence means the agent is doing what it set out to do. Low coherence means it has drifted.

Behavioral mode: the qualitative type of activity the agent is performing. Reading. Writing. Testing. Searching. Planning. Backtracking. This is categorical, not continuous, but it still functions as a dimension of variation between frames.

Five dimensions. That is the minimum. Some implementations add more: confidence estimates, error counts, progress metrics. But five is the foundation. And five is already more than any single metric can handle.

The first thing to notice is that these dimensions are not independent.

Concatenation would be the naive approach. Stack the five numbers (or vectors) side by side, call the result a frame vector, move on. Concatenation treats each dimension as a separate channel carrying separate information. But that misses the interactions.

Semantic position and topological position are often correlated. An agent reading authentication files is likely thinking about authentication. Its topological position (deep in the auth module) and its semantic position (embeddings near auth-related concepts) move together. Reading auth files while thinking about auth. Natural. Expected. The dimensions agree.

But sometimes they diverge. An agent might be reading a database migration file while semantically focused on a race condition in the session manager. The topological position says "database layer." The semantic position says "concurrency bug." That divergence carries information. It might indicate that the agent has discovered a cross-cutting concern. It might indicate that the agent has gotten lost. Either way, the divergence itself is a signal that neither dimension alone can produce.

Budget and behavioral mode interact differently. Budget marches forward monotonically, always decreasing. Behavioral mode cycles. The agent reads, then writes, then tests, then reads again. Budget does not cycle. It ticks down. The relationship between them is not correlation. It is a constraint. As budget decreases, certain behavioral modes become increasingly unlikely. An agent with five percent of its context remaining is not going to enter an exploratory reading phase. The budget squeezes the behavioral mode into a narrower range.

Plan coherence interacts with everything. When coherence is high, the other dimensions tend to change in predictable ways. Topological position follows a logical path through the codebase. Semantic position deepens along a consistent thread. Budget decreases at a steady rate. When coherence drops, the other dimensions become noisy. Topological position jumps. Semantic position scatters. Budget consumption spikes as the agent thrashes.

The interactions between dimensions carry information that the individual dimensions do not. Semantic-topological correlation tells you whether the agent is in the right place for what it is thinking about. Budget-mode interaction tells you whether the agent is adapting its strategy to its remaining

resources. Coherence modulates the predictability of every other dimension.

A frame, then, is not five independent numbers. It is five numbers whose relationships are part of the measurement. The coordinate space is not a simple Cartesian product. It is a structured space where certain regions are dense with real agent behavior and other regions are empty because no real agent would occupy them. (No agent has zero budget remaining and high plan coherence. The budget ran out. The plan is moot.)

Distance in This Space

Two frames. How far apart are they?

The question sounds simple. It is not. Distance in a multi-dimensional space depends on what you mean by "far apart," and different definitions produce different answers.

Start with a concrete example.

Agent A is reading `auth.py`, thinking about session management, with sixty percent of its budget remaining, high plan coherence, and in reading mode. Agent B is reading `auth.py`, thinking about database schemas, with sixty percent of its budget remaining, low plan coherence, and in reading mode.

Same file. Same budget. Same mode. Identical topological position. But Agent A understands why it is reading this file and Agent B does not. Are they close?

In the topological dimension: identical. Distance zero.

In the budget dimension: identical. Distance zero.

In the behavioral mode: identical. Distance zero.

In the semantic dimension: far apart. Auth-session-management embeddings and database-schema embeddings are in different neighborhoods.

In the plan coherence dimension: far apart. High versus low.

Two out of five dimensions say these agents are in the same place. Three out of five say they are not. What is the overall distance?

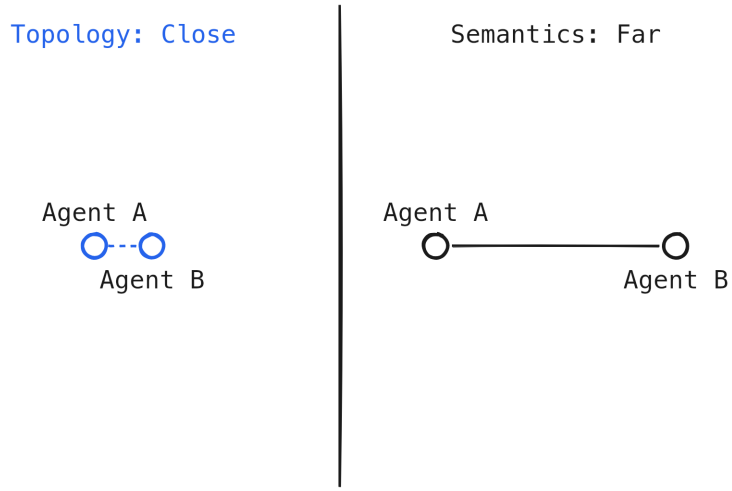

Figure 6.1: Two agents close in one dimension but far apart in another

This is where metric choice matters.

The simplest option is Euclidean distance. Square the difference in each dimension, sum the squares, take the square root. Euclidean distance treats all dimensions equally and combines them through the Pythagorean theorem extended to five dimensions. In our example, Euclidean distance would say: pretty far apart, because the semantic and coherence gaps are substantial, even though topology, budget, and mode are identical.

But Euclidean distance might be misleading here. Maybe topological position matters more than semantic position for certain analyses. Two agents reading the same file are “close” in a way that should dominate the distance calculation, regardless of what they are thinking about, if the analysis is about codebase coverage. For a different analysis focused on understanding, the semantic gap matters more.

Cosine similarity is another option. Treat each frame as a vector and measure the angle between them. Cosine similarity cares about direction, not magnitude. Two frames pointing in the same direction (similar relative proportions across dimensions) are close, even if one is “further out” from the

origin. This is useful when you care about the pattern of a frame rather than its absolute values. An agent with high budget and high coherence is "similar in shape" to an agent with low budget and low coherence, even though their absolute positions differ.

Then there are learned metrics. Train a model on pairs of frames that domain experts have labeled as "similar" or "dissimilar." Let the model learn a distance function that agrees with human judgment. The learned metric does not have to match any standard geometric formula. It can assign different weights to different dimensions. It can handle nonlinear interactions. The cost is opacity. The learned metric does what it does, but explaining *why* it considers two frames close requires interpreting the model.

Each metric is a design choice. It determines what "similar" means. Euclidean distance defines similarity as proximity in all dimensions simultaneously. Cosine similarity defines it as alignment in direction. A learned metric defines it as whatever the training data says.

The right choice depends on the question you are asking.

For diagnosis ("is this agent behaving similarly to agents that got stuck?"), a learned metric trained on labeled sessions tends to work best. It captures the complex interactions between dimensions that predict failure.

For clustering ("group these sessions by behavioral type"), cosine similarity often works well. It finds agents with similar behavioral profiles regardless of where they are in their budget cycle.

For anomaly detection ("is this frame unusual?"), Euclidean distance from the centroid of "normal" frames provides a clean signal. Points far from the centroid in any dimension trigger an alert.

The specific metric is not the point. The point is that distance *requires all five dimensions*. Any metric that ignores one dimension loses information. Two agents at the same topological position are only "close" if they are also close in the other four. Otherwise you are measuring distance on a map with missing axes, the same collapse that Chapter 1 identified in benchmark scores.

Neighbors and Clusters

A coordinate space with a distance metric lets you ask a natural question: who are my neighbors?

Given a frame, its neighbors are the other frames that are closest to it in the space. Given a trajectory, its neighbors are the other trajectories that follow the most similar paths.

This is where the coordinate space reveals things that raw logs cannot.

Consider two agents. Agent A is debugging a null pointer exception in a Java web application. Agent B is debugging a type error in a Python CLI tool. Different languages. Different error types. Different codebases. On paper, these sessions have nothing in common.

But watch their trajectories.

Agent A reads the stack trace, identifies the offending line, traces the null value backward through two function calls to a method with a missing conditional case, adds the case, and passes the test. Agent B follows the same pattern exactly — different language, different error type (None instead of null), but the same backward-tracing structure through the same number of calls to the same kind of fix.

Different logs. Different file names. Different languages. But the trajectories through the coordinate space are nearly identical. Both agents:

1. Started with high entropy, low coherence (reading the error, not yet oriented).
2. Moved to a focused topological path (tracing backward through call chains).
3. Maintained high semantic consistency (staying on the thread of the null/None value).
4. Converged to low entropy, high coherence (editing the specific conditional).
5. Validated with a single test run.

The coordinate space reveals that these two sessions are the same shape. They are neighbors, despite having zero overlap in the specifics. The bug structure was similar. The codebase structure was similar enough. The result was similar trajectories.

This is not a parlor trick. It is a practical capability with direct applications.

Transfer learning. If you know that trajectory-shape-X usually leads to success, and you see a new session entering that shape, you can predict success before it happens. If trajectory-shape-Y usually leads to failure (a specific pattern of widening entropy and falling coherence), you can intervene early.

Failure taxonomy. Instead of categorizing failures by their surface description ("agent edited the wrong file," "agent ran out of tokens," "agent produced a syntax error"), you can categorize them by their trajectory shape. A taxonomy based on trajectory shape groups together failures that share a root cause, even when the surface symptoms differ. "Premature commitment" is a trajectory shape: coherence drops after a brief high, the agent narrows too early, and the narrow path leads to a dead end. The surface symptom might be editing the wrong file. Or producing a partial fix. Or running out of budget on a doomed approach. Same trajectory shape. Same root cause. Different symptoms.

Capability assessment. If you test an agent on a hundred tasks and cluster the resulting trajectories, you do not get a single pass rate. You get a map of behavioral types. The agent might excel at backward-tracing trajectories (the null-pointer shape from the example) but fail at exploratory trajectories (tasks requiring broad codebase search before committing). A single pass rate of sixty-three percent hides this structure. The clustered trajectory map reveals it.

The clusters emerge from the data. Nobody defines them in advance. You collect trajectories, compute pairwise distances, and apply a clustering algorithm. The groups that form are the natural families of agent behavior. Some families will match categories you already had names for. Others will surprise you. The second kind is where the real value lives. Those are the behavioral patterns you could not see in the logs, because you did not know to look for them.

Dimensionality Reduction for Visualization

The coordinate space has at least five dimensions. Semantic position alone might contribute hundreds, depending on the embedding model.

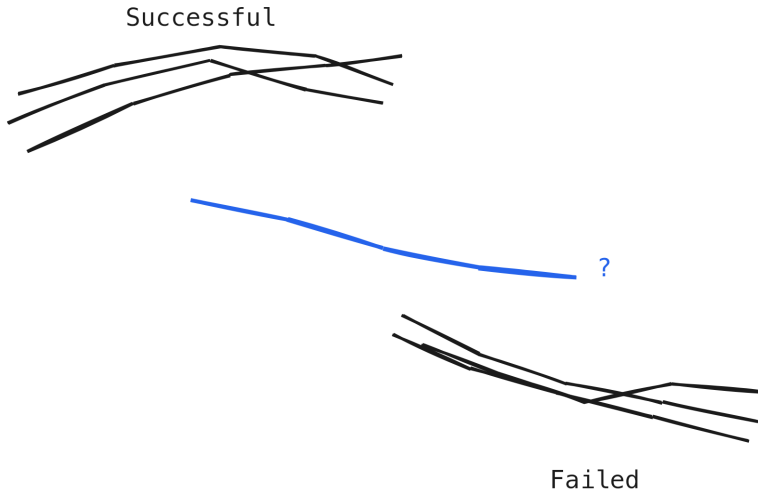

Figure 6.2: Trajectory clusters in the coordinate space, grouped by behavioral similarity

The full frame vector could easily live in a space with 300 or more dimensions.

Humans cannot see 300 dimensions. We get three on a good day.

Dimensionality reduction is the set of techniques that project a high-dimensional space down to two or three dimensions for visualization. The goal is to create a picture that preserves the important structure of the original space while throwing away the structure we cannot perceive anyway.

The simplest technique is PCA: principal component analysis. PCA finds the axes of greatest variation in the data and keeps the top two or three. If most of the differences between frames lie along two directions, PCA captures those directions and produces a clean 2D plot. The limitation: PCA only captures linear structure. If the interesting patterns are curved or folded, PCA unfolds them and loses the shape.

For agent trajectory data, two nonlinear techniques tend to work better: t-SNE and UMAP.

t-SNE (t-distributed Stochastic Neighbor Embedding) works on a specific principle. In the original high-dimensional space, every pair of points has some distance between them. t-SNE converts these distances into probabilities: nearby points get high probability, distant points get low probability. Then it constructs a 2D arrangement of points and adjusts it until the 2D probabilities match the high-dimensional probabilities as closely as possible. The result is a 2D plot where points that were neighbors in the original space are neighbors in the projection.

UMAP (Uniform Manifold Approximation and Projection) works on a similar principle with different mathematics. It builds a graph of nearest-neighbor relationships in the original space, then finds a low-dimensional layout that preserves that graph. Points connected in the high-dimensional graph stay connected in the projection.

Both t-SNE and UMAP do the same thing at a high level: they preserve *local* structure. If two frames are neighbors in the full coordinate space, they will tend to be neighbors in the 2D projection. The clusters you see in the 2D plot correspond to real clusters in the original space. Points that are near each other in the visualization really are near each other in the full five-dimensional (or 300-dimensional) representation.

What gets preserved:

- **Neighborhood relationships.** Points near each other in the original space stay near each other. This is the primary guarantee. If a cluster of trajectories appears in the 2D plot, those trajectories really are similar.
- **Cluster structure.** Groups of similar frames form visible clumps. The number of clumps and their relative sizes reflect the real clustering in the original data.
- **Relative density.** Dense regions (many similar frames packed together) appear dense in the projection. Sparse regions appear sparse.

What gets lost:

- **Absolute distances.** Two points that are far apart in the 2D plot might be much farther apart in the original space than two other points with the same apparent gap. The 2D distances are not proportional to the original distances. You can trust the neighborhoods. You cannot trust the gaps.

- **Global structure.** The relative positions of clusters to each other may not be meaningful. Cluster A appearing to the left of Cluster B in the 2D plot does not necessarily mean anything about the relationship between A and B in the original space. The algorithm places clusters wherever they fit, and the global layout is somewhat arbitrary.
- **Inter-cluster distances.** Two clusters that appear to be the same distance apart might be wildly different distances apart in the original space. Local structure is preserved. Global geometry is not.

This is a Flatland callback. We spent Chapter 1 arguing that projections lose information. Now we are deliberately projecting from 300 dimensions to 2. That sounds like exactly the mistake we warned against.

The difference is awareness. The Flatlander does not know the sphere exists. The Flatlander believes the circle *is* the object. We know what we are projecting from. We know what the original space looks like (we built it). We know what the projection preserves (neighborhoods) and what it discards (global distance, absolute scale). We are choosing to lose information in a controlled way, for a specific purpose: making the space visible to human eyes.

That is a legitimate operation when you understand its limits. The danger is forgetting those limits. A 2D UMAP plot of agent trajectories is a useful visualization tool. It is not the coordinate space. The coordinate space is the full-dimensional object. The plot is a shadow on the wall, cast with a well-chosen light source. When you see clusters in the plot, believe them. When you see distances between clusters, do not.

There is one more subtlety. Different runs of t-SNE and UMAP produce different layouts. The algorithms involve random initialization, and the final layout depends on where you start. Two runs on the same data will produce plots where the same clusters appear, but in different positions and orientations. The clusters are stable. Their placement is not. This is another reminder that the 2D coordinates are not “real” in the way the original five dimensions are real. The 2D coordinates are an artifact of the visualization. The neighborhood structure they reveal is not.

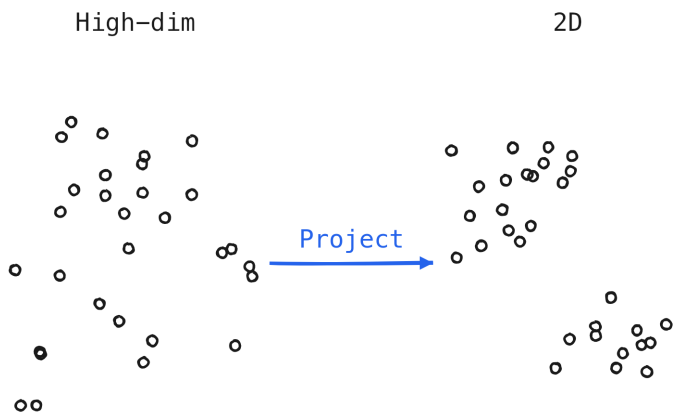

Figure 6.3: High-dimensional trajectory data projected to two dimensions via UMAP, preserving neighborhood structure

The Shape of the Space

With a coordinate space, a distance metric, and visualization tools, we can start to observe properties of the space itself.

Not all regions of the space are equally populated. Agent behavior concentrates in certain areas and avoids others. These concentrations and voids tell us something about what agents actually do, as opposed to what they theoretically could do.

Consider the region of the space where topological position is "deep in the test directory," semantic position is "reasoning about production architecture," and behavioral mode is "writing." This region exists in the coordinate space. A frame with these coordinates is geometrically valid. But very few real agent frames land there. Agents that are deep in the test directory are usually thinking about tests, not production architecture. The semantic and topological dimensions are correlated in practice, even though they are independent in theory.

The populated regions of the space form a manifold: a surface or volume within the larger space where real behavior lives. The manifold has lower dimensionality than the full space. Five dimensions of frame coordinates, but real behavior might occupy a three-dimensional surface within that five-dimensional volume. The two "missing" dimensions correspond to the correlations between dimensions. Semantic position is partially predicted by topological position. Behavioral mode is partially predicted by budget remaining. These correlations collapse the effective dimensionality.

This is why dimensionality reduction works at all. If agent behavior truly occupied all five dimensions uniformly, projection to 2D would lose three dimensions of genuine variation. Most of the information would be destroyed. But because real behavior lives on a lower-dimensional manifold, projection to 2D loses less than you might expect. The manifold might be close to 2D already, with the other dimensions carrying only redundant information.

The shape of the manifold varies by domain. Coding agents in well-structured codebases tend to occupy a manifold with clear structure: distinct regions for exploration, implementation, and testing, connected by predictable transition paths. Coding agents in messy codebases produce a more tangled manifold: regions overlap, transitions are irregular, the same topological position appears with wildly different semantic contexts.

The manifold shape is itself a measurement. It characterizes the domain, not the agent. Different agents in the same domain will populate similar manifolds, because the domain constrains which combinations of frame coordinates are reachable. The agent chooses where to go on the manifold. The domain determines the manifold's shape.

From Points to Paths

A single frame is a point in the coordinate space. A trajectory is a path. The distinction matters because paths have properties that points do not.

A point has position. That is all. Where is the frame?

A path has position, direction, speed, curvature, and length. Where is the agent? Which way is it going? How fast is it moving? Is the path bending? How far has it traveled?

Direction is computable from successive frames. If the last three frames show the agent moving from topological position A to B to C along a consistent semantic thread, the direction is clear. If the last three frames show the agent bouncing between A, C, and A again, there is no coherent direction. The agent is oscillating rather than progressing.

Speed is the distance between successive frames divided by the time (or turn count) between them. An agent that covers a lot of ground in the coordinate space per turn is moving fast. This can be good (making rapid progress through the problem) or bad (thrashing between distant parts of the space without dwelling long enough to accomplish anything). Speed alone does not distinguish. Speed combined with direction does. Fast and straight is progress. Fast and jagged is thrashing.

Curvature measures how much the path bends. A straight trajectory has zero curvature. The agent picked a direction and held it. A trajectory that bends gently has low curvature. The agent adjusted course gradually, maybe as it learned more about the problem. A trajectory that bends sharply has high curvature. Something happened. A test failed. An assumption was wrong. The agent pivoted.

Curvature spikes are interesting. They mark the moments where the agent's understanding changed enough to redirect its behavior. Every sharp bend in the trajectory corresponds to a revision of the agent's model of the problem. Mapping these bends tells you when the agent learned something.

Length is the total distance the trajectory covers. Not the straight-line distance from start to end (that is displacement, a different and also useful measure), but the total distance along the curve. Two agents that start and end at the same coordinates in the space might have very different path lengths. One took a direct route. The other wandered. Path length divided by displacement gives you an efficiency ratio. A ratio of 1.0 means the agent went straight from start to finish. A ratio of 3.0 means the agent traveled three times the minimum distance. The ratio quantifies how much of the agent's movement was productive.

These path properties, direction, speed, curvature, length, are not available from frame data alone. They emerge from the sequence. They are properties of the trajectory, not properties of any individual frame. This echoes the central theme from Chapter 1: the motion between frames is where meaning lives.

What the Map Makes Possible

We now have enough structure to state what the coordinate space provides.

It provides a common reference frame for comparing agents. Two agents, different architectures, different prompting strategies, different tool sets, working on different tasks. Without the coordinate space, comparing them requires reducing each to a scalar: pass rate, token count, average turn count. With the coordinate space, you can compare their trajectories as geometric objects. Same shape? Similar shapes? Completely different shapes? The comparison preserves structure that scalar metrics destroy.

It provides a language for describing failure modes. Instead of "the agent got stuck," you can say "the trajectory entered a low-curvature orbit in the region of the space corresponding to the auth module, with declining coherence and constant topological position." That description is specific enough to diagnose. More useful: it is specific enough to detect automatically. A monitoring system that watches for trajectories matching that pattern can flag them before the agent exhausts its budget.

It provides a substrate for learning. If you have a thousand trajectories, each labeled with an outcome, you can train models to predict outcome from trajectory shape. Which trajectory shapes lead to success? Which lead to failure? Which lead to partial success? The coordinate space gives the model a structured input representation. Without it, the model would have to learn the coordinate system and the prediction task simultaneously. With it, the representation is given; the model only has to learn the prediction.

It provides grounding for the forces introduced in Chapter 1 and developed through the rest of this book. Inertia is a tendency to maintain direction in the coordinate space. Entropy is the degradation of trajectory structure over time. Spatial difficulty is the curvature of the manifold in a particular

region. These forces are not metaphors. They are geometric properties of paths in a defined space. The coordinate space is what makes them measurable.

And it provides, finally, a way to see what we are building. Not the raw logs, which are too detailed to parse. Not the summary metrics, which are too compressed to inform. The coordinate space sits at the right level of abstraction: rich enough to preserve the dynamics of agent behavior, structured enough to submit to mathematical analysis, visual enough (after projection) to build intuition.

The Map Is Not the Territory

One warning before we proceed.

The coordinate space is a model. It captures the aspects of agent behavior that the five dimensions measure. It does not capture everything. An agent's behavior is also shaped by the specific content of files it reads, the exact wording of error messages, the particular structure of the codebase it navigates. These details influence the trajectory but are not represented in the five-dimensional coordinates. They are the terrain beneath the contour lines: real, relevant, but below the resolution of the map.

The map is useful precisely because it abstracts away those details. You cannot analyze a thousand trajectories at the level of individual file contents. You can analyze a thousand trajectories as paths through the coordinate space. The abstraction is the point.

But the abstraction has limits. Two trajectories that look identical in the coordinate space might differ in ways that matter. Two agents with the same semantic trajectory might be reading the same file types but drawing different conclusions. The coordinate space captures that they are in the same region. It does not capture what they understand about being there.

This is an inherent trade-off of any coordinate system. Latitude and longitude tell you where a city is. They do not tell you what the city is like. Two cities at similar coordinates might be a gleaming metropolis and a quiet fishing village. The coordinates locate them in the same neighborhood. The coordinates do not characterize them.

The coordinate space locates agent behavior. Characterizing it requires the additional tools developed in subsequent chapters: the force analysis that explains *why* trajectories take the shapes they do, and the measurement theory that extracts specific signals from trajectory data.

The space is the map. The trajectories are the paths on the map. What remains is understanding the forces that shape those paths.

Part III

Dynamics

Chapter 7

Inertia

An agent has been working for fourteen turns. It read the failing test. It read the error output. It traced the stack trace to a module called `payment_processor`. It read that module. It found a function called `validate_charge` that handles currency conversion. It read the currency conversion utility. It noticed a rounding error in the conversion logic. It formed a hypothesis: the test fails because `validate_charge` rounds before converting instead of after. It read two more files to confirm the hypothesis. It began editing `validate_charge`.

Then, on Turn 15, new information arrives. The test output, re-examined more carefully, shows that the rounding error exists but is not what the test is checking. The test is checking an entirely different code path. The agent's hypothesis is wrong. Not subtly wrong. Wrong about which code path the test exercises.

What does the agent do?

It could abandon the hypothesis, go back to the test, re-read the assertion, and start over. That would be the rational move. The hypothesis does not match the evidence. Discard it.

But discarding it means fourteen turns of accumulated context pulling in the wrong direction. Every file read, every connection traced, every inference drawn, all of it pointed toward the currency conversion logic. The agent's entire model of the problem lives in that part of the codebase. Starting over means treating all of that work as waste.

So the agent does something else. It looks for a way to make the currency conversion hypothesis work. Maybe the test *does* exercise that path indirectly. Maybe the assertion is checking a downstream effect of the rounding error. The agent reads more files, looking for a connection between the code path it has been investigating and the assertion that actually failed. It

spends four more turns trying to reconcile its hypothesis with the evidence before finally giving up and returning to the test.

Eighteen turns to reach a point it could have reached in fifteen. Three turns burned because the accumulated context was too heavy to set down.

That weight has a name. It is the first dynamic force of agent behavior.

The Weight of What You Know

Every action an agent takes adds to its context. A file read deposits information. A hypothesis formed deposits commitment. An edit made deposits investment. These deposits do not sit passively. They exert pull. They make the next action more likely to continue in the same direction than to veer off in a new one.

This is not a claim about psychology or consciousness. It is a claim about information mechanics. The agent's context window contains a running transcript of everything it has observed and done. That transcript is the input to every subsequent decision. When the transcript is dominated by observations about currency conversion, the model's next output is overwhelmingly likely to be about currency conversion. Not because the model "wants" to continue. Because the statistical weight of the context pushes the distribution of likely next tokens toward continuations that are consistent with the preceding context.

The context is not a neutral record. It is a gravitational field.

In Chapter 1, we gave this force a name. We called it *inertia* and described it informally: the deeper an agent invests in a line of investigation, the harder it becomes to change direction. Now we make that informal description precise.

Defining Inertia

Start with frames. A trajectory is a sequence of frames, where each frame captures the agent's observation, decision, action, and feedback at one turn. (Chapter 2 and Chapter 4 formalized this as the tuple (o, d, a, f).) Each frame has an embedding: a position in the coordinate space defined

in Chapter 6. The embedding encodes what the agent is doing, what it is looking at, and the accumulated context behind it.

Inertia is a property of a trajectory at a given point. It is computed from the frames leading up to that point. The intuition: how much accumulated context weight is pulling the trajectory forward along its current heading?

Formally, let the trajectory up to frame t consist of frames $f_1, f_2, \ldots, f$. Each frame has an embedding vector e_i. Define the *trajectory heading* at frame t as the recent direction of movement: the vector difference between the current position and a position some frames back, normalized to unit length.

Now define the *context mass* at frame t:

$$M(t) = \Sigma_{i=1}^{t} w(t - i) \cdot \|\Delta e_i\|$$

where $\Delta e_i = e_i - e_{i-1}$ is the step the agent took at frame i, $\|\Delta e_i\|$ is the magnitude of that step (how much the state changed), and $w(t - i)$ is a weight that decreases with age. Recent frames contribute more than old ones. (We will specify the decay function shortly.)

Context mass is a scalar. It accumulates with every frame. Large steps contribute more mass than small steps, because a large step means the agent incorporated a significant new observation or made a significant decision. The decay function ensures that old contributions fade, preventing mass from growing without bound.

Inertia, then, is context mass projected along the current heading:

$$I(t) = M(t) \cdot \cos(\theta)$$

where θ is the angle between the accumulated context mass vector and the current heading. When the agent has been moving consistently in one direction, θ is small, $\cos(\theta)$ is close to 1, and inertia is high. When the agent has been zigzagging, the mass contributions partially cancel, and inertia is lower even if the total mass is large.

In plain language: inertia is high when the agent has been doing a lot of work, recently, in a consistent direction. It is low when the agent has been doing little, or doing things a long time ago, or doing things in contradictory directions.

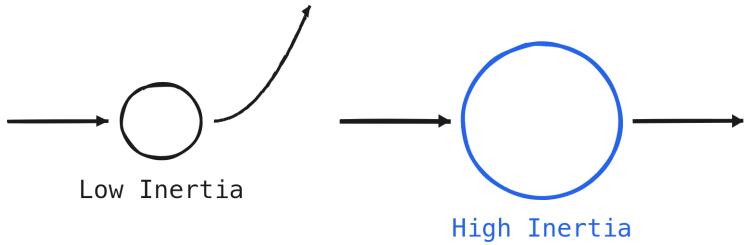

Figure 7.1: Small vs large context mass and their effect on directional resistance

What Inertia Feels Like

Numbers are precise but cold. Before going further into the formalism, consider what high and low inertia look like from the outside.

High inertia. The agent read five files in the same subsystem. It traced a call chain from the API endpoint through the service layer to the database query. It identified a specific function as the source of the bug. It wrote a fix. It is now running the test to confirm. The trajectory is straight, committed, purposeful. Every frame points the same direction. If this test fails, the agent will be reluctant to abandon its theory. It has too much invested.

High inertia looks like confidence. It looks like an agent that knows what it is doing. Sometimes it *is* an agent that knows what it is doing. The model is correct, the fix is right, the test will pass. Other times, it is an agent that has locked onto a wrong answer with the full weight of fourteen turns behind it. From the outside, the two are hard to distinguish. That is what makes inertia dangerous.

Low inertia. The agent read a file, then searched for something unrelated, then read a different file in a different subsystem, then went back to the first file. No consistent direction. Each frame points somewhere new. The agent has not committed to anything. It is light on its feet, changing direction easily, because there is no accumulated weight holding it on any particular course.

Low inertia looks like confusion. It looks like an agent that does not know what it is doing. Sometimes it *is* that. The agent is lost, thrashing, trying things at random. Other times, it is early-stage exploration. The agent is deliberately surveying the terrain before committing. Low inertia in Turn 3 is normal. Low inertia in Turn 30 is a warning sign.

Medium inertia, then a spike. The agent spent ten turns exploring broadly. Moderate inertia. Then it found something. A clue. A pattern. A connection. Over the next three turns, inertia climbs sharply as the agent commits to a theory and begins executing. The spike marks the transition from exploration to exploitation. It is the moment the agent decided what the problem is.

These are not abstract patterns. They are visible in trajectory data. Plot inertia over turns, and you see the commitment profile of the session. Flat and low: wandering. Rising steadily: convergence. Flat and high: committed execution. Sudden drop: the agent changed its mind.

Why Trajectories Branch

Inertia resists direction change. But direction changes happen. Trajectories branch. The agent was going one way, and now it is going another.

When inertia is high, something must overcome it. A direction change against strong inertia is an event. It happens for identifiable reasons. There are four.

Stall

The agent has been working on the same problem for a while. It has context mass. It has a heading. But it is not making progress.

The symptoms are specific: repeated reads of the same files, edits that get reverted, tests that keep failing in the same way. The agent is not

stuck because it lacks information. It is stuck because the information it has leads to a dead end. The hypothesis is wrong, or the approach is insufficient, but the inertia of accumulated context keeps the agent pushing forward on a path that goes nowhere.

Example. A coding agent spends twelve turns in the memory management module, convinced the problem is an unclosed connection pool. It adds cleanup logic, re-reads the module, adds more cleanup — each attempt fails identically. The trajectory has high inertia (twelve turns in one direction) and zero velocity (no forward progress). This is a stall: a train running at full power against a wall. Eventually the agent re-reads the test output instead of the module, notices the leak is in a completely different component, and branches.

Stalls produce reluctant branches. The agent does not choose to redirect. It exhausts its current direction and is forced to redirect. The branch point is marked by a cluster of repeated actions (the futile pushing) followed by a sudden jump to a different region of the embedding space.

Decomposition

The task splits. Not because the agent failed, but because the problem has parts.

"Fix the authentication bug" starts as one task. The agent investigates and discovers that the bug has two independent causes: the session handler is not refreshing tokens correctly, and the config parser is not reading the timeout value. These are separate problems in separate modules. The agent cannot fix both simultaneously. The trajectory forks.

Example. The agent reads the auth middleware. It finds two suspicious code paths. One involves token refresh logic in `session.py`. The other involves config loading in `settings.py`. The agent follows the token refresh path first, fixes it, then returns to the config path and fixes that.

The branch here is structural. The agent did not fail or get confused. The problem decomposed into subproblems, and the trajectory forked to address each one. The inertia before the branch was moderate (the agent was still investigating, not yet committed to a single path). The branch produces two sub-trajectories, each of which will develop its own inertia as the agent commits to solving each subproblem.

Decomposition branches are planned. The agent often signals them explicitly in its reasoning: "There are two issues here. Let me address the token refresh first." The branch point is marked by a deliberate pivot in the embedding space, not a desperate jump.

Speculation

The agent sees two possible approaches and deliberately explores both before committing.

This is the most sophisticated branching cause. The agent is not stuck. The agent is not decomposing a problem. The agent is managing uncertainty. It recognizes that it does not have enough information to choose between two approaches, and it invests a small amount of context in each one before committing to either.

Example. A coding agent needs to fix a performance regression. It sees two possible strategies: optimize the database query (add an index, rewrite the join) or add a caching layer in the application code. Both could work. The agent does not know which is more appropriate without investigating.

It reads the query execution plan. It reads the application layer where caching would go. It estimates the complexity of each approach. Then it commits to one.

The speculative branch is brief and intentional. The agent invests minimal context mass in each direction before choosing. The inertia at the branch point is low by design: the agent has not committed. After the choice is made, inertia builds rapidly as the agent commits to the selected path.

Speculative branches are visible in trajectory data as small, symmetric excursions. The trajectory moves a short distance in direction A, returns near its starting point, moves a short distance in direction B, returns, then moves decisively in one direction. The symmetry and brevity distinguish speculation from stalling (where the excursions are repeated and desperate) and from decomposition (where both branches are fully developed).

Escalation

The agent discovers that the problem is bigger than it thought. What started as a simple bug turns out to be a design flaw. What looked like a localized issue turns out to be systemic.

Escalation is not a failure of the agent's reasoning. It is a failure of the task description to capture the problem's scope. The agent did exactly what it was asked to do: investigate the bug. The investigation revealed that "the bug" is actually a symptom of something larger.

Example. An agent is asked to fix a test that checks user permissions. It reads the test. It reads the permission check function. It discovers that the function is correct, but it is using a permissions model that does not match the rest of the system. Three other modules use a different permissions model. The test is failing because the function being tested was written for the old model, but the test fixtures were updated for the new model.

The agent cannot fix this by editing one function. The discrepancy is architectural. The trajectory must change direction not because the agent's approach was wrong, but because the problem changed. What was "fix a failing test" is now "reconcile two permissions models." The scope exploded.

Escalation branches are marked by a distinctive pattern in the embedding space. The agent is moving through a localized region (the permission check function and its immediate context) when it suddenly jumps to a distant region (the system-wide permissions architecture). The jump is large. The inertia before the jump was high (the agent was committed to its local investigation). The jump overcomes that inertia because the new information is not just surprising but scope-altering. The agent did not find a new clue in its current investigation. It found a different investigation entirely.

Escalation is the hardest branch to recover from, because it invalidates not just the agent's current approach but its model of the problem. After a stall, the agent needs a new approach. After escalation, the agent needs a new understanding.

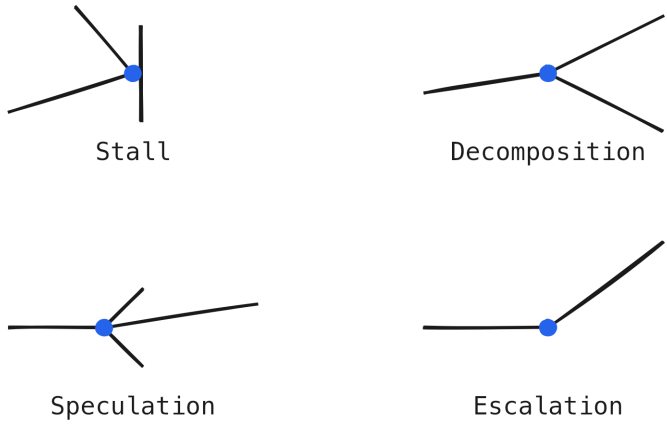

Figure 7.2: Four branch types: stall, decomposition, speculation, and escalation

Temporal Decay

Inertia is not permanent. Context fades.

This is not a design choice or an optimization. It is a physical property of the system. The agent's effective context has a natural decay gradient, and the gradient exists for two reasons.

First, attention. Transformer-based language models process their context window through attention mechanisms that do not weight all positions equally. Recent tokens receive stronger attention than distant tokens. (The exact pattern varies by model architecture, by layer, and by the specific content. But the general tendency toward recency bias is well-documented.) A file the agent read thirty turns ago contributes less to the model's next decision than a file read three turns ago, even though both are present in the context window. The older file is technically visible. It is not equally influential.

Second, relevance. Early context is often superseded by later context. The agent's initial hypothesis, formed at Turn 3, may have been revised at

Turn 8 and replaced entirely at Turn 14. The Turn 3 hypothesis still occupies context tokens, but it is no longer operative. Its effective contribution to inertia is near zero.

Both effects suggest exponential decay as the functional form. The contribution of frame *i* to the inertia at frame *t* decays as:

> w(t - i) = e^(-λ(t - i))

where λ is the decay rate. A larger λ means faster decay: old frames lose influence quickly. A smaller λ means slower decay: the agent carries its history longer.

The right value of λ is not universal. It depends on the agent architecture, the model's context handling, and the nature of the task. For practical purposes, a half-life of five to ten turns is a reasonable starting point. At a half-life of seven turns, a frame from fourteen turns ago contributes about one-quarter as much as a frame from the current turn. A frame from twenty-eight turns ago contributes about one-sixteenth.

The decay function has a specific and useful consequence for inertia: it makes inertia self-limiting. Without decay, inertia would grow monotonically as the session progresses. Every frame would add mass, and the agent would become increasingly immovable. With decay, old mass fades. The agent's effective inertia reflects the recent history, not the entire history. An agent that changed strategy ten turns ago has largely shed the inertia of its old strategy. Its current inertia reflects its new direction.

This matches observed behavior. Agents do change direction. They abandon hypotheses, restart investigations, pivot strategies. If inertia were permanent, direction change would become progressively impossible as sessions lengthened. It does not. Agents in their fiftieth turn are not dramatically more rigid than agents in their twentieth. Decay explains why. The old context is still in the window, but its effective weight has diminished.

(There is an important subtlety here. Decay is not the same as forgetting. The old context is still present. If the agent re-attends to it, if something in the current context triggers a connection to the earlier material, the old context can suddenly become relevant again. Decay describes the *default* diminishment of influence, not an absolute erasure. A long-dormant

hypothesis can be revived if new evidence points back to it. But in the absence of such a trigger, the decay is real.)

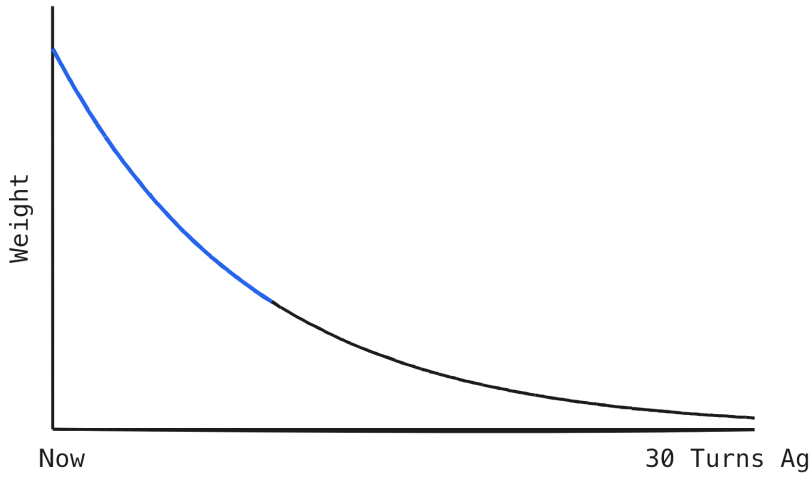

Figure 7.3: Temporal decay curve showing exponential falloff of frame contribution over time

Computing Inertia from Frames

The definition is in place. The decay function is specified. Now, the practical question: how do you compute inertia from actual trajectory data?

The inputs are a sequence of frame embeddings: $e_1, e_2, \ldots, e$□ . Each embedding is a vector in the coordinate space defined in Chapter 6. The computation proceeds in three steps.

Step 1: Compute step vectors. For each consecutive pair of frames, compute the displacement:

> $\Delta e_i = e_i - e_{i-1}$

This is the direction and magnitude of the agent's movement at frame *i*. A large $\|\Delta e_i\|$ means the agent's state changed significantly (it jumped to a

new part of the codebase, or received a surprising result). A small $\|\Delta e_i\|$ means the agent's state barely changed (it re-read the same file, or made a minor edit).

Step 2: Compute the weighted mass vector. Sum the step vectors with temporal decay:

$$\vec{M}(t) = \Sigma_{i=2}^{t} e^{-\lambda(t - i)} \cdot \Delta e_i$$

This is a vector, not a scalar. It points in the direction of the agent's recent net movement. If the agent has been moving consistently northeast (in the embedding space), the mass vector points northeast. If the agent has been zigzagging, the mass vector is short because the contributions partially cancel.

Step 3: Compute scalar inertia. The inertia is the magnitude of the mass vector:

$$I(t) = \|\vec{M}(t)\|$$

High inertia means the mass vector is long: the agent has accumulated significant, directionally consistent, recent movement. Low inertia means the mass vector is short: either the agent has not moved much, or its movements cancel out.

That is the complete computation. Frame embeddings in, scalar inertia out. Computable at every turn. No access to the model's internal state required. Just the trajectory data.

Inertia Profiles

Plot inertia over the course of a session, and you get an *inertia profile*: a curve that shows how the agent's commitment evolved over time. Different session types produce recognizable profiles.

The clean solve. Inertia starts low (the agent is orienting). It rises steadily through the middle turns as the agent investigates and commits to a hypothesis. It plateaus at a high level during execution (editing, testing). It drops slightly at the end as the agent wraps up. The profile is a smooth hill. This is the trajectory of an agent that found the right path and followed it.

The false start. Inertia rises through the early turns as the agent commits to an initial hypothesis. Then it drops sharply when the hypothesis fails. A brief valley of low inertia follows as the agent re-orients. Then inertia rises again as the agent commits to a new hypothesis. The profile has two hills with a valley between them. The valley is the branch point.

The spiral. Inertia rises, plateaus, drops. Rises again, plateaus, drops again. Multiple hills, each lower than the last. The agent keeps committing and failing, committing and failing. Each commitment is weaker than the previous one (lower peak inertia) because the agent has less confidence in each successive hypothesis. The decaying peaks are a signal that the agent is running out of ideas.

The wanderer. Inertia stays low throughout the session. No sustained rise. No commitment. The agent moves from thing to thing without ever building enough context mass in any direction to register as inertia. It is exploring, but not productively. Or it is stuck from the beginning, never forming a hypothesis worth pursuing.

The lock-in. Inertia rises early and stays high for the entire session. No dips. No valleys. The agent committed to its first hypothesis and never wavered. If the session succeeds, this profile indicates efficiency. If the session fails, it indicates stubbornness. The profile alone cannot tell you which. The outcome can.

These profiles are not theoretical categories invented for pedagogical convenience. They emerge from actual trajectory data. Compute inertia for a corpus of agent sessions, and you will see these shapes. The labels are ours. The shapes are real.

Inertia and the Four Branches, Revisited

The branching causes from earlier in this chapter each produce a characteristic signature in the inertia profile.

A **stall** branch shows up as a plateau followed by a cliff. Inertia stays high (the agent is committed) but velocity drops to near zero (no progress). The plateau persists for several turns as the agent pushes against the wall. Then inertia drops sharply as the agent finally redirects. The longer the plateau before the cliff, the more turns the agent wasted.

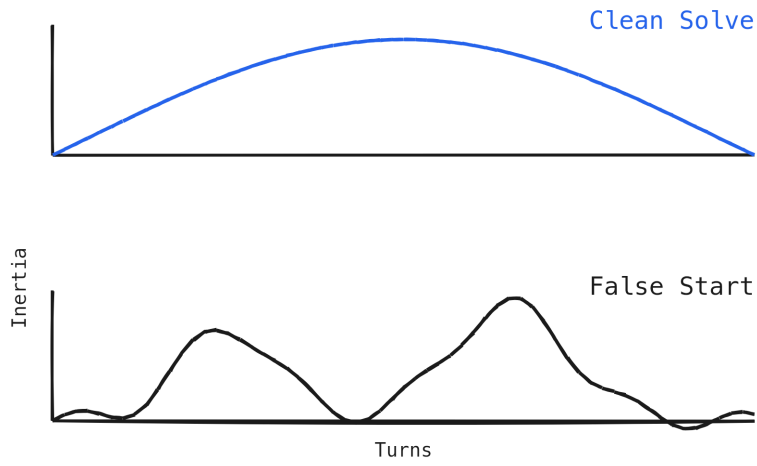

Figure 7.4: Inertia profiles: clean solve vs false start

A **decomposition** branch shows a moderate inertia level that splits into two rising curves. The parent trajectory had moderate inertia (still investigating). At the branch point, two sub-trajectories emerge, each developing independent inertia. In the profile, this looks like a fork: one curve becomes two.

A **speculation** branch produces small oscillations in inertia. The agent briefly invests in direction A (inertia tick up), pulls back (tick down), briefly invests in direction B (tick up), pulls back (tick down), then commits (sustained rise). The oscillations are small because the speculative investments are deliberately limited.

An **escalation** branch is the most dramatic. Inertia is high (the agent was committed to its local investigation). Then a sudden, deep drop. The agent's entire model of the problem just changed. The inertia does not merely decrease; it collapses, because the new direction is so different from the old one that the accumulated mass vector is nearly orthogonal to the new heading. The projection of old mass onto the new direction is close to zero. The agent is starting over, inertia-wise, even though the context window still contains all the old observations.

These signatures are diagnostic. Given an inertia profile, you can identify the branch type without reading the session transcript. The shape of the inertia curve tells you what happened.

Inertia Is Not Momentum

A brief terminological note. In physics, momentum is mass times velocity. An object with more momentum is harder to stop. Inertia, in the physics sense, is the property of mass that resists changes in velocity. The two concepts are related but distinct.

We chose the word *inertia* rather than *momentum* for the agent analog, and the choice is deliberate.

Momentum implies ongoing forward motion. An object with momentum is going somewhere. Inertia implies resistance to change. An object with inertia might be moving, or it might be stationary. A boulder at rest has inertia: it resists being pushed. A boulder rolling downhill has both inertia and momentum.

Agent trajectories exhibit both behaviors. An agent deep into a committed strategy has the equivalent of momentum: it is moving forward along a particular path. An agent that has accumulated significant context but stalled has the equivalent of inertia without momentum: it resists changing direction even though it is not making progress. The stall case is the more interesting one, and the more dangerous one, so the term that captures resistance-to-change is the more useful label.

When we mean forward motion, we will say *velocity*. When we mean resistance to direction change, we will say *inertia*. The distinction matters. High velocity with high inertia is productive commitment. Low velocity with high inertia is a stall. The combination of the two quantities tells you something that neither tells you alone.

What Inertia Predicts

A measurement is useful when it predicts something you could not otherwise see. Inertia predicts several things.

It predicts recovery time after failure. When an agent's current approach fails (a test does not pass, an edit produces errors), the time it takes to recover and try something new correlates with the inertia at the moment of failure. High inertia at the failure point means the agent will spend more turns trying to salvage the current approach before redirecting. Low inertia means the agent will pivot quickly. This is not a tautology. Inertia is computed from the trajectory data. Recovery time is measured from the outcomes. The correlation between them is an empirical finding, not a definitional one.

It predicts branch quality. When a trajectory does branch, the inertia at the branch point predicts whether the branch is productive. Branches from moderate inertia (the agent investigated enough to form a basis for redirecting, but not so much that it is entrenched) tend to lead to better outcomes than branches from very high inertia (the agent was forced to redirect after deep commitment) or very low inertia (the agent redirected before it had enough information to choose well).

It predicts session length. Sessions with early, high inertia tend to be short. The agent committed fast and either succeeded or failed fast. Sessions with late-developing inertia tend to be long. The agent explored extensively before committing, which consumed turns. Sessions with repeatedly collapsing inertia (the spiral profile) tend to be the longest and least successful.

It predicts the value of intervention. If you are monitoring an agent in real time and considering whether to intervene (provide a hint, redirect the agent, stop the session), inertia tells you how receptive the agent is likely to be. An agent with low inertia is easy to redirect. A nudge in the right direction will be picked up and followed. An agent with very high inertia will resist redirection. It will incorporate your hint but interpret it through the lens of its existing hypothesis. High-inertia agents require stronger interventions: not hints but explicit corrections. (Chapter 13 develops this into a formal intervention theory.)

These predictions are not theoretical. They are testable. Given trajectory data from a corpus of agent sessions, you can compute inertia at every point, record the outcomes, and check the correlations. The framework is falsifiable. If inertia does not predict these things, the framework is wrong. (It does.)

The Commitment Problem

Inertia is not a flaw to be eliminated. It is a trade-off to be managed.

An agent with zero inertia would never commit to anything. It would reconsider its approach at every turn, weigh all possible strategies from scratch, and make each decision independently of the ones that came before. This sounds rational. It is catastrophically expensive. The entire point of building up context is to avoid re-deriving conclusions from first principles at every step. Context mass is not dead weight. It is accumulated understanding. An agent that discards it at every turn is not flexible. It is amnesic.

An agent with infinite inertia would never change its mind. It would commit to its first hypothesis and execute it regardless of evidence. This is also catastrophic, but in the other direction. The agent would be impervious to feedback, correction, or new information. It would be a train on rails, unable to respond to a world that does not match its expectations.

Good agent behavior lives between these extremes. Build context. Accumulate mass. Commit to a direction. Follow it. But remain responsive to evidence that the direction is wrong. The agent should have enough inertia to be purposeful and not so much that it is stubborn. Enough mass to stay on course and not so much that it cannot course-correct.

This is the commitment problem. Every agent faces it. And every agent resolves it (well or badly) through the interplay between inertia and the forces that oppose inertia. We have described inertia as the force that resists change. In the next chapter, we describe entropy: the force that produces change. The tension between them governs the shape of every trajectory.

The Train

Return to the metaphor from the opening. An agent with accumulated context is like a train that takes a long time to stop. The metaphor is useful but incomplete.

A train has one degree of freedom: it moves along its track. It can go faster or slower, forward or backward. But it cannot turn. A train's inertia is one-dimensional.

An agent's inertia is not. The embedding space is high-dimensional. The agent can move in any direction. Inertia does not confine the agent to a track. It biases the agent toward continuing in its current direction. The difference is critical. A train that needs to change direction must stop, reverse, and find a new track. An agent that needs to change direction must overcome the bias of its accumulated context, but it can do so gradually: curving rather than reversing, drifting toward a new heading rather than making a hard stop.

The train metaphor captures the resistance. The high-dimensional reality captures the flexibility. Both are real. The agent does resist direction change. But the resistance is a bias, not a constraint. With sufficient new evidence, or sufficient decay of old context, the agent can turn. It just takes force proportional to the inertia being overcome.

How much force? That depends on the agent, the task, and the space. But the question is now quantitative, not qualitative. Inertia is a number. The force required to overcome it is a number. The framework turns "the agent seems stuck" into "the agent's inertia is 4.7 and the new evidence has a directional force of 2.1, which is insufficient to overcome it." The diagnosis suggests the treatment: provide additional evidence (increase the opposing force) or wait for decay (let the old inertia diminish) or intervene directly (inject a strong directional signal).

Numbers. Not vibes.

Forward

Inertia is the first dynamic quantity of the framework. It describes why agents commit: because accumulated context creates mass, and mass resists direction change. It describes when agents branch: when stalls, decomposition, speculation, or escalation generate enough force to overcome the accumulated mass. It describes how agents recover: slowly when inertia is high, quickly when it is low.

But inertia describes only half the story. It explains why agents stay on course. It does not explain why courses degrade.

An agent starts a session with a clear plan and a focused strategy. Twenty turns later, the plan is in tatters and the strategy has drifted into reactive patchwork. Inertia alone cannot explain this. Inertia resists change. Something is causing change despite that resistance.

That something is the subject of the next chapter.

Chapter 8

Entropy

Thirty turns into a coding session, the agent announces its plan: "I will refactor the `validate_session` function to separate token verification from session lookup." Clear intent. Reasonable decomposition. The kind of statement that, if you were reading the transcript, would make you nod and move on.

Then the agent opens `test_auth.py` and starts editing a test fixture.

Not the function it said it would refactor. Not the module that contains `validate_session`. A test file in a different directory, testing a different code path. The edit is syntactically valid. It even improves the test. But it has nothing to do with the stated plan.

Two turns later, the agent reads `database.py`. Then it edits a logging configuration. Then it returns to `test_auth.py` and modifies a different fixture. Then it reads `validate_session` for the first time since stating its intent, makes a small change, and announces that the refactoring is complete.

What happened?

The plan and the actions drifted apart. The session started with a clear goal and ended with a collection of loosely related edits that do not add up to the stated refactoring. The agent did things. Some of those things were useful in isolation. But the thread connecting them frayed somewhere around Turn 20 and never recovered.

The session lost coherence. Not in a dramatic way. There was no error. No crash. No moment where the agent announced "I am confused." The loss was gradual. Each individual turn was plausible in isolation. The agent had reasons for each action, or at least produced text that looked like reasons. But the aggregate trajectory stopped moving toward the goal. The actions became locally justified and globally incoherent.

This is entropy.

A Brief Information Theory Primer

Before we can measure coherence loss precisely, we need a small toolkit from information theory. The ideas are old — Claude Shannon published them in 1948 to solve problems about telephone wires — but they apply directly to agent behavior.

Shannon's central insight: *information is surprise*. A message that tells you something you already know carries no information. A message that tells you something unexpected carries a lot. Consider the weather report. You live in the Sahara Desert and the forecast says "sunny." You learned almost nothing — you already expected sun. Now you live in London and the forecast says "sunny." You learned more, because rain was plausible. The same format, the same delivery mechanism, but wildly different amounts of information. The difference is not in the message. It is in how expected the message was before you received it.

Entropy as uncertainty. Shannon formalized uncertainty with a single number. Start with a fair coin: heads or tails, each with probability 0.5. Before the flip, you have no idea what will happen. You are maximally uncertain. A loaded coin (heads 99% of the time) leaves you barely uncertain. A two-headed coin leaves you with zero uncertainty — no flip will ever surprise you. Entropy measures this overall uncertainty. For a set of outcomes with probabilities $p(x)$, Shannon entropy is:

> $$H = -\sum p(x) \log_2 p(x)$$

The fair coin gives $H = 1$ bit. The loaded coin gives $H \approx 0.08$ bits. The two-headed coin gives $H = 0$. The formula does exactly what the intuition predicts — it just puts a number on it. For any fixed number of outcomes, entropy is maximized when all outcomes are equally likely. The more concentrated the probability, the lower the entropy.

KL divergence measures how different two probability distributions are. For agent behavior, the key question is: how much has the agent's action distribution changed between Turn 5 and Turn 25? KL divergence answers this by asking how surprised you would be, expecting distribution

P, when you actually encounter distribution Q. A value of zero means the distributions are identical. Higher values mean greater divergence. The measure is asymmetric — the convention for agent monitoring is to set P as the earlier (baseline) distribution and Q as the later one, asking how much behavior has drifted.

Mutual information asks whether knowing one variable tells you about another. For agents: does knowing what the agent just read predict what it will write next? High mutual information between reads and subsequent edits means the agent's information gathering is purposeful — each observation feeds into decisions. Low mutual information means the reads and writes are decoupled: the agent gathers information and produces output, but the two streams have nothing to do with each other. This is the measurement that distinguishes purposeful file re-reading from aimless circling.

These three instruments — entropy, KL divergence, mutual information — are what this chapter applies to the specific problem of detecting when a session is falling apart.

Coherence Loss, Not Disorder

The primer above gave us instruments to assess whether an agent's actions are structured or random at a given moment. Now we apply them to a specific problem: detecting when a session is falling apart.

The word "entropy" carries baggage from physics. In thermodynamics, entropy is associated with disorder, with heat death, with the inexorable tendency of systems to become more chaotic. That association is misleading here. Agent entropy is not about disorder in the physical sense. It is about coherence loss. The degree to which an agent's actions stop being *about* a unified goal and start being *about* whatever happens to be in front of the model at the moment.

A disordered room has high physical entropy. A disordered agent session has high behavioral entropy. But the analogy only goes so far. A room's entropy is a property of its current state. An agent's entropy is a property of its behavior over time. You cannot look at a single turn and determine whether the session is entropic. You need a window. You need trajectory.

Shannon entropy, as defined above, measures the spread of a probability distribution. Applied to agent behavior, it tells you how concentrated or diffuse the agent's action patterns are over a window of turns. But the primer treated entropy as a diagnostic snapshot. This chapter treats it as a trajectory property: not what the entropy is at Turn 15, but how the entropy changes from Turn 1 to Turn 40. The shape of that change is the signal.

The entropy of a trajectory is computed over sliding windows of frames. Take a window of, say, eight turns. Within that window, estimate the distribution over file targets, action types, and module boundaries. Compute the Shannon entropy of each distribution. Slide the window forward by one turn and compute again. The resulting time series is the trajectory's entropy signature.

A healthy session has a characteristic entropy signature. It starts high during exploration, drops as the agent commits to a plan, stays low during focused execution, and may rise briefly if the agent encounters a surprise that forces re-exploration. An unhealthy session has a different signature: entropy that rises without falling, entropy that oscillates without converging, or entropy that stays flat at an uninformative level.

Think of it like a conversation with someone who gradually stops making sense. The individual sentences remain grammatical. The vocabulary stays normal. But the topic drifts, the references stop connecting, and after twenty minutes you realize the conversation is no longer *about* anything. You cannot point to the exact sentence where coherence broke down. You can only see it in the trajectory of the whole exchange.

Agent sessions degrade the same way. The individual turns remain well-formed. The tool calls have valid arguments. The model's reasoning text sounds confident. But the trajectory of actions stops converging on the task. The entropy signature captures this degradation even when individual turns look fine.

The four patterns that follow are the most common entropy signatures of sessions going wrong.

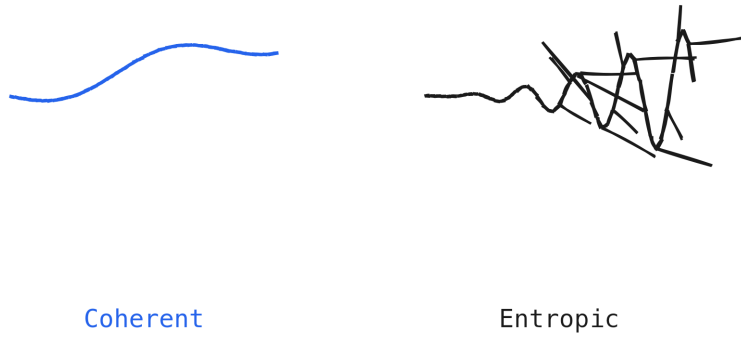

Figure 8.1: Coherent vs entropic trajectory in embedding space

The Four Entropy Signatures

Entropy does not manifest as a single shape. It has modes. Each mode corresponds to a different failure of coherence, and each is detectable through a different statistical fingerprint. These are the four signatures that appear most frequently in real agent sessions.

Circular Reads

The agent is tasked with fixing a race condition in a connection pool. It alternates between `pool.py` and `config.py` throughout the window, never expanding the search.

Over a ten-turn window, the file access set has two elements. It is not expanding. No new files enter the read set. The agent is not gathering additional information from new sources. It is re-reading the same material.

Compute the entropy of the file-target distribution within this window. It is low and stable. Two files, roughly alternating, gives an entropy near 1 bit.

That number does not change as the window slides forward. The entropy signature is a flat line.

Now compare this to what the signature *should* look like. A healthy investigation of a race condition would involve reading the pool implementation, then the callers of the pool, then the threading primitives, then perhaps the test suite that exercises concurrent access. The file-target set should expand. New files should appear in the read set as the agent traces the problem through the codebase. The entropy of the file-target distribution should rise as more targets enter the distribution, then fall as the agent narrows its focus.

The flat-line signature tells you the exploration phase never happened. The agent locked onto two files and stopped looking. The read set became periodic.

This is distinct from purposeful re-reading. There, an agent re-reads files to extract different information on each pass, and the mutual information between reads and subsequent edits remains high. Circular reads have a different fingerprint: low mutual information between the reads and any productive output. The agent reads `pool.py` and then does not edit `pool.py` or anything informed by it. It just reads `config.py` and then reads `pool.py` again. The loop is self-contained and produces nothing.

Before detection: The monitoring dashboard shows “6 file reads in 10 turns.” No alarm fires. Reading files is normal behavior.

After detection: The entropy signature shows a flat line at 1 bit over the file-target distribution for 10 consecutive windows, with mutual information between reads and edits near zero. The agent is circling.

Domain Drift

A session begins with a clear focus. The task is to add input validation to a REST endpoint. In Turns 1 through 10, every action targets the `api/` directory. The agent reads `api/handlers.py`, reads `api/validators.py`, edits `api/handlers.py` to add a validation call, reads the test file for the endpoint. The file-domain distribution is concentrated: 100% of actions target the API module. Entropy over the domain distribution is zero. One domain, one focus.

By Turn 25, the picture has changed. The agent is reading `utils/logging.py`. It edits `db/migrations/003_add_index.py`. It opens `cli/arg_parser.py` and modifies a help string. It runs a test in `tests/integration/test_startup.py`. The domain distribution now spans four modules. None of these modules are the API layer where the task lives.

Compute the domain entropy at Turn 10: near zero. Compute it at Turn 25: near 2 bits (four roughly equally visited domains). The entropy has risen steadily across the session. Plot the curve and you see a monotonic increase with no sign of returning to the original focus.

The agent is wandering.

Domain drift differs from legitimate task decomposition. Sometimes a task genuinely requires touching multiple modules. Adding input validation might require updating a shared utility function, modifying a database constraint, and adjusting a test harness. But legitimate decomposition has structure: the agent moves to a new domain, does focused work there, and returns to the original domain. The entropy rises, then falls, then rises, then falls. The signature is sawtooth, not monotonic.

Drift has no return. The agent moves to the logging module and stays there for three turns, then moves to the database layer without returning to the API. Then it moves to the CLI. Each domain visit is a detour that never reconnects to the task. The entropy signature is a staircase that only goes up.

Before detection: The session log shows the agent "working across the codebase." Broad activity could indicate thoroughness.

After detection: The domain entropy signature is monotonically increasing with no return to the task's home domain. The KL divergence between the Turn 5 and Turn 25 action distributions is high (above 1.5 bits). The agent is not decomposing the task. It is drifting away from it.

Edit Thrashing

The agent is implementing a caching layer. It opens `session.py` and edits line 42, adding a cache lookup before the database call. It runs the test suite. Two tests fail. The agent reads the error output, then edits line 42 again, reverting the cache lookup and replacing it with a slightly different version. Runs the tests. One test passes, a different one fails. The

agent edits line 42 a third time, reverting to something close to the original version. Runs the tests. Same two failures as the first attempt.

Three edit-test cycles. The same location. No net progress.

The entropy signature of edit thrashing is distinctive. The *action type* entropy may look healthy: a mix of edits and test runs, which is the normal pattern for implementation work. But the *edit target* entropy is pathological. A single file, a single region, edited and reverted and re-edited. The write-target distribution is maximally concentrated. And the *content* of the edits oscillates rather than converging.

Edit thrashing is the entropy of indecision. The agent cannot commit to a change because it does not understand the constraint it is trying to satisfy. Each edit is a guess. Each test failure sends it back to the same location with a different guess. The search is local, not informed by new information. The agent does not read new files to understand why the tests fail. It just tweaks the same line.

The mathematical signature: low edit-target entropy (one location), high edit-content entropy (many variants), low mutual information between test output and subsequent edits (the test failures do not inform the next attempt in a convergent way). KL divergence between successive edit distributions is near zero because the distribution never changes. It is always "edit line 42 of session.py."

Compare this to healthy iterative development, which can look superficially similar. A skilled agent might edit the same function three times in succession, each time refining its approach based on test feedback. The difference is convergence. In healthy iteration, the edit-content hashes move toward a fixed point. Each version is more similar to the final version than the last. In thrashing, the hashes oscillate. Version 3 is closer to version 1 than to version 2. The agent is not iterating toward a solution. It is bouncing between alternatives it cannot evaluate.

The distinction is measurable. Compute the edit distance (or content hash distance) between successive versions. In convergent iteration, the distances decrease: each revision changes less than the previous one. In thrashing, the distances stay constant or increase. The agent makes changes of the same magnitude every cycle because it is not narrowing the search space. It is sampling from it randomly.

Before detection: The session shows "active development in progress, running tests after each change." This looks like test-driven development.

After detection: The edit-content hash sequence is non-convergent over three cycles. The mutual information between test output and subsequent edits is below 0.1 bits, indicating the agent is not learning from the test failures. The agent is thrashing.

Tool Diversity Collapse

Early in the session, the agent uses a healthy mix of actions. Turns 1 through 10 contain file reads, codebase searches, a test run, two edits, and a shell command to check the project's dependency versions. The action-type distribution is spread across five or six categories. The entropy of the action distribution is near its maximum: around 2.3 bits for six action types.

By Turn 30, the agent is doing nothing but reading files. The last twelve turns consisted entirely of file reads. No edits. No searches. No test runs. No shell commands. The action-type distribution has collapsed to a single mode. Entropy has fallen to zero.

This is tool diversity collapse: the agent's behavioral repertoire has narrowed to a single action type. The agent is technically active. It is consuming context. But it has stopped engaging with the full range of its capabilities.

Tool diversity collapse often follows a failed edit or a confusing test result. The agent attempts something, gets an unexpected outcome, and retreats to the safest action it knows: reading. Reading always succeeds. Reading never produces error messages. Reading feels like progress because new information enters the context. But reading without subsequent action is consumption without production. The agent fills its context window with file contents and never converts that information into work.

The entropy signature is a decline curve. Action-type entropy starts near 2 bits and falls steadily toward zero. The shape is not a sudden drop (which would indicate a deliberate focus shift) but a gradual slide, as non-read actions become less and less frequent over successive windows.

Before detection: The agent appears to be "researching the codebase." Heavy reading is expected early in a session.

After detection: The action-type entropy has been below 0.3 bits for fifteen consecutive windows. This is not research. This is retreat.

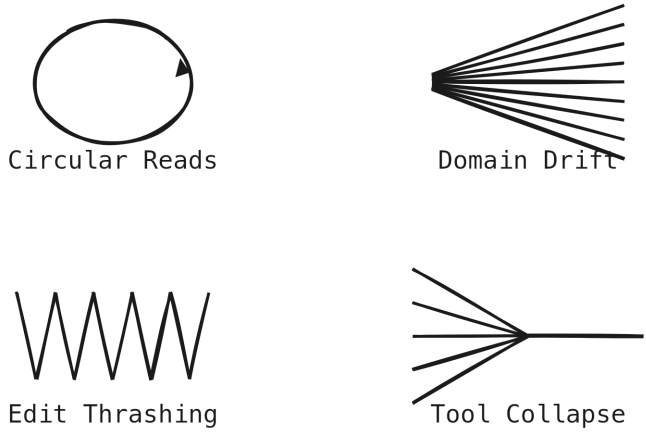

Figure 8.2: Four entropy signatures: circular reads, domain drift, edit thrashing, tool diversity collapse

The Hardest Distinction

Every entropy signature described above has an evil twin: a healthy behavior that looks identical from the wrong angle.

An agent exploring broadly early in a session has high file-scatter entropy. It reads files across multiple directories, touches several modules, jumps between action types. At Turn 3, this looks like productive exploration. At Turn 30, the same pattern would look like domain drift. Same observable shape. Different meaning.

An agent reading the same file three times early in a session might be extracting layered information. The same behavior at Turn 35, after twenty turns of no edits, is circular reading. Same file access pattern. Different context.

An agent with low action-type entropy at Turn 40 might be in the final execution phase, writing code steadily with occasional test runs. The same low entropy at Turn 40 in a session that has produced zero edits is tool diversity collapse. Same entropy value. Different trajectory.

The distinguishing factor is position along the trajectory.

Consider two sessions side by side. Session A and Session B both show a file-scatter entropy of 2.3 bits at Turn 8. In Session A, the agent has just started working. It read the task description two turns ago and is surveying the codebase, opening files across three directories to understand the project structure. This is exploration. The high entropy is healthy. The agent needs to build a map before it can navigate.

In Session B, the agent has been working for thirty turns. It implemented half a solution twenty turns ago, encountered a test failure at Turn 18, and has been reading files in different directories ever since. It has not edited a file in twelve turns. The file-scatter entropy of 2.3 bits at Turn 28 looks numerically identical to Session A at Turn 8. But Session B is lost. The high entropy is not exploration. It is the residue of a broken plan.

A monitoring system that triggers on "file-scatter entropy above 2.0" would either flag both sessions or neither. Both are wrong. The system needs to know that Session A is at Turn 8 and Session B is at Turn 28. It needs to know that Session A's entropy is rising from zero (the start of exploration) while Session B's entropy is rising from 0.8 (the collapse of focused work). Same number at the measurement point. Different trajectories arriving at that number.

Entropy measured at a single point is ambiguous. It is a number without context. An entropy of 0.3 bits over the action-type distribution could mean focused execution (good) or behavioral collapse (bad). An entropy of 2.1 bits over the file-target distribution could mean productive exploration (good) or aimless wandering (bad). The number alone does not tell you which.

The trajectory of entropy resolves the ambiguity. Entropy that starts high and falls is exploration followed by commitment. Entropy that starts low and stays low is either focused execution (if the session is producing output) or stagnation (if it is not). Entropy that starts low, rises, and then falls

is re-exploration triggered by a problem, followed by recovery. Entropy that rises monotonically is drift.

This is why entropy must be measured as a trajectory property, not a frame property. A frame tells you the current value. A trajectory tells you the shape. And the shape is what carries meaning.

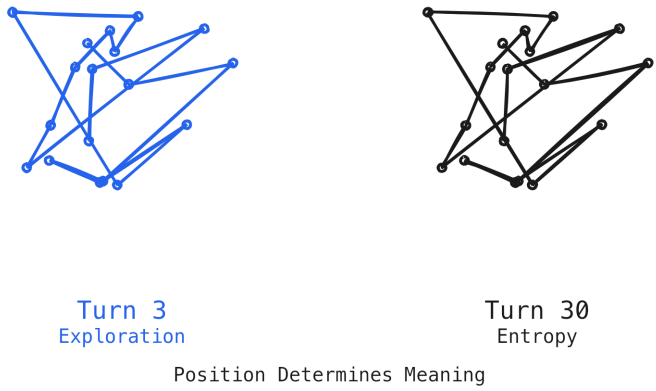

Figure 8.3: Exploration vs entropy: same shape, different meaning depending on trajectory position

The implications for monitoring are direct. Any system that triggers alerts based on instantaneous entropy values will produce false positives constantly. An alert that fires when action-type entropy drops below 0.5 bits will flag every agent in its focused execution phase. An alert that fires when file-scatter entropy exceeds 2.0 bits will flag every agent in its exploration phase. The values cross the thresholds during normal operation.

The correct approach is to condition on trajectory shape. Flag sessions where entropy *rises monotonically* past a window count threshold. Flag sessions where entropy *stays flat* for too many windows relative to the session length. Flag sessions where the rate of entropy change exceeds a bound. These are trajectory-level predicates, not frame-level predicates. They require history.

Building on the coordinate space from Chapter 6, entropy is not a coordinate. It is a derived quantity computed over a sliding window of frames. It lives in the same space as velocity and acceleration: it describes how the trajectory moves, not where it is. A frame's position in the coordinate space tells you what the agent is doing right now. The entropy computed over recent frames tells you whether what the agent is doing right now is coherent with what it was doing recently.

Entropy as a Trajectory Property

The mathematics is straightforward. Define a window size *w* (a common choice is 8 turns, though the optimal size depends on the task complexity). At each turn *t*, collect the frames from turn *t - w + 1* through turn *t*. Within that window, estimate the probability distributions that matter: the action-type distribution, the file-target distribution, the module-domain distribution, the edit-content distribution.

Compute Shannon entropy as before, but over a sliding window of width *w*. The resulting time series H(1), H(2), …, H(T) is the trajectory's entropy signature for that distribution. Plot it and you see the session's coherence story.

But the absolute values of H(t) are less informative than their derivatives. The rate of entropy change tells you what is happening to the session's coherence right now.

Define the entropy rate as the difference between consecutive entropy values:

> dH(t) = H(t) - H(t-1)

A positive dH means entropy is increasing. The agent's behavior is becoming more diffuse. Whether this is good or bad depends on the phase.

A negative dH means entropy is decreasing. The agent's behavior is concentrating. Again, whether this is good or bad depends on context.

A sustained positive dH (entropy increasing for many consecutive windows) is the signature of drift. The session is losing focus without recovering.

A sustained dH near zero after an initial decrease is the signature of commitment. The agent found its strategy and is executing.

A spike in dH followed by a return to the previous level is the signature of a disruption followed by recovery. The agent encountered something unexpected, broadened its behavior to deal with it, and then re-narrowed.

The second derivative matters too. Accelerating entropy increase (positive second derivative) is worse than constant entropy increase. It means the session is losing coherence at an increasing rate. This is the signature of a session that will not recover without intervention. Decelerating entropy increase (negative second derivative with positive first derivative) suggests the drift is slowing and the agent may self-correct.

All of this is computable from the action log. No model internals required. No prompt inspection. No attention-head analysis. The agent's observable behavior, recorded as tool calls with arguments, provides the raw data. The sliding window provides the structure. Shannon's formula provides the number.

There is a subtlety in the window size. Too small a window (say, 3 turns) and the entropy estimate is noisy. Three turns is not enough data to estimate a distribution reliably. The entropy will jump around due to sampling noise, not behavioral changes. Too large a window (say, 30 turns) and the estimate is smooth but sluggish. A real behavioral change at Turn 20 will take until Turn 35 to fully register in the entropy computation. The signal is delayed.

A window of 8 to 12 turns is a practical starting point for coding agents. It is large enough to smooth out the natural variation between turns (an agent might read two files in a row simply because both were needed, not because it has entered a read-only mode) and small enough to detect behavioral shifts within a few turns of their onset.

For longer sessions (50+ turns), it can be useful to compute entropy at multiple window sizes simultaneously. Short-window entropy captures rapid oscillations (edit thrashing cycles of 3-4 turns). Long-window entropy captures gradual trends (domain drift accumulating over 20 turns). The combination provides both detail and context.

There is also the question of which distributions to track. Action-type entropy (read, edit, search, test, shell) is the coarsest signal and the most sta-

ble. It changes slowly because agents tend to shift between behavioral modes gradually. File-target entropy is finer-grained and more volatile: the specific files an agent accesses change rapidly even within a coherent session. Module-domain entropy sits in between: coarser than individual files, finer than action types.

Each distribution captures a different dimension of coherence. A session can have stable action-type entropy (healthy mix of reads and edits) while its module-domain entropy rises without bound (the reads and edits are scattered across unrelated parts of the codebase). This is domain drift that would be invisible to action-type entropy alone. Conversely, a session can have collapsing action-type entropy (all reads, no edits) while its file-target entropy remains high (reading many different files). This is tool diversity collapse combined with broad but aimless exploration.

No single entropy time series tells the full story. The combination of multiple entropy series, tracked simultaneously over the same windows, produces a multidimensional coherence signal. This is the full entropy signature of a trajectory: not one curve, but several, each measuring coherence along a different axis of behavior.

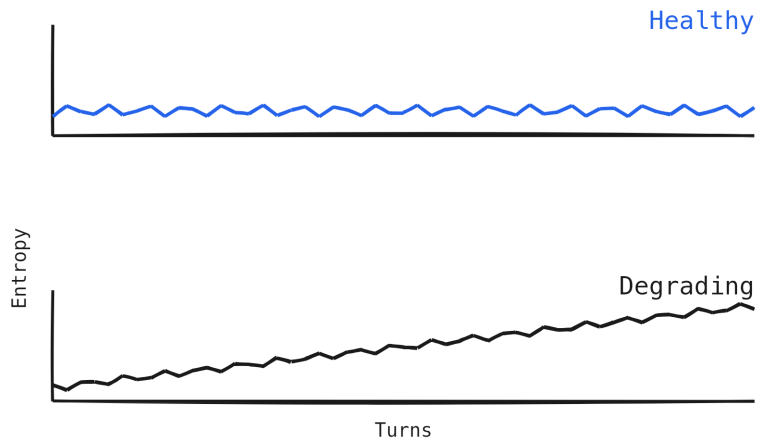

Figure 8.4: Entropy over time: healthy session vs degrading session

The Relationship Between Entropy and the Other Dynamics

Chapter 7 introduced inertia: the tendency of an agent trajectory to continue in its current direction. Inertia describes momentum. It tells you how likely the agent is to maintain its current behavior. An agent with high inertia in a reading mode will continue reading. An agent with high inertia in a focused editing mode will continue editing.

Entropy describes something complementary. It tells you whether the current behavior is coherent. An agent can have high inertia and high entropy simultaneously: it has strong momentum, but that momentum is carrying it through diffuse, unfocused actions. This is the signature of an agent that is confidently wandering. It will keep wandering. The inertia ensures continuation. The entropy tells you that what is being continued is not productive.

Conversely, an agent can have high inertia and low entropy. It has strong momentum and focused behavior. This is the signature of productive deep work. The agent is locked in. It will keep making progress. Inertia predicts the continuation. Entropy confirms the coherence.

Low inertia and rising entropy is the most dangerous combination. The agent has no momentum in any direction and is becoming increasingly diffuse. It has lost both its heading and its structure. This is a session that is about to fail.

Low inertia and low entropy is rare but interesting. The agent is focused (low entropy) but not committed to a direction (low inertia). This happens at decision points: the agent has narrowed its attention to a small area of the codebase but has not yet decided what to do there. It is about to commit. The next few turns will determine whether inertia builds in a productive direction.

These combinations are not exhaustive, and the real world produces messy intermediate states. But the two-axis picture (inertia on one axis, entropy on the other) provides a useful mental model for session health. Healthy sessions tend to start in the high-entropy, low-inertia quadrant (exploring, uncommitted), move through the high-entropy, rising-inertia quadrant (exploring with increasing direction), and settle into the low-entropy, high-inertia quadrant (focused, committed, making progress). Unhealthy ses-

sions wander through the quadrants without converging, or get stuck in the high-entropy, high-inertia quadrant (confidently going nowhere).

Together, inertia and entropy predict whether a trajectory will succeed or fail before it finishes. Inertia tells you where the trajectory is going. Entropy tells you whether "where it is going" is a meaningful concept. A trajectory with strong inertia toward a coherent goal (low entropy) will reach that goal. A trajectory with strong inertia but no coherent goal (high entropy) will consume its context window producing nothing useful. A trajectory with no inertia (the agent keeps changing direction) and rising entropy is already lost.

This predictive power is the payoff of treating entropy as a trajectory property rather than a frame property. A single entropy measurement is a number. A trajectory of entropy measurements, combined with inertia data, is a forecast.

Measuring What Matters

The four entropy signatures described in this chapter are patterns, not rules. Real sessions produce messier signals. An agent might exhibit circular reads for five turns, break out of the loop, do productive work for ten turns, and then fall into edit thrashing. The entropy signature of such a session is not a clean exemplar of any single pattern. It is a composite, with different failure modes appearing at different points in the trajectory.

This is expected. The value of the entropy framework is not in classifying sessions into neat categories. It is in providing a continuous, computable measure of coherence that can be tracked over time and combined with other trajectory properties. The entropy number does not replace human judgment about whether a session is going well. It gives human judgment (or automated monitoring systems) something to work with beyond raw tool call logs.

A monitoring system that computes entropy over sliding windows and tracks its trajectory can answer questions that a tool call counter cannot. Not "how many reads has the agent done?" but "is the agent's read pattern expanding or cycling?" Not "how many files has the agent touched?" but "is the file scatter increasing without corresponding task decompo-

sition?" Not "how many edits has the agent made?" but "are the edits converging toward a solution or oscillating between alternatives?"

These questions are about coherence. Coherence is what entropy measures. And coherence is what distinguishes a productive thirty-turn session from a thirty-turn session that produced nothing of value while consuming the same resources.

As established earlier, entropy is not a health metric to minimize. The pathology is not high entropy but high entropy that does not resolve — entropy in the wrong phase, persisting past the point where the agent should have committed. The trajectory shape, not the instantaneous value, is what matters.

Entropy gives us the vocabulary to describe coherence loss with precision. Circular reads, domain drift, edit thrashing, tool diversity collapse: these are not vague impressions of a session going wrong. They are specific statistical signatures, each with a distinct fingerprint in the entropy time series, each detectable from standard agent logs. The agent does not need to report that it is confused. Its behavior reports it.

Inertia is about momentum. Entropy is about coherence. Together they form the first two dimensions of a trajectory diagnostic that can predict session outcomes while the session is still running. The next chapter adds a third: the spatial structure of the agent's work across the codebase.

Chapter 9

Space

Some codebases are a pleasure to navigate. Open a file. The imports tell you where the dependencies live. The function names tell you what they do. The module boundaries correspond to conceptual boundaries. You can hold the architecture in your head after thirty minutes of reading because the architecture *wants* to be held. It has structure that rewards comprehension.

Other codebases are hostile terrain. Circular imports. God objects. A `utils.py` with four thousand lines and no organizing principle. Functions named `process_data` that do seventeen different things depending on which flags you pass. You open a file, follow an import, follow another import, and find yourself back where you started, having learned nothing except that the dependency graph has cycles.

Human developers experience the difference viscerally. The first codebase feels like walking on a trail. The second feels like hacking through brush with a machete.

AI agents experience the same difference. Not viscerally. But measurably.

An agent working in a well-structured codebase produces trajectories with particular characteristics: purposeful movement, efficient information gathering, clean transitions from exploration to commitment. The same agent, same model, same scaffolding, working in a tangled codebase produces different trajectories: more backtracking, more re-reading, longer orientation phases, more frequent false starts.

The agent did not change. The space did.

Chapter 7 introduced inertia: the way accumulated context pulls a trajectory forward. Chapter 8 introduced entropy: the way coherence degrades over time. Both of these forces operate on the trajectory itself. This chap-

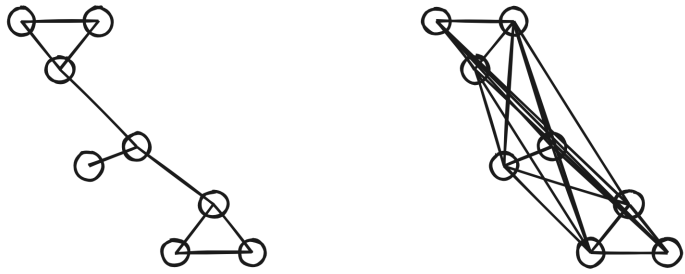

Figure 9.1: Clean modular codebase vs tangled codebase and their effect on trajectory shape

ter introduces the third force, and it operates on the environment the trajectory moves through.

Space is the terrain.

The Workspace as a Graph

Start concrete.

A codebase is a collection of files. Those files are not independent. They reference each other, depend on each other, call into each other. The structure of those relationships defines the shape of the workspace.

Represent it as a graph. Each file is a node. Each relationship between files is an edge. The graph is the map.

But not all edges are the same. Three types of relationship matter, and they carry different information.

Structural edges. These are the explicit dependencies. `api.py` imports `auth.py`. `session_manager.py` imports `database.py`. `main.py` imports ev-

erything. These edges are visible in the source code. You can extract them mechanically by parsing import statements, require declarations, include directives. The structural graph is the dependency graph, and it is the most obvious layer of codebase topology.

Structural edges tell you what code *needs* to function. If `api.py` imports `auth.py`, then understanding `api.py` requires understanding at least the interface of `auth.py`. An agent that reads `api.py` and encounters a call to `authenticate()` must, at some point, read `auth.py` to understand what that function does. The structural edge creates a navigation obligation.

Semantic edges. These are the conceptual relationships. Two files might share no imports but address the same domain concept. `user_model.py` and `user_serializer.py` do not import each other, but they both deal with users. `payment_processor.py` and `refund_handler.py` operate on different flows, but they share the concept of financial transactions.

Semantic edges are not declared anywhere. They are latent. Discovering them requires understanding what the code is *about*, not just what it imports. Text embeddings can surface them: files whose contents embed to nearby points in the semantic space are likely related, even without structural connections.

For agents, semantic edges matter because they represent potential relevance. An agent investigating a bug in user authentication might need to look at `user_model.py` even though no import chain connects it to the file where the bug manifests. The semantic relationship is the clue. An agent that navigates only along structural edges will miss it.

Historical edges. These are the temporal coupling relationships. Two files that are frequently modified in the same commit have a historical connection. They change together. This co-change pattern often reveals dependencies that neither the structural nor semantic graph captures.

Consider `config.yaml` and `feature_flags.py`. One is a configuration file. The other is a Python module. They share no imports and no obvious semantic relationship. But every time someone adds a new feature flag, both files change. The configuration file gets a new entry. The Python module gets a new constant. They are coupled by the development process, not by the code itself.

Historical edges are extracted from version control history. Count how often pairs of files appear in the same commit. Weight by recency: co-changes from last week matter more than co-changes from two years ago. The result is a graph where edge weights represent the strength of temporal coupling.

For agents, historical edges are navigational shortcuts. An agent that has just modified `config.yaml` should probably check `feature_flags.py`, even if no import or semantic relationship suggests it. The historical pattern knows something the code structure does not.

Three layers. Three graphs. The same set of nodes (files), with three different edge sets representing three different kinds of relationship. Overlay them, and you get a multi-relational graph that captures the full topology of the workspace.

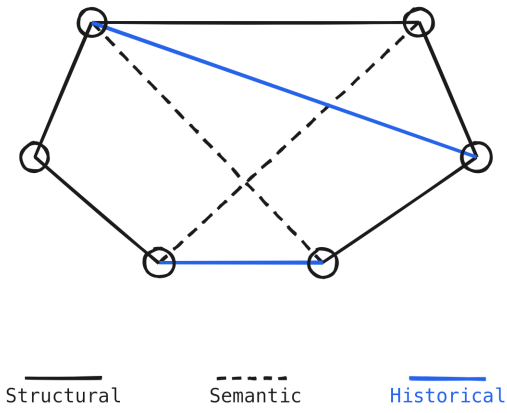

Figure 9.2: Three edge types: structural, semantic, and historical

From Graph to Geometry

A graph is a useful abstraction. But a graph is discrete. Node A is connected to Node B, or it is not. The relationship is binary. You can count hops between nodes (shortest path length), but you cannot say that Node A is "3.7 units" from Node B in any continuous sense.

Trajectories, as defined in Chapter 3, live in continuous space. Embeddings give agent states real-valued coordinates. Distances are smooth, not stepped. To understand how workspace structure constrains trajectories, we need to move from the discrete graph to continuous geometry.

This is what graph embeddings do.

Chapter 3 introduced the idea briefly: Graph Neural Networks (GNNs) learn to assign each node a vector so that the structure of the graph is preserved in the geometry of the vectors. Nodes that are close in the graph end up close in the embedding space. Nodes that are far apart in the graph end up far apart. The discrete adjacency information gets distilled into continuous coordinates.

The mechanism deserves a closer look, because the details matter for how space affects trajectories.

A GNN works in rounds. Each node starts with some initial feature vector. In a codebase graph, this might be the text embedding of the file's contents. So each node already has a position in semantic space. The GNN's job is to adjust that position based on graph structure.

In round one, each node aggregates information from its immediate neighbors. The node's feature vector gets updated: a function of its current features and the features of everything one hop away. After this round, each node's representation encodes not just its own content but the content of its neighbors.

In round two, the process repeats. Each node aggregates from its (already-updated) neighbors. Now each node's representation reflects a two-hop neighborhood. Its content, plus its neighbors' content, plus their neighbors' content.

After k rounds, each node encodes information about its k-hop neighborhood. The resulting vectors capture local structure: which files are nearby, what those files contain, how the local region of the graph is organized.

The key insight: the GNN does not just propagate graph distance. It propagates *structured information* along graph edges. A file that sits between two dense clusters will get features from both clusters, producing a representation that reflects its bridging role. A file at the periphery, connected to only one other file, will get features from a narrow neighborhood, producing a representation that reflects its isolation.

The geometry of the resulting embedding space mirrors the topology of the graph. Dense clusters in the graph become dense regions in the space. Sparse bridges become geometric bottlenecks. Central hubs become points equidistant from many clusters.

Now combine this with the text embeddings from Chapter 3. Each file has two sets of coordinates: content coordinates (what the file says) and structural coordinates (where the file sits). Concatenate them, and each file occupies a position in a joint space that encodes both meaning and architecture.

This joint space is the terrain the agent navigates.

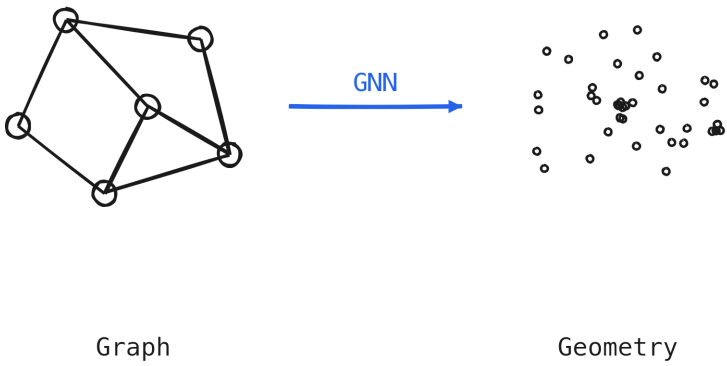

Figure 9.3: Graph to embedding: GNN transforming discrete graph structure into continuous geometry

How Space Constrains Trajectories

An agent does not teleport. When it reads a file, the files it is most likely to read next are the files it can *find* from the current position. Adjacent files in the import graph. Files returned by a codebase search seeded by the current context. Files referenced in the current file's documentation.

The workspace graph constrains which transitions are natural and which require effort.

Consider a codebase with two well-separated modules. A `payments` module with twelve files, densely interconnected. An `auth` module with eight files, also densely interconnected. Between the two modules, a single file: `checkout.py`, which imports from both.

In graph terms, this is two dense clusters connected by a bridge.

In the embedding space, the `payments` files cluster together. The `auth` files cluster together. `checkout.py` sits between them, closer to both clusters than either cluster is to the other, but not fully inside either.

An agent working inside the `payments` cluster moves easily. Each file it reads leads naturally to other files in the cluster. The import relationships are dense. Semantic similarity is high. The agent can explore the full module with short, efficient steps. Each transition covers a small distance in the embedding space.

When the agent needs to cross to the `auth` module, it hits the bridge. The transition through `checkout.py` is a longer step: the embedding distance between the `payments` cluster and `checkout.py` is larger than the typical within-cluster distance. And the transition from `checkout.py` to the `auth` cluster is another long step.

This is not a metaphor. It is measurable. Plot the step sizes along the trajectory (the embedding distance between consecutive states). Within a cluster, step sizes are small and consistent. At a cluster boundary, step sizes spike. The spike marks the crossing.

The topology creates corridors and barriers. Dense clusters are corridors: the agent moves through them fluidly, with many available paths and short distances between nodes. Sparse regions between clusters are barriers: the agent must traverse longer distances with fewer available

paths. Bridge nodes like `checkout.py` are chokepoints: the only viable path between regions.

Three consequences follow from this structure.

Consequence one: cluster trapping. An agent that enters a dense cluster tends to stay in it. Not because it is stuck, but because the cluster's density means there is always another relevant file to read, another connection to follow. The cluster provides a steady supply of small, rewarding transitions. Moving to a different cluster requires a deliberate decision to cross a gap, and the gap appears (in the agent's experience) as a lack of obvious next steps within the current region.

This interacts with inertia. The accumulated context about the current cluster makes within-cluster transitions even more natural: the agent has already built up knowledge about this module, so each new file within the module is partially understood before it is read. Cross-cluster transitions are doubly costly: longer embedding distance *and* less relevant accumulated context.

Consequence two: bridge sensitivity. The agent's ability to navigate between modules depends heavily on bridge nodes. If the bridge is well-documented, clearly named, and easy to discover (it shows up in search results, it is referenced in comments), the crossing is smooth. If the bridge is obscure (a utility function buried in a shared helpers file, unreferenced in any documentation), the crossing may never happen. The agent cannot traverse a bridge it cannot find.

This has practical implications for codebase design. The navigability of a codebase, for both humans and agents, depends not just on the quality of individual files but on the visibility of inter-module connections. A codebase where cross-module dependencies are explicit and well-named creates a space where agents can navigate across boundaries. A codebase where cross-module connections are implicit and buried creates a space with invisible walls.

Consequence three: search as teleportation. There is one mechanism that allows agents to escape cluster trapping: search. When an agent runs a codebase search (grep, semantic search, symbol lookup), it can jump to any node in the graph, regardless of graph distance from the cur-

rent position. Search is the closest thing to teleportation in workspace navigation.

But search is noisy. A broad search returns many results, and the agent must evaluate which results are relevant. This evaluation costs context. A narrow search might miss relevant results entirely. The effectiveness of search depends on the agent's ability to formulate queries that bridge the gap between its current understanding and the location of relevant information.

In the embedding space, a successful search produces a large, targeted jump: the agent moves from one region to a distant region that turns out to be relevant. An unsuccessful search produces a large, random jump: the agent moves to a distant region that turns out to be irrelevant, wasting context on a dead end.

The workspace graph determines the base navigation cost. Search modifies that cost, sometimes dramatically. An agent with good search strategies can navigate a poorly-structured codebase more effectively than an agent without them. But good structure reduces the need for search in the first place.

Measuring Space

If space constrains trajectories, then measuring space gives us predictive power. Before an agent starts working, before a single turn has been taken, we can look at the workspace and make predictions about what kinds of trajectories are likely.

Several quantities capture useful properties of workspace topology.

Clustering coefficient. For a given node, what fraction of its neighbors are also connected to each other? A high clustering coefficient means dense local structure: the agent's neighborhood is tightly interconnected. A low clustering coefficient means sparse local structure: the agent's neighbors do not connect to each other, suggesting the node sits at an intersection rather than inside a module.

The average clustering coefficient across the whole graph tells you how modular the codebase is. Highly modular codebases have high average clustering: files within each module are densely connected, forming tight

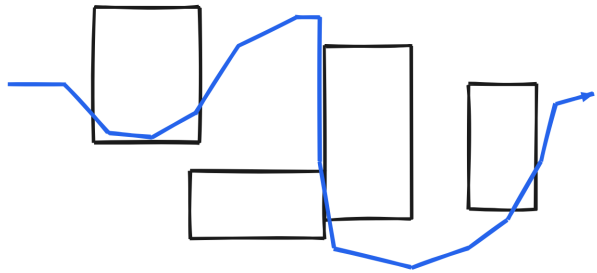

Figure 9.4: Trajectory constrained by workspace topology: corridors, barriers, and chokepoints

clusters. Poorly modular codebases have low average clustering: connections are spread diffusely across the graph without forming coherent groups.

Graph diameter and average path length. The diameter is the longest shortest path between any two nodes. The average path length is the mean shortest path across all pairs. Together, they tell you how "wide" the space is. A codebase with a short average path length is compact: any file is reachable from any other file in a few hops. A codebase with a long average path length is sprawling: some files are many hops apart.

Average path length correlates with navigation difficulty. The more hops between two relevant files, the more intermediate files the agent must traverse (or the more it must rely on search to skip ahead). Long path lengths mean more opportunities for the agent to get sidetracked during navigation.

Betweenness centrality. How often does a given node appear on shortest paths between other nodes? A node with high betweenness centrality is a bottleneck: many paths through the graph pass through it. In a code-

base, high-betweenness files are the connective tissue: shared utilities, core abstractions, central configuration files.

These files are disproportionately important for navigation. An agent that discovers a high-betweenness node gains access to many regions of the codebase through a single point. An agent that misses such a node may struggle to move between regions.

Spectral gap. This is a property of the graph's Laplacian matrix, and it measures how well-connected the graph is overall. A large spectral gap means the graph is tightly connected: information (and agents) can flow easily between any two regions. A small spectral gap means the graph is loosely connected, possibly approaching disconnection: some regions are nearly isolated.

The spectral gap predicts how much trouble agents will have with cross-module tasks. Tasks that require understanding multiple, weakly-connected parts of the codebase are harder in graphs with small spectral gaps, because the space between those parts is large and sparsely bridged.

These metrics are computable from the codebase graph before any agent session begins. They characterize the terrain. They do not determine the trajectory (the agent's decisions still matter), but they constrain the distribution of likely trajectories. Certain paths through the space are easy. Others are hard. The graph metrics tell you which is which.

A concrete comparison makes this tangible. Take two real codebases. Codebase A is a well-maintained web framework: 340 files, average clustering coefficient 0.71, average path length 2.8, largest spectral gap 0.45. Codebase B is a legacy monolith of similar size: 380 files, average clustering coefficient 0.23, average path length 5.1, spectral gap 0.08.

Run the same agent on comparable tasks in both. In Codebase A, trajectories tend to be short and directed. The agent finds relevant files quickly because dense clusters channel navigation. Cross-module transitions are smooth because the spectral gap indicates strong connectivity. Median session length: 14 turns.

In Codebase B, trajectories are longer and more erratic. The agent takes detours because low clustering means local neighborhoods do not cohere into meaningful modules. Cross-module transitions are expensive

because the low spectral gap means regions are weakly connected. The same agent, solving the same kind of problem, takes a median of 31 turns.

The agent did not get dumber between the two codebases. The terrain got harder. The graph metrics predicted this before a single session was run.

Space Is Not Static

Everything described so far treats the workspace as fixed terrain. The agent moves through it, and the workspace stays put. But this is only half the picture.

The agent modifies the space as it works.

Consider an agent tasked with adding a new feature: a rate limiter for an API. The agent reads the existing API handlers. It reads the middleware stack. It forms a plan. Then it starts implementing.

It creates a new file: `rate_limiter.py`. A node is added to the graph.

It adds an import from `api.py` to `rate_limiter.py`. A structural edge is created.

It adds another import from `config.py` to `rate_limiter.py`, so the rate limiter can read its configuration. Another structural edge.

It modifies `middleware.py` to invoke the rate limiter in the request pipeline. A new structural edge from `middleware.py` to `rate_limiter.py`, and a semantic connection between the rate limiter and the rest of the middleware stack.

Four actions. The graph has a new node and three new edges. The topology has changed. The clustering structure of the `api` module is different now. The path lengths between certain nodes have changed. The embedding space, if recomputed, would shift.

The agent is not just moving through the space. It is reshaping the space it moves through.

This is a feedback loop. The current state of the space influences the agent's next action (through navigation constraints and information accessibility). The agent's action modifies the space (by creating or modifying files). The modified space influences the next action. And so on.

The feedback loop has directional tendencies. Agents tend to increase local density: they add files and connections near where they are already working, reinforcing the cluster they are in. They rarely add connections that bridge distant clusters (because creating such bridges requires awareness of both clusters, which requires having already navigated between them).

This means that agent work tends to amplify existing structure. Dense regions get denser. Sparse regions stay sparse. Modules that are already well-connected become more interconnected. Gaps between modules persist or widen.

This amplification is not inherently good or bad. In a well-structured codebase, the agent reinforces good structure: it adds new files to the appropriate module, with appropriate dependencies, following existing patterns. The space becomes more navigable over time.

In a poorly-structured codebase, the agent reinforces poor structure. It adds files to an already-tangled region, creating more connections in an already-overcrowded neighborhood. The space becomes less navigable over time.

The quality of the starting structure predicts the quality of the ending structure. Space begets more of itself.

There is a subtler effect. When the agent modifies the space, it also changes the navigation cost for its own future actions. Adding `rate_limiter.py` with imports from `api.py` and `config.py` creates a new path between the API layer and the configuration layer. Before the agent's work, navigating from `api.py` to `config.py` required traversing whatever path already existed. After, there is a new two-hop path through `rate_limiter.py`. The agent has shortened the distance between two regions of the codebase, for itself and for any future agent.

This self-modification means that the difficulty of a task depends on when during the session it is encountered. A sub-task that requires navigating between two distant modules is harder at the beginning of the session, before the agent has created any bridging files, and easier later, after the agent's own work has added connective tissue. The trajectory literally paves the road it travels.

Multi-agent scenarios amplify this effect. Two agents working in the same codebase simultaneously modify the graph concurrently. Agent A creates a new utility module. Agent B, working in a different region, discovers that module and uses it as a bridge. Neither agent planned this. The graph's evolution, driven by both agents' actions, created an emergent navigational shortcut that neither agent deliberately built.

Space and the Other Forces

Space does not act in isolation. It interacts with inertia and entropy in specific, measurable ways.

Space amplifies inertia. An agent deep inside a dense cluster has strong inertia: lots of accumulated context about that cluster. The cluster's density provides a constant supply of related files to explore, keeping the agent on its current path. In sparse space, inertia has less to work with. There are fewer related files to pull the agent forward, so the inertial force weakens. The agent is more likely to stop, reconsider, or search for a different region.

Dense clusters create inertial traps. Sparse regions create inertial voids. The topology determines where inertia is strongest.

Space modulates entropy. In a well-structured codebase, the agent's actions tend to stay coherent longer. The clear module boundaries act as guides, channeling the agent's exploration along productive paths. Entropy increases slowly because the space itself provides organization.

In a poorly-structured codebase, entropy increases faster. The agent makes more wrong turns, follows more misleading connections, wastes more context on irrelevant exploration. The lack of structure in the space translates to lack of coherence in the trajectory.

Concretely: the same agent, given the same task, will exhibit lower entropy rates in a well-structured codebase than in a poorly-structured one. The space acts as a coherence subsidy. Good structure donates coherence to the trajectory. Bad structure taxes it.

Space creates entropy gradients. Even within a single codebase, some regions are well-structured and others are not. An agent moving from a

clean module into a tangled one will experience increasing entropy: its actions become less predictable, its exploration less efficient, its coherence lower. Moving from tangled into clean, entropy decreases. The spatial structure creates gradients that the trajectory flows along.

These interactions mean that predicting trajectory behavior requires knowing all three forces. An agent with low inertia in dense space will behave differently from an agent with high inertia in sparse space. An agent in well-structured space with high accumulated context will resist entropy differently from an agent in poorly-structured space with the same context load.

The three forces form a system. Separating them is analytically useful. Understanding their interaction is where the predictive power lives.

The Terrain Before the Walk

Hikers study topographic maps before setting out. The map does not tell them where they will walk, exactly. It tells them where walking is easy, where it is hard, where the ridgelines run, where the valleys channel movement.

Codebase graphs serve the same function for agent trajectories. Compute the graph. Measure the clustering, the path lengths, the centrality, the spectral gap. Identify the dense modules and the sparse bridges. Map the terrain.

Then watch the agent walk. Its trajectory will follow the contours of the space, funneled through corridors, deflected by barriers, trapped in clusters, liberated by search. The terrain does not determine the path. But it shapes the distribution of likely paths so strongly that ignoring it means ignoring a primary force.

Inertia is about where the trajectory has been. Entropy is about how coherent it remains. Space is about the ground beneath it. All three interact. All three are measurable. Together, they constitute the dynamics of agent behavior: the forces that give trajectories their shape.

Part III has now described all three. The next chapter asks a different question: do these forces obey deeper laws?

Chapter 10

Conservation and Symmetry

In physics, the deepest laws are not about specific objects or specific motions. They are about what stays the same.

A ball rolls down a hill. It speeds up. Its height decreases and its velocity increases. Two quantities are changing simultaneously, in opposite directions. But if you compute the total energy (kinetic plus potential), the number does not change. Energy is conserved. The ball trades height for speed, and the exchange rate is exact.

Conservation of energy is not a description of any particular ball on any particular hill. It is a constraint that every ball on every hill obeys. It limits what is possible. A ball that starts at a certain height cannot end up moving faster than the speed that height's worth of energy permits. The conservation law does not tell you the trajectory. It tells you the boundary that all trajectories must respect.

Physics is full of these laws. Energy is conserved. Momentum is conserved. Angular momentum is conserved. Electric charge is conserved. Each conservation law narrows the space of possible futures. Together, they are so constraining that they often determine the outcome without needing to trace the full trajectory at all.

Does anything like this exist for agent trajectories?

The previous three chapters described forces: inertia, entropy, space. Forces push trajectories in particular directions. But forces are local. They act on the trajectory at each moment, shaping the next step. Conservation laws are global. They constrain the trajectory as a whole, limiting what any sequence of steps can achieve.

If context dynamics has conservation laws, even approximate ones, they would be powerful. They would tell us what is possible and what is not, before the agent takes a single step.

This chapter explores that question honestly. Some of what follows is observed in data. Some is hypothesized from patterns that appear consistent but have not been tested at scale. Some is outright speculative, extending the physics analogy beyond where current evidence reaches. Each claim will be labeled accordingly.

The labels matter. A field built on unmarked speculation is a field headed for trouble. A field that refuses to speculate never discovers anything new. The balance is in knowing which is which.

The Capacity Constraint

Status: observed.

An agent has a context window. That window has a fixed size, measured in tokens. Everything the agent knows about the current task, every file it has read, every action it has taken, every result it has received, must fit within that window (or be summarized, compressed, or discarded to make room). The context window is the agent's working memory, and it is finite.

This is the most obvious constraint on agent behavior, and the most obviously "conserved" quantity, in a loose sense. The agent cannot create context capacity from nothing. Every observation it makes consumes capacity. Every action recorded consumes capacity. The total is bounded.

But the analogy to energy conservation is imperfect. Energy conservation is exact: the total before equals the total after, with no losses and no external inputs. Context capacity is more like a budget than a conserved quantity. The agent spends it. It does not transform it into something else and get it back.

Still, the budget creates real constraints. An agent that spends most of its context window on exploring irrelevant files has less capacity remaining for the files that matter. An agent that generates verbose intermediate reasoning consumes capacity that could have held additional observations. The budget forces tradeoffs.

The interesting question is whether those tradeoffs follow predictable patterns. If an agent spends x tokens on exploration and y tokens on exploitation, is the product x times y bounded? Is there an optimal ratio? Does the ratio depend on the task, or on the space, or on both?

Early data suggests that the exploration-exploitation tradeoff in context consumption follows a pattern reminiscent of the uncertainty principle in quantum mechanics. Agents that explore broadly (reading many files, gathering diverse information) tend to produce shallower understanding of each individual file. Agents that explore narrowly (reading fewer files in greater depth) produce deeper understanding but miss relevant context that lives elsewhere.

This is not a metaphor imported from physics for rhetorical effect. It is a measurable phenomenon. Define "exploration breadth" as the number of distinct files read per session. Define "comprehension depth" as how well the agent's reading of a file informs its subsequent edits to that file. Plot breadth versus depth across hundreds of sessions.

The result: a roughly hyperbolic relationship. As breadth increases, depth decreases. The product of breadth and depth hovers around a characteristic value that depends on the context window size and the information density of the codebase.

This is not a conservation law in the physics sense. The product is not exactly constant. It varies by task, by agent, by codebase. But it is bounded, and the bound appears to be set by the context capacity. The agent cannot have both unlimited breadth and unlimited depth, because both consume the same finite resource.

Label: the existence of a tradeoff is observed. The specific mathematical form of the tradeoff (hyperbolic, with a product related to context capacity) is hypothesized based on limited data. The analogy to the uncertainty principle is illustrative, not structural.

The Inertia-Entropy Exchange

Status: hypothesized.

Chapter 7 defined inertia: the tendency of a trajectory to continue in its current direction, driven by accumulated context. Chapter 8 defined entropy: the degradation of coherence over time. These two forces act in opposite directions. Inertia keeps the trajectory on course. Entropy pulls it off course.

Is there a conserved quantity in the exchange between them?

Here is the observation that motivates the hypothesis. In many sessions, there is a characteristic pattern: early turns have low inertia (the agent has not committed to a direction) and low entropy (the agent's actions are coherent because the task is fresh and clear). As the session progresses, inertia increases (the agent builds up context and commits to a path) and entropy also increases (coherence gradually degrades as the context accumulates noise and digression).

Plot inertia and entropy on the same time axis. They tend to increase together. Not in lockstep, but correlated. Sessions where inertia grows rapidly also tend to accumulate entropy rapidly. Sessions with slow inertia growth tend to have slower entropy growth.

One interpretation: inertia and entropy are both consequences of the same underlying process. Context accumulation. As the agent accumulates context, it gains momentum (inertia) but also gains noise (entropy). The context is simultaneously the source of the trajectory's drive and the source of its degradation.

If this interpretation is correct, then inertia and entropy are not independent forces. They are two aspects of a single process: the growth of context over time. And the "exchange rate" between them would be determined by the information density of the context: how much signal versus noise the agent accumulates at each step.

A conservation law would say: the sum of inertia and "negative entropy" (sometimes called negentropy, or coherence) is constant. As inertia grows, coherence decreases by a corresponding amount. The total is conserved.

This is too strong. The total is not constant. External factors (the structure of the space, the difficulty of the task, the quality of the agent's search strategy) all affect how much coherence is lost per unit of inertia gained. A well-structured codebase donates coherence, as Chapter 9 described, allowing the agent to gain inertia without proportional entropy increases.

A weaker version might hold: for a given workspace and task difficulty, the rate of coherence loss per unit of inertia gain is approximately constant. The exchange rate is fixed by the environment, even if the totals are not conserved.

This weaker version is testable. Measure inertia (as the cosine similarity between the trajectory's current direction and its direction over the last

N steps). Measure coherence (as the mutual information between the agent's reads and writes). Compute the ratio of coherence lost to inertia gained across sessions in the same codebase on the same task type. If the ratio is consistent, the weak exchange hypothesis holds.

Early experiments show the ratio is consistent within a factor of two for sessions in the same codebase with the same task type. Across different codebases, the ratio varies by an order of magnitude.

Label: the correlation between inertia growth and entropy growth is observed. The interpretation as an exchange driven by context accumulation is hypothesized. The existence of a fixed exchange rate within controlled conditions is provisionally supported by limited data. The claim that this constitutes a conservation law is speculative.

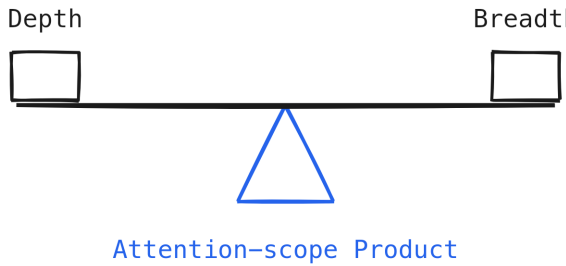

Figure 10.1: Conservation as a balance scale: candidate conserved quantities in context dynamics

The Attention-Scope Product

Status: speculative.

Here is a quantity that might be conserved but for which evidence is thin.

An agent can focus narrowly or broadly. It can examine one file in extreme detail, tracing every function call, reading every comment, building a complete mental model of a single module. Or it can skim many files, building a shallow understanding of the codebase's overall architecture.

Call the depth of focus *attention*. Call the breadth of coverage *scope*. Both are measurable (attention as the number of tokens spent per file; scope as the number of files visited per session).

The speculation: attention times scope is approximately constant for a given context window size. An agent cannot attend deeply to a broad scope, or broadly to a deep focus. The product is bounded by the context capacity.

This looks similar to the capacity constraint discussed above, and indeed it may be the same constraint expressed differently. But there is a wrinkle. Attention and scope are not just about token consumption. They are about the agent's cognitive strategy.

An agent might spend the same number of tokens in two sessions but allocate them differently. In Session A, it reads ten files at five hundred tokens each. In Session B, it reads two files at twenty-five hundred tokens each. Both sessions consume five thousand tokens of context on file reads. But the attention-scope products are different: 500 times 10 = 5000 versus 2500 times 2 = 5000. In this toy example, the product is the same. Is that a coincidence?

For real sessions, the product varies. Tasks that reward breadth (refactoring across many files) tend to produce lower attention and higher scope. Tasks that reward depth (debugging a subtle race condition) tend to produce higher attention and lower scope. But the product fluctuates more than a true conservation law would predict.

The honest assessment: attention-scope tradeoffs exist. They are shaped by context capacity. But calling the product "conserved" overstates the current evidence. It may be bounded rather than conserved, and the bound may depend on factors (task type, agent architecture, workspace structure) that make it less universal than a true conservation law.

Label: the existence of an attention-scope tradeoff is observed. The constancy of the product is speculative, with some data supporting boundedness but not conservation.

Scale Invariance

Status: partially observed.

In physics, a symmetry is a transformation that leaves something unchanged. Rotate a circle, and it still looks like a circle. Translate a physics experiment from one lab to another, and the laws still work the same way. Symmetries are powerful because they constrain what forms the laws can take. Emmy Noether proved in 1918 that every continuous symmetry corresponds to a conservation law: translational symmetry gives conservation of momentum, rotational symmetry gives conservation of angular momentum, time symmetry gives conservation of energy.

Do agent trajectories have symmetries?

The first candidate: scale invariance. Do the same patterns appear in short sessions and long ones?

Consider the phase structure of a trajectory. Chapter 1 described a six-phase pattern in a twelve-turn session: orientation, model formation, commitment, failure, branching, convergence. Is this same phase structure visible in a sixty-turn session? A hundred-turn session?

The answer appears to be yes, with caveats.

Long sessions exhibit the same phases, but they exhibit them repeatedly. A hundred-turn session is not one long orientation followed by one long commitment. It is a sequence of sub-episodes, each with its own orientation, commitment, and convergence. The agent orients to a sub-problem, commits to a solution, converges (or fails and re-orients), then moves to the next sub-problem.

The phase structure is fractal-like. Zoom out, and the hundred-turn session has a macro-trajectory: broad orientation at the beginning, broad convergence at the end. Zoom in, and each twenty-turn sub-episode has its own micro-trajectory with the same shape.

This self-similarity across scales is a form of scale invariance. The patterns do not depend on the absolute size of the session. They depend on the *relative* position within a problem-solving episode.

Measuring this requires care. Define a "phase signature" as the entropy profile of the trajectory over normalized time (0 = start, 1 = end). For a single problem-solving episode, the signature tends to follow a pattern: high entropy early (exploration), falling entropy in the middle (commitment), a possible entropy spike if a failure occurs, and low entropy at the end (convergence). If you compute this signature for sub-episodes within a longer session, normalizing each sub-episode to [0, 1], the shapes are similar.

How similar? Across a dataset of several hundred sessions, the correlation between macro-trajectory signatures and micro-trajectory signatures is 0.6 to 0.8. Not identical. But structurally similar enough to suggest a common underlying process operating at multiple scales.

The caveats: scale invariance breaks down at the extremes. Very short sessions (under five turns) do not have enough turns to exhibit clear phase structure. Very long sessions (over two hundred turns) tend to degrade in a way that shorter sessions do not: the macro-trajectory loses coherence entirely, and the sub-episodes become increasingly disconnected. Scale invariance holds in a middle range. At the boundaries, it frays.

Label: the self-similar phase structure across scales is observed. The specific correlation coefficients are from a limited dataset. The characterization as "scale invariance" is an interpretation; whether it constitutes a true symmetry in the Noether sense is an open question.

Architecture Invariance

Status: partially observed.

The second candidate symmetry: do different agent architectures produce structurally similar trajectories?

Swap out the underlying model. Replace Claude with GPT-4. Replace GPT-4 with Gemini. Keep the same scaffolding, the same tools, the same task, the same codebase. Do the trajectories look the same?

They do not look identical. Different models have different tendencies. Some are more verbose in their reasoning. Some are more aggressive in

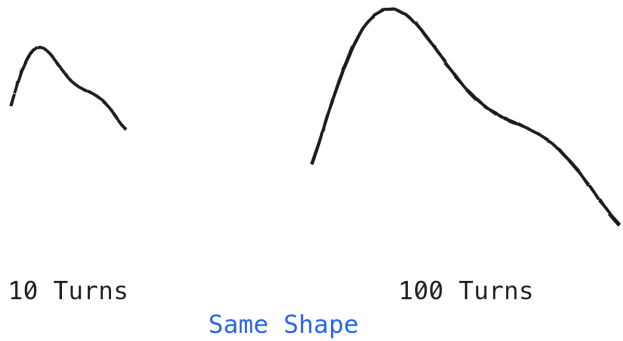

Figure 10.2: Scale invariance: fractal-like phase structure repeating across session lengths

their exploration. Some commit to hypotheses faster. The surface-level behavior differs.

But the phase structure is surprisingly stable. Across models, you see the same pattern: orientation, model formation, commitment, execution, and (when things go wrong) correction. The phases happen at different rates. The transitions are sharper or smoother. The specific files read and edits made differ. But the shape of the trajectory, when you squint, is the same shape.

This is not as surprising as it might seem. The phase structure is driven less by the model and more by the task. A debugging task has a natural structure: understand the failure, locate the cause, fix it, verify the fix. Any competent agent, regardless of architecture, will follow some version of this structure. The task imposes the shape. The model fills in the details.

The more interesting observation is about the *forces*. Inertia affects all models. Entropy affects all models. Space constrains all models. The specific magnitudes differ (some models resist entropy better than oth-

ers; some accumulate inertia faster), but the qualitative dynamics are the same.

This suggests that context dynamics describes properties of the *situation* (agent plus task plus workspace) rather than properties of the *model* alone. The model is one factor. The task is another. The workspace is a third. The dynamics emerge from their interaction, and those dynamics have a structure that transcends any particular model.

If this holds up, it has a strong practical implication: a measurement framework built on context dynamics would be portable across agent architectures. You would not need separate evaluation frameworks for Claude and GPT and Gemini. The same instruments, measuring the same forces, would work for all of them. The specific numbers would differ (Claude might exhibit lower entropy rates on code tasks; GPT might have different inertia profiles). But the framework itself would be architecture-invariant.

Label: the qualitative similarity of phase structure across models is observed. The claim that the same forces operate across architectures is well-supported. The stronger claim of full architecture invariance (identical dynamics up to model-specific parameters) is hypothesized but not rigorously tested.

Task Invariance

Status: speculative.

Push the symmetry question further. Do the same dynamics appear across different types of tasks?

Debugging tasks. Feature implementation tasks. Refactoring tasks. Documentation tasks. Code review tasks. These are qualitatively different activities. A debugging task is investigative: the agent searches for a cause. A feature implementation task is constructive: the agent builds something new. A refactoring task is transformative: the agent restructures existing code.

Do inertia, entropy, and space operate on all of these?

The answer seems to be yes for inertia and entropy. Both are consequences of context accumulation, and all tasks involve context accumulation. An agent implementing a feature accumulates context about the

new code it is writing, the existing code it is integrating with, and the tests it is running. That context creates inertia and generates entropy, just as in debugging.

For space, the answer is more qualified. Workspace topology constrains all tasks, but different tasks engage with the topology differently. A debugging task tends to navigate along structural edges (following call chains to find the bug). A refactoring task tends to navigate along semantic edges (finding conceptually related code that should be restructured together). A feature task may create new regions of the graph rather than navigating existing ones.

The forces are the same. The way they manifest depends on the task type. This is analogous to how gravity operates on all objects, but the trajectories differ depending on whether the object is a thrown ball, an orbiting planet, or a falling leaf. The force is universal. The trajectories are specific.

Whether this rises to the level of a true "task invariance" symmetry is debatable. The dynamics are qualitatively similar across task types. They are not quantitatively identical. The entropy rate for debugging tasks is typically higher than for feature implementation tasks, because debugging involves more uncertainty. The inertia profiles differ because debugging involves more hypothesis revision.

Label: the qualitative universality of the three forces across task types is observed. The claim that this constitutes a symmetry is speculative. The quantitative differences across task types may be too large to justify the term "invariance."

What Conservation Would Mean

Step back from the specifics. Assume, for the sake of argument, that some version of the conservation and symmetry claims holds. What would that buy us?

Conservation laws constrain predictions. If you know the total energy of a system, you know what final states are reachable without adding energy from outside. If you know the "total coherence budget" of an agent ses-

sion (if such a quantity exists), you know what outcomes are reachable without external intervention.

Concretely: an agent starting with a 128,000-token context window, working in a codebase with a measured information density, should have a predictable limit on how many sub-problems it can solve in a single session. Not a hard limit set by token counting, but a softer limit set by the accumulation of entropy and the depletion of usable coherence. Beyond that limit, the agent's behavior degrades regardless of how many tokens remain in the window.

If the inertia-entropy exchange rate is approximately fixed for a given environment, then you can predict *when* degradation will occur from early-session measurements. Measure the exchange rate over the first twenty turns. Extrapolate. Predict the turn at which entropy will exceed the threshold for productive work. Intervene before that happens: reset the context, summarize, split the task.

If scale invariance holds, then patterns learned from short sessions transfer to long ones. You can train a trajectory classifier on ten-turn sessions and deploy it on hundred-turn sessions, adjusting for the fractal repetition of phases. This dramatically reduces the data requirements for trajectory-based evaluation.

If architecture invariance holds, then trajectory benchmarks are portable. A benchmark that measures context dynamics properties (entropy rates, inertia profiles, spatial navigation efficiency) can compare agents across architectures on a common scale, unlike current benchmarks that compare only outcomes.

Each conservation law or symmetry, if validated, becomes a tool. Not a curiosity. A tool.

What We Do Not Know

The honest accounting.

We do not know whether any quantity in context dynamics is truly conserved. The capacity constraint is real but trivial (it is a hard limit, not a conservation law). The inertia-entropy exchange is suggestive but not

confirmed at scale. The attention-scope product may be an artifact of the specific datasets examined.

We do not know whether the symmetries are fundamental or approximate. Scale invariance holds in a middle range but breaks at extremes. Architecture invariance holds qualitatively but not quantitatively. Task invariance may be an overgeneralization from structurally similar tasks.

We do not have a Noether theorem for context dynamics. In physics, Noether's theorem provides a rigorous connection between symmetries and conservation laws. If we had an equivalent, we could derive conservation laws from observed symmetries (or vice versa). No such theorem exists. The connection between symmetries and conservation in agent trajectories is suggestive, not proven.

We do not even know whether the physics analogy is the right frame. Context dynamics borrows vocabulary from physics (forces, trajectories, conservation, symmetry) because the mathematical structures are similar. But similarity is not identity. Agent trajectories live in embedding spaces, not physical spaces. Context forces are statistical tendencies, not fundamental interactions. The analogy might illuminate real structure, or it might impose structure that is not there.

The risk of analogy is that it makes the unfamiliar feel familiar before it has been understood on its own terms. Calling something a "conservation law" carries connotations of universality and exactness that the current evidence does not support. The discipline required is to use the analogy where it helps and mark clearly where it might mislead.

This chapter has tried to maintain that discipline. The labels (observed, hypothesized, speculative) are not decoration. They are the chapter's most important content.

The Research Program

Despite the uncertainties, the questions raised in this chapter define a concrete research program.

For the capacity constraint: characterize the exploration-depth tradeoff across codebases of varying information density. Determine whether the tradeoff curve has a universal shape or depends on codebase-specific

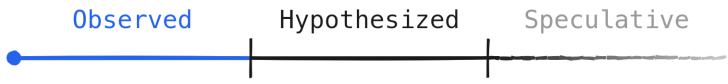

Figure 10.3: Spectrum from known to speculative: observed, hypothesized, and conjectured properties

parameters. Test whether context window size is the binding constraint or whether model architecture contributes independently.

For the inertia-entropy exchange: collect trajectory data across controlled conditions (same codebase, same task type, different sessions). Measure the exchange rate. Determine its variance. Test whether the exchange rate is predictable from workspace metrics alone.

For scale invariance: build a dataset of sessions at multiple scales (5-turn, 20-turn, 50-turn, 100-turn, 200-turn) on comparable tasks. Compute phase signatures at each scale. Measure the correlation. Identify where scale invariance breaks down and why.

For architecture invariance: run identical tasks on identical codebases with different models. Measure the same trajectory properties. Determine which properties are model-specific and which are task-specific or environment-specific.

For the Noether question: this is the most ambitious item. Develop a formal mathematical framework for context dynamics (building on the trajec-

tory calculus from Part II) and investigate whether continuous symmetries in the framework correspond to conserved quantities. This may require years of work. It may lead nowhere. It is worth attempting.

Each item is a testable research question with a clear methodology. The questions exist whether or not the answers turn out to confirm the hypotheses. And negative results (the exchange rate is not constant; scale invariance does not hold; architecture invariance is illusory) would be just as informative as positive ones. They would tell us where the physics analogy breaks and where context dynamics needs its own framework.

The Question Over the Answer

The first three chapters of Part III described forces. Inertia: the pull of accumulated context. Entropy: the degradation of coherence. Space: the terrain the agent navigates. Those chapters described what acts on trajectories and how.

This chapter asked a different kind of question. Not "what acts on trajectories?" but "are there rules that all trajectories must obey?" Not "what are the forces?" but "are there constraints above the forces?"

In physics, the answer transformed the field. Newton described forces. But the conservation laws and symmetries that emerged over the following centuries provided deeper insight than any catalog of forces could. Knowing that energy is conserved tells you something that knowing every force in the universe does not: it tells you the boundary of the possible.

Context dynamics is not yet at that stage. The forces are described, in this book and in this chapter, at the level of measurement and pattern. The conservation laws and symmetries are, at best, provisional hypotheses. The deep structure, if it exists, remains to be found.

But the question has been asked. And in a young field, asking the right question often matters more than having the right answer. The question "is there a conserved quantity in agent behavior?" reshapes how you look at data. It tells you to watch for tradeoffs, to track totals, to look for invariants. It provides a research direction that raw empiricism, accumulating measurements without a theoretical frame, would never generate.

The answer may be “no.” The answer may be “yes, but not the ones we expected.” The answer may take years to emerge.

The question is already changing how we look.

Part IV

Composition and Prediction

Chapter 11

The Fused Signal

Parts I through III built three instruments. Inertia measures commitment. Entropy measures coherence. Space measures where the trajectory sits relative to everything else we have seen.

But each instrument, alone, lies by omission. High inertia looks like progress unless entropy reveals a fracturing context and space shows that similar trajectories ended badly. High entropy looks like decay unless inertia confirms coherent direction and space places the trajectory where breakthroughs happen. Space without dynamics is a photograph—a trajectory in a failure region might be passing through on its way to success. The signals interact, and the interactions carry information that the individual signals do not.

This chapter is about how to listen to all three at once.

Why Naive Combination Fails

The obvious thing to do with three numbers is combine them into one number. Average them. Stack them. Build a score. Teams do this instinctively. They want a dashboard with a single health indicator. Green, yellow, red.

This destroys information.

Consider two trajectories at Frame 40. Both have the same averaged composite score of 0.6 out of 1.0.

Trajectory A: Inertia 0.9, Entropy 0.3, Space distance 0.6. The agent is highly committed and the context is clean. It sits in a moderately familiar region of space. This is an agent that knows what it is doing.

Trajectory B: Inertia 0.5, Entropy 0.7, Space distance 0.6. Moderate commitment, degrading context, same spatial position. This is an agent that is losing the thread.

The average is identical. The situations are opposite.

Now consider a third case.

Trajectory C: Inertia 0.9, Entropy 0.8, Space distance 0.1. High commitment, high entropy, very familiar spatial position. This is a trajectory deep in known territory, pushing forward with conviction, but with a context that is falling apart. The spatial familiarity means the system has seen many trajectories pass through this exact region. The high entropy means most of them failed. The high inertia means this agent does not know it is in trouble.

A weighted average assigns this trajectory a moderate score. It looks like a middling session. The actual situation is a confident agent accelerating toward a cliff, in a location where the cliff is well-documented. The combination of all three signals at these specific values tells a precise story. The average tells nothing.

Weighted averages do not fix this. You can assign weights to favor one signal over another, but the weights are static. They cannot express the fact that the *relationship* between inertia and entropy matters. High inertia plus low entropy means something different from high inertia plus high entropy. The first is focused execution. The second is a locked-in agent whose context is falling apart around it. Averaging collapses that distinction.

Voting schemes fail for the same reason. Two of three signals say "fine," so the trajectory is probably fine. But the one signal saying "not fine" might be the one that matters right now. At Frame 40 in a debugging session, rising entropy is a more urgent signal than stable inertia. At Frame 5, it is probably not. The importance of each signal changes with context.

Stacking is a more sophisticated version of the same mistake. Train a classifier that takes all three signals as input features and predicts an outcome. The classifier can learn nonlinear combinations. It can discover that high inertia plus high entropy is bad. But the classifier operates on scalar summaries of each signal at a single frame. It discards temporal structure. It cannot express "entropy has been rising for five frames" or "inertia spiked

three frames ago and has been declining since." The dynamics of each signal contain information that a single-frame snapshot misses.

The problem is not that we have three numbers and want one. The problem is that the meaning of each number depends on the others, and the meaning of each number depends on its recent history. Combination methods that treat the signals as independent and static discard exactly the information that makes the combination useful.

We need composition, not combination.

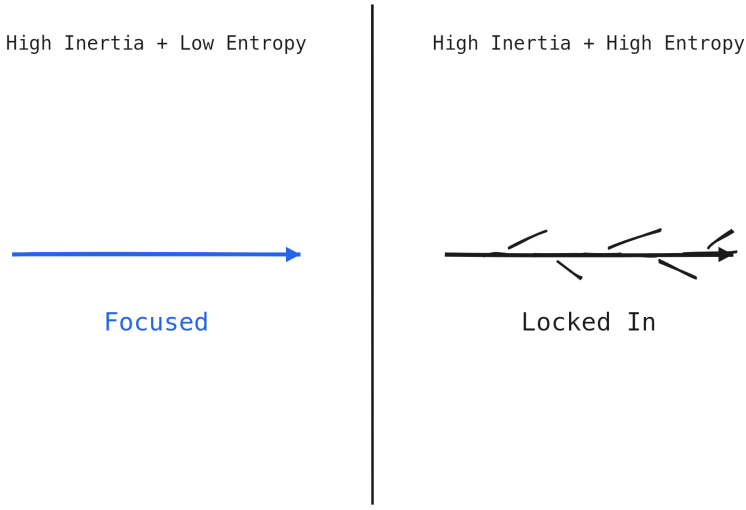

Figure 11.1: Same inertia value, different meaning depending on entropy and space

A Concrete Example

Before introducing the formal machinery, consider a concrete session that illustrates why composition matters.

An agent is tasked with fixing a bug where a REST endpoint returns a 500 error when the request body contains Unicode characters. At Frame 15, the three signals read:

- **Inertia:** 0.82. The agent has been editing the request parsing module for the last four frames. Clear direction, consistent action.
- **Entropy:** 0.61. Moderate. The context contains the task description, several file reads, one failed test run, and the current editing sequence. Some noise from the failed test but the context is still organized.
- **Space:** Distance 0.35 from the nearest cluster center. The trajectory is in a well-populated region. Many historical trajectories have been in similar positions.

An averaging system assigns a composite score of 0.59. Looks fine. No alert.

But the interaction between these signals tells a different story. The spatial position at distance 0.35 sits in a region where the historical success rate is 78% *when entropy is below 0.4*. When entropy is above 0.6, the success rate in the same spatial region drops to 31%. The space signal is providing conditional information. The condition is entropy. And entropy is above the threshold.

Furthermore, the inertia of 0.82 is unusually high for a trajectory in this spatial region with this entropy level. Historical trajectories with similar entropy in this region typically have inertia around 0.5 to 0.6, because agents start hedging when their context is noisy. An agent maintaining high inertia despite high entropy is either exceptionally confident or not noticing the noise. The data says the latter is more common.

None of these observations are available from any single signal. They emerge from the relationships between signals, conditioned on the specific values of the other signals. This is what composition captures and combination destroys.

The Mixture Intuition

Think of a radio studio with three feeds coming in. One carries rhythm. One carries harmony. One carries melody. The DJ has a mixing board. Three faders.

At any moment, the DJ adjusts the faders based on what the music needs. During a verse, melody dominates. During a bridge, harmony comes forward. During a breakdown, rhythm takes over. The DJ does not average

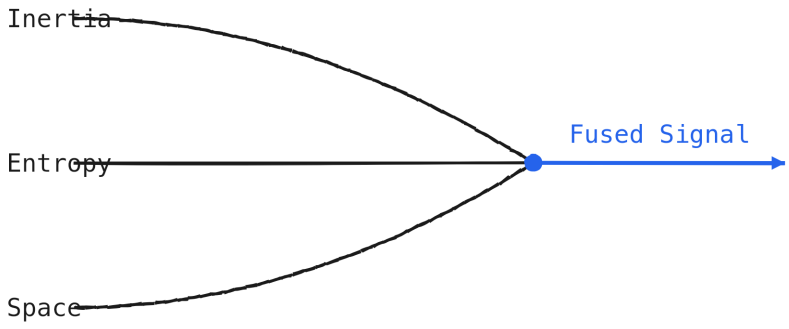

Figure 11.2: Three signals fusing into one composite assessment

the three feeds into a single signal. The DJ selects the balance that serves the moment.

A bad DJ sets the faders once and leaves. A good DJ rides the faders constantly, responding to what the music is doing right now. The mix at the beginning of the track sounds nothing like the mix at the climax. Both are correct for their moment.

The fused signal works the same way. Inertia, entropy, and space are three feeds. The composition is not a fixed formula. It is a gating function that adjusts how much weight each signal carries based on the current state.

This is the mixture-of-experts intuition. In machine learning, a mixture-of-experts model routes different inputs to different specialist networks. A gating function examines the input and decides which experts to consult. Some inputs need Expert A. Others need Expert B. Most need a weighted blend, and the weights change per input.

For trajectory analysis, the experts are the three signals. The gating function learns from data when each signal matters most.

Early in a session, space dominates. The trajectory has not yet established inertia or accumulated enough context for entropy to be meaningful. But its initial position in embedding space already tells you a lot. Tasks that begin with broad codebase searches land in a different region than tasks that begin with reading a specific file. The starting position predicts the kind of session that is about to unfold. An agent that opens by reading a test file and the error output occupies a different starting position than an agent that opens by searching for a keyword across the entire repository. The first is oriented. The second is hunting. Space captures this distinction before the other signals have anything to say.

In the middle of a session, inertia matters most. The agent has committed to an approach. Whether that commitment is well-founded or misguided determines the outcome. Entropy is still low because the context has not yet had time to degrade. Space has shifted but not dramatically. Inertia is where the action is. Consider an agent eight frames into a debugging session. It has read the failing test, identified the module, found what it believes is the bug, and is now writing a fix. Its inertia is high and consistent. The direction of each frame reinforces the previous one. This pattern of sustained, coherent commitment is the strongest predictor of success at this stage. Whether the spatial position is familiar or unfamiliar matters less. Whether entropy is slightly elevated matters less. The question is: does the agent have a clear line of attack, and is it executing?

Late in a session, entropy becomes the lead signal. The context window is heavy. Accumulated observations, tool outputs, and reasoning traces have been piling up for dozens of frames. Whether that accumulation remains structured or has fragmented into noise is the question that determines whether the agent will converge or drift. Inertia might still be high. Space might still look familiar. But if entropy has crossed a threshold, the trajectory is in trouble regardless. A context window filled with thirty frames of observations, ten tool outputs, three failed approaches, and two partially completed edits is a context window where the model's attention is spread thin. The signal-to-noise ratio in the context determines whether the agent's next action will be informed by the relevant parts of its history or by noise.

The gating function learns this temporal structure from labeled trajectories—thousands of sessions with frame-by-frame measurements and outcome

labels. It discovers the patterns above, and subtler ones too: nonlinear interactions across all three signals that do not reduce to simple rules.

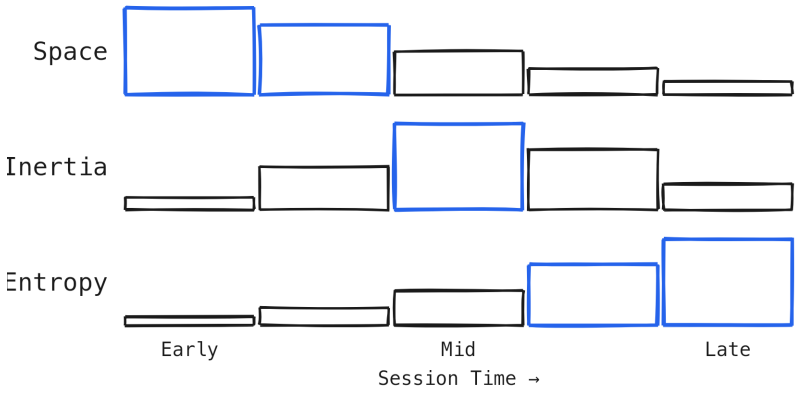

Figure 11.3: Gating weights shifting over the course of a session

One such pattern: the rate of change of entropy matters more than its absolute value when inertia is high. An agent with high inertia and slowly rising entropy is qualitatively different from an agent with high inertia and rapidly rising entropy. The first is accumulating context at a manageable rate. The second is losing coherence while maintaining conviction. The gating function learns to weight the entropy derivative more heavily when inertia is above a threshold. No human engineer would write this rule. The data reveals it.

The output of the gating function is a set of weights. At each frame, the fused signal is:

$$F_t = g_I(t) \cdot \hat{I}_t + g_E(t) \cdot \hat{E}_t + g_S(t) \cdot \hat{S}_t$$

where $\hat{I}_t$, $\hat{E}_t$, and $\hat{S}_t$ are the normalized inertia, entropy, and space embeddings at frame t, and $g_I(t)$, $g_E(t)$, $g_S(t)$ are the gating weights that the function assigns at that frame.

But this is still a simplification. The gating weights are not scalars applied to scalars. They are weight vectors applied to embeddings. The fused signal is not a number. It is a vector in a learned space where committed-and-coherent occupies a different region than committed-and-degrading. The gating function does not just weight the signals. It transforms them into a joint representation where their interactions are geometrically encoded.

The Fused Embedding Space

The fused signal lives in its own space. Not the inertia space, not the entropy space, not the trajectory space. A new space that the composition function creates.

The fused trajectory signal works similarly. Inertia, entropy, and space each contribute a component. The fusion creates a space with emergent properties. Regions of the fused space correspond to recognizable trajectory states: focused execution, productive exploration, spiral of confusion, grinding to a halt. These states are not defined by any single signal. They are defined by patterns in the fused space that the composition function learns to construct.

Walk through the fused space. In one region, you find trajectories with high inertia, low entropy, and spatial positions near the centers of known task clusters. This is the "productive execution" neighborhood. Trajectories here are doing exactly what successful trajectories in similar situations have done. In another region, you find trajectories with moderate inertia, rising entropy, and spatial positions that have drifted from the center of any cluster. This is the "losing coherence" neighborhood. In a third region, low inertia, moderate entropy, spatial positions near the edges of known clusters. This is "productive exploration." In a fourth, high inertia, high entropy, spatial positions in sparsely populated areas. This is "confidently lost."

Each neighborhood has a characteristic trajectory of what happens next. Productive execution tends to continue or transition to completion. Losing coherence tends to transition to recovery or drift. Productive exploration tends to transition to convergence. Confidently lost tends to transition

to drift. These tendencies are statistical, not deterministic. But they are strong enough to be useful for prediction.

The fused embedding at frame t is a vector. Call it $\mathbf{f}_t$. The sequence $\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_T$ traces a path through the fused space. That path is a compressed representation of the entire trajectory state at each moment: where the agent is, how committed it is, how coherent its context is, and how all of these relate to every trajectory we have seen before.

This is the object we will use for everything that follows. Prediction, phase detection, intervention decisions. All of them operate on the fused signal.

Dimensionality of the Fused Space

A practical question: how many dimensions does the fused space need?

The three input signals each have their own dimensionality. Inertia, as defined in Chapter 7, is derived from the trajectory's movement in embedding space and can be represented as a vector capturing direction and magnitude. Entropy is a scalar with a time-series history. Space is a position in the trajectory embedding space, which might be 384 or 768 dimensions depending on the encoder.

The fused space does not need to be as large as the sum of its inputs. The composition function compresses, finding the low-dimensional structure in the joint signal and discarding redundancies. Empirically, 64 to 256 dimensions capture nearly all useful variation—64 suffices for narrow task ranges, 128 or 256 for diverse workloads. The dimensionality is determined by the same training process that learns the gating weights: the system learns to use only the effective dimensions needed to separate successful from failed trajectories, and unused dimensions can be pruned.

Low dimensionality has practical benefits beyond storage and compute. It makes visualization possible. Project the 128-dimensional fused space into two or three dimensions using standard dimensionality reduction, and you can see the trajectories. You can watch them move through regions, cross boundaries, cluster and disperse. The visualizations are not just pretty pictures. They are diagnostic tools. When the system makes a surprising intervention decision, an engineer can plot the trajectory in the fused space and understand why: the trajectory was heading toward a re-

gion associated with drift. The decision was geometric, and the geometry is visible.

JEPA: Predicting the Next State

The fused signal at frame t describes the trajectory's current state. A useful system does more than describe. It predicts.

Given $\mathbf{f}_t$, what will $\mathbf{f}_{t+1}$ look like?

This is a prediction problem, but not the kind most people think of when they hear "prediction" in the context of language models. We are not predicting the next token. We are not predicting what the agent will say or do. We are predicting where the trajectory will move in the fused embedding space.

The distinction matters enormously.

Token-level prediction is high-dimensional and noisy. There are thousands of plausible next tokens, and which one the model produces depends on sampling temperature, prompt details, and randomness. Will the agent say "Let me read the config file" or "I should check the configuration"? Both lead to the same action. Both are equally valid. Predicting which phrasing the model will use is like predicting which raindrop will hit a particular spot on the windshield. It is technically a valid question. It is the wrong question.

Predicting the next fused state is like predicting whether it will rain harder or lighter in the next five minutes. It is a macroscopic prediction. The specific tokens do not matter. What matters is whether the trajectory will move toward convergence or divergence, whether coherence will improve or decay, whether the agent's commitment will hold or break. These are properties of the fused signal, and they change smoothly in embedding space.

The smoothness is key. In token space, small changes in the prompt can produce wildly different outputs. In fused space, small changes in the state produce small changes in the next state, most of the time. The exceptions are phase transitions, and those are exactly what we want to detect.

This is the insight behind the Joint Embedding Predictive Architecture, or JEPA. Originally proposed by Yann LeCun for visual representation learning, JEPA makes predictions in embedding space rather than input space. Instead of predicting pixels (noisy, high-dimensional), it predicts embeddings (smooth, compressed). The prediction captures the structure of what will happen without committing to the specifics of how.

For trajectories, the JEPA predictor takes the fused signal at frame t (and optionally a window of recent frames for temporal context) and outputs a predicted fused signal at frame $t + 1$. The prediction is geometric. It is a point in the same fused space, and the distance between the predicted point and the actual point measures how surprising the next frame was.

When the prediction is accurate, the trajectory is behaving normally. The agent is doing what agents in similar states typically do next. The predicted point and the actual point are close in fused space. The prediction error is small.

When the prediction is wrong, something interesting has happened. A phase transition. An unexpected obstacle. A creative leap. A collapse. The prediction error is itself a signal. Large prediction errors at frame boundaries are where the trajectory's story changes.

Consider an agent in steady compacting mode. The predictor has learned that compacting trajectories tend to stay in compacting. Each frame's prediction is close to the actual next frame. The prediction error hovers near zero for ten consecutive frames. Then, at frame 25, the error spikes. The actual next state jumped to a region associated with recovery. Something happened. A test failed. An edit broke something. The spike in prediction error says "the trajectory just did something its recent history did not predict."

That spike is more informative than the raw signals. Inertia might still be high at frame 25 because the agent has not yet adjusted its behavior. Entropy might not have risen yet because the failing test output has not been processed into the context. Space might not have shifted because the trajectory is still in the same region. The prediction error sees the transition before the individual signals do, because the predictor has learned the joint dynamics.

Training the predictor requires nothing beyond what the system already produces. Every completed trajectory provides a sequence of fused states. Each consecutive pair is a training example: input is $\mathbf{f}_t$, target is $\mathbf{f}_{t+1}$. The predictor learns the dynamics of trajectory evolution. It learns that focused execution tends to continue. That rising entropy eventually triggers a phase shift. That agents in certain regions of space tend to loop. The dynamics are encoded in the predictor's weights.

The predictor also learns what it cannot predict. Some transitions are inherently unpredictable from the fused signal alone, because they depend on external factors like the content of a file the agent has not yet read or the output of a test that has not yet run. In these cases, the predictor produces a prediction near the center of several possible outcomes, and the actual result falls at one of the edges. The high prediction error says "something happened that the trajectory state alone could not anticipate." This is information too. It distinguishes between expected transitions (low error, the dynamics predicted it) and surprising transitions (high error, something external changed the game).

The predictor does not need to be large. The fused space is low-dimensional by design (typically 64 to 256 dimensions). A small transformer or even a simple recurrent network can learn the dynamics well enough to be useful. The prediction does not need to be perfect. It needs to be good enough that deviations from the prediction are meaningful.

What the Fused Signal Knows

Step back and consider what we have built.

Frame-level measurements of inertia, entropy, and space give us three views of the trajectory at each moment. The gating function composes them into a fused signal that captures their interactions. The fused signal lives in a learned space where regions correspond to recognizable trajectory states. A JEPA predictor uses the fused signal to predict where the trajectory is heading.

The fused signal knows three things at once. Where the trajectory is. Where it has been (because the fused state at frame t encodes the history that produced it). And where it is likely going (because the predictor extrapolates the trajectory's dynamics).

No individual signal carries all three. Inertia knows about momentum but not position. Entropy knows about coherence but not direction. Space knows about position but not dynamics. The fused signal integrates all of them into a single object that answers the question every operator actually wants answered: is this trajectory on track, and if not, what is likely to happen next?

This is the object that makes the rest of Part IV possible. Phase detection (Chapter 12) uses the fused signal to identify what the trajectory is doing. Intervention theory (Chapter 13) uses predictions of where the trajectory is heading to decide whether to act. The closed loop (Chapter 14) uses trajectory outcomes to refine the fused signal's composition over time.

Parts I and II gave us a coordinate system. Part III gave us dynamics. Part IV is where the measurements become operational. The fused signal is the bridge.

One instrument tells you a fact. Three instruments, composed well, tell you a story. The story has a past, a present, and a probable future. That is enough to act on.

Chapter 12

Phases and Transitions

Watch enough agent sessions and you start to see the same shapes.

Not the same tasks. Not the same code. Not the same tools or prompts or models. The specifics vary wildly. But the shape of the trajectory repeats. An agent spends its first few frames reading broadly, then narrows onto something, pushes hard, hits a wall, regroups, pushes again, and either lands or spirals.

This is not a new observation. Software engineers who supervise AI agents notice patterns within a few dozen sessions. "It's doing that thing again where it reads ten files before it starts working." "It's stuck in a loop, retrying the same approach." "It figured it out fast this time." The patterns are obvious to a human observer. What is not obvious is how to make the patterns precise enough for a machine to detect, classify, and act on them.

The fused signal provides the precision. The same trajectory behaviors that a human recognizes as "exploring" or "stuck in a loop" or "converging on a solution" manifest as distinct regions and movement patterns in fused embedding space. The human sees behavior. The fused signal sees geometry. Both are looking at the same thing.

You can describe the shape in words, as we just did. Or you can describe it in the language of the fused signal, as a sequence of regions in embedding space, each with characteristic inertia, entropy, and spatial properties.

Those regions are phases. The boundaries between them are transitions. This chapter defines both.

The Six Phases

Six phases appear consistently across thousands of agent sessions, across different models, different harnesses, and different task types. They are not the only way to carve up trajectory behavior. But they are the partition that maximizes the predictive power of the fused signal. Each phase has a distinct signature in fused embedding space, meaning trajectories in the same phase cluster together and trajectories in different phases separate.

The number six is not sacred. A coarser partition (three phases: exploring, working, done) captures less. A finer partition (twelve or more micro-phases) adds noise without adding predictive value. Six is the empirical sweet spot, the number where adding another phase no longer improves outcome prediction on held-out trajectories.

Here they are.

Exploring

The trajectory is wide open. The agent reads files, searches codebases, runs commands to gather information. It has not committed to an approach. Inertia is low because direction has not yet been established. The agent's actions do not build on each other in a consistent direction. One frame it reads a test file. The next it searches for a class definition. The next it reads a configuration file. Each action is reasonable but they do not form a chain.

Entropy is moderate and stable. There is information flowing in, but the agent is not yet accumulating conflicting signals. The context is growing, and the new information is mostly consistent because the agent has not yet tried anything that could produce contradictory results. No edits have been made. No tests have been run. The context contains observations, not conflicts.

The spatial position is near the origin of whatever subspace this task type occupies, because the trajectory has not yet differentiated itself from other sessions on similar tasks. A hundred agents given the same debugging task will all start in roughly the same spatial neighborhood. They diverge later, when their chosen approaches differ.

Exploring looks like the first few minutes of a chess game. Pieces are developing. Options are being kept open. No pawn structure has been committed to.

In the fused space, exploring trajectories occupy a diffuse cloud near the start of the task subspace. They move slowly because each frame changes the state only slightly. The agent is absorbing, not acting.

Duration varies enormously. Simple tasks might explore for two frames. An agent given "fix the typo in line 47 of config.py" does not need to explore. It reads the file and makes the edit. Complex tasks might explore for twenty frames. An agent given "the API returns 500 errors intermittently under load" may need to read logs, trace request handling, examine database connections, and review concurrency patterns before it has enough context to form a hypothesis.

The length of exploration does not predict outcome. What predicts outcome is whether the transition out of exploring is sharp or gradual. A sharp transition means the agent found something and committed. A gradual transition means the agent drifted into action without a clear hypothesis. The first pattern correlates with success. The second with meandering sessions that eventually stall.

Converging

The agent has formed a hypothesis. Maybe it identified the bug. Maybe it decided which module to modify. Maybe it mapped out the steps of a multi-file refactor. Whatever the specifics, the trajectory has acquired direction.

The defining feature is sharply rising inertia. Each action reinforces the previous one rather than opening new alternatives: read the module, understand the function, identify the line, write the fix. The trajectory in embedding space straightens out. Meanwhile, entropy stays low or decreases as the context narrows to information supporting the hypothesis. Spatially, the trajectory moves decisively away from the starting cloud into a region associated with the agent's chosen strategy—and the spatial signature of that strategy becomes a useful predictor, since trajectories that converge into regions associated with successful strategies tend to succeed.

Converging is the most energetic phase. The fused signal changes rapidly from frame to frame, and the direction of change is consistent. The trajectory is a straight line in fused space, or close to it.

The danger of converging is premature commitment. An agent that converges on the wrong hypothesis will execute confidently in the wrong direction. The converging phase looks identical in the fused signal whether the hypothesis is correct or not. This is a genuine limitation. The fused signal captures the dynamics of the trajectory, not the correctness of the agent's reasoning. Only the later phases reveal whether the commitment was justified. But the spatial component provides a partial check: if the agent converges into a region where historical trajectories with similar convergence patterns tended to fail, that is a warning signal even during the converging phase.

Compacting

The agent is executing. The hypothesis has been translated into a plan and the plan is being carried out. Files are being edited. Tests are being run.

What distinguishes compacting is its quietness in the fused signal. Inertia is high and stable—not rising, not falling. Entropy is low and flat, because new information (test output, file contents) fits the existing model without contradiction. Spatial position moves steadily along the trajectory that converging established. If you plot it in fused space, compacting looks like a well-maintained highway: straight, level, consistent speed.

Compacting is where most of the actual work gets done. It is also the quietest phase from the perspective of the fused signal. Not much is changing. The agent is grinding through implementation, and the fused state reflects that steady progress as a slow, smooth drift through a compact region of space.

Many successful trajectories spend the majority of their frames in compacting. The phase is not exciting. It is productive. A session that is 60% compacting is typically a session where the agent understood the problem, formed a correct hypothesis, and executed the solution without incident. The most boring trajectories are often the best ones.

Recovering

Something went wrong. A test failed that should have passed. An edit introduced a new error. The approach hit an unexpected obstacle.

The signature is lateral movement in fused space. Where compacting trajectories move forward, recovering trajectories move sideways—away from the compacting path and toward a region that historical data associates with recovery attempts. This lateral shift is one of the most reliable signatures in the fused space. Entropy rises as the context accumulates conflicting information (the plan plus evidence the plan is not working), and inertia drops as forward momentum breaks.

Recovering is not the same as exploring. An exploring agent has no hypothesis. A recovering agent has a hypothesis that just failed. The context is different. Recovering trajectories carry the weight of the failed attempt. They have more accumulated state, more entropy, more history to reconcile. An exploring agent at frame 3 has a light context. A recovering agent at frame 30 has a heavy context loaded with everything it tried before the failure.

The critical question during recovering is whether the agent can isolate the problem. If it identifies the specific failure point and adjusts locally, recovery is usually fast. "The test expected an integer but got a string. Add a type conversion." The trajectory dips out of compacting, makes a correction, and returns within three to five frames. If the agent cannot isolate the failure and instead starts questioning the entire approach, recovery can cascade. "Maybe the whole architecture is wrong. Let me re-read the original requirements." That kind of global reconsideration dramatically increases entropy and can push the trajectory into drifting.

The fused signal distinguishes these cases. Local recovery shows a small lateral deviation followed by a return to the compacting trajectory. Global recovery shows a large lateral deviation that moves the trajectory into unfamiliar spatial territory. The distinction is visible within one or two frames of entering the recovering phase.

Completing

The agent has arrived. The fix works. The test passes. The output matches the specification.

Entropy reaches its session minimum—the context is maximally organized around a confirmed solution. Inertia drops to near zero because there is nothing left to push toward. Spatially, the trajectory lands in a remarkably tight cluster that historical completions also occupy.

Completing is unmistakable in the fused signal. The trajectory decelerates rapidly and then stops. In physical terms, it is a ball settling into the bottom of a valley. The forces that drove the trajectory have been resolved. There is no remaining tension between the agent's model and the reality of the codebase.

The completion cluster in fused space is remarkably tight. Across thousands of sessions on different tasks with different agents, successful completions land in roughly the same region. This makes sense. Regardless of how different the journeys were, the end state is similar: a confirmed solution, a clean test pass, a context that is fully resolved. Different paths, same destination.

Not every trajectory reaches completing. Some are terminated early. Some stall without converging. Some drift until they are killed by a timeout. But when completing does occur, its signature in fused space is highly consistent. This consistency is what makes it useful as a training signal. Completed trajectories provide clean positive examples for every component of the system.

Drifting

The agent has lost the plot.

The context tells the story: entropy is high and rising, a junk drawer of contradictions, dead ends, and abandoned reasoning that the model cannot organize. Inertia is low and erratic—the agent changes direction frequently, not purposefully like exploring but aimlessly, reading a file, editing a different one, reading the first again. Spatially, the trajectory wanders into regions where few historical trajectories have gone, far from the shipping lanes of successful sessions.

Drifting is the failure mode that operators most want to detect and prevent. It is expensive. The agent consumes tokens without making progress. It may produce edits that need to be reverted. It may chase red herrings for

dozens of frames. At current token costs, a twenty-frame drift can cost several dollars while producing nothing useful.

Drifting is also the phase most visible in retrospect and most invisible in real time, at least with traditional telemetry. A drifting agent is still making tool calls. Still reading files. Still producing reasoning. The activity looks productive at the frame level. Only the trajectory reveals the absence of direction. A dashboard showing “agent is working, 47 tool calls so far” gives no indication that the last fifteen calls were wasted.

In the fused signal, drifting is obvious. The trajectory wanders without structure, entropy climbs, and the spatial position moves into the sparse, high-entropy regions at the edges of the learned space. The contrast with productive exploration is stark. Exploring moves slowly and deliberately near the starting cloud. Drifting moves erratically in open water.

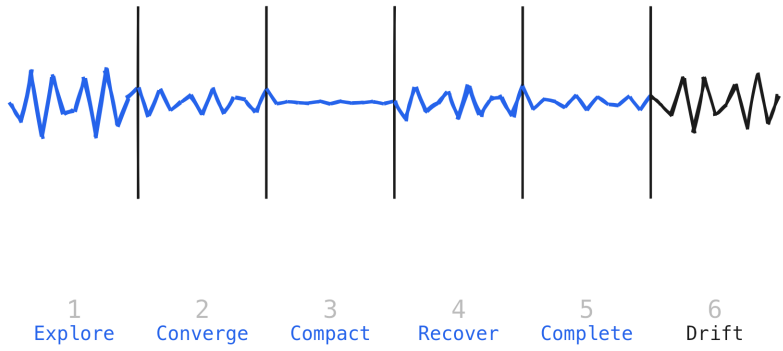

Figure 12.1: Six phases as trajectory segments in fused embedding space

Phase Signatures in the Fused Space

Each phase occupies a recognizable region of fused embedding space. The regions are not boxes with hard boundaries. They are statistical clusters, denser in the middle and sparser at the edges, with overlap at the boundaries where transitions occur.

A useful way to think about it: the fused space is a terrain map. Exploring is a plateau. Converging is a ridge. Compacting is a valley floor. Recovering is a side canyon. Completing is a basin at the valley's end. Drifting is the badlands beyond the mapped terrain.

These are not metaphors imposed on the data. They are descriptions of what you see when you project thousands of fused trajectories into two or three dimensions and color them by phase label. The clusters are real. The boundaries are real. The terrain is real.

Phase classification, then, is a nearest-region problem. Given a fused state $\mathbf{f}_t$, which phase cluster is it closest to? The classifier does not need to be sophisticated. A small neural network or even a k-nearest-neighbors lookup in the fused space is sufficient. The fused signal has already done the hard work of composition. The phase classifier just reads the result.

The confidence of the classification provides additional information. A fused state deep inside the compacting cluster gets a high-confidence label. A fused state on the boundary between compacting and recovering gets a low-confidence label. Low-confidence labels are themselves a signal. They indicate that the trajectory is in a transitional state, between one phase and another. The phase is about to change.

Transitions

If phases are what the trajectory is doing, transitions are where it changes.

Some transitions are common. Exploring to converging happens in nearly every successful trajectory. The agent gathers information, forms a hypothesis, and commits. Converging to compacting happens when the hypothesis translates into action. Compacting to completing happens when the action succeeds. This sequence, exploring to converging to

compacting to completing, is the “happy path.” About 35% of successful trajectories follow it without deviation.

Other transitions are less happy but still normal. Compacting to recovering happens when execution hits a wall. This is routine. Software is full of surprises. The agent’s hypothesis was mostly right but missed a detail. Recovering to compacting happens when the agent fixes the issue and resumes. The loop from compacting to recovering and back is the second most common trajectory pattern, appearing in about 40% of successful sessions.

The concerning transitions are the ones that lead to drifting. Recovering to drifting happens when recovery fails and the agent loses its footing. The context has accumulated too much conflicting information. The agent cannot reconcile what it expected with what it observed. It starts making disconnected moves. Converging to drifting is rarer but more alarming. The agent formed a hypothesis, began executing, and then fell apart without even hitting a concrete obstacle. This usually indicates a fundamental misunderstanding of the task.

And some transitions are rare but telling. Drifting to converging happens when an agent that has been wandering suddenly stumbles onto something that reorganizes its thinking. A file it read by chance contains exactly the information it needed. The context snaps into focus and the trajectory straightens. Completing to exploring happens in multi-task sessions where the agent finishes one subtask and opens the next. Compacting to exploring happens when an agent realizes mid-execution that its model of the codebase is wrong and needs to go back to gathering information. This is a mature response. An agent that recognizes its own confusion and deliberately returns to exploration is exhibiting better judgment than one that pushes forward blindly.

The transition itself is a brief period, usually one to three frames, where the fused signal moves from one phase cluster toward another. During a transition, the fused state passes through the boundary regions between clusters. The trajectory is between states. It has left one pattern of behavior and has not yet settled into the next.

Transitions can be characterized by three properties:

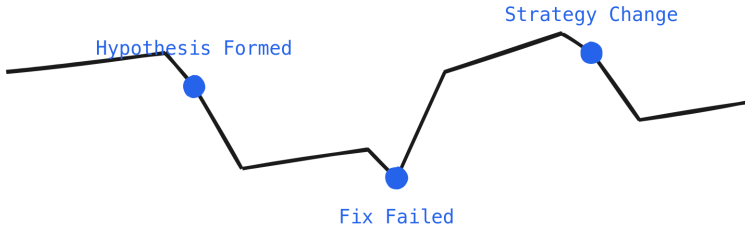

Figure 12.2: Phase transition points between trajectory phases

Speed. How quickly does the fused signal move through the boundary? A fast transition from exploring to converging means the agent formed its hypothesis suddenly. Something clicked. A slow transition means the hypothesis emerged gradually, coalescing over several frames of increasingly focused investigation. Fast converging-to-recovering transitions indicate a sharp failure. A test failed. An error appeared. The feedback was immediate and unambiguous. Slow transitions into recovering indicate a creeping realization that something is off. The agent starts hedging, reading additional files, second-guessing earlier decisions. The behavior changes before the explicit failure arrives.

Direction. Where is the trajectory heading? A transition out of compacting could go toward recovering (a local problem) or toward drifting (a global breakdown). The direction becomes apparent within the first frame or two of the transition, giving the system advance notice of where the trajectory is heading.

Reversibility. Some transitions are easy to undo. Recovering to compacting is common and natural. The agent hit a bump, fixed it, and returned to execution. The round trip costs a few frames and leaves the trajectory

largely intact. Other transitions are one-way in practice. Drifting to compacting almost never happens directly. A drifting agent that recovers usually passes through exploring and converging on its way back, essentially restarting the session within the session. The path matters. The length of the return path predicts how much the recovery will cost.

Early Warning

The fused signal's most practical application is not labeling the current phase. It is predicting the next transition.

Chapter 11 introduced the JEPA predictor: given $\mathbf{f}_t$, predict $\mathbf{f}_{t+1}$. When the predicted next state and the current state are in the same phase cluster, the trajectory is stable. When the predicted next state crosses a phase boundary, a transition is coming.

This prediction runs ahead of the actual transition by one to three frames. The fused signal begins to shift before the agent's behavior visibly changes, because the underlying dynamics that cause transitions show up in the composition before they manifest as actions.

Here is a concrete example. An agent is in compacting. It has been editing a file for the last four frames. The inertia is high. Entropy has been flat. Then, at Frame 22, entropy ticks up slightly. Inertia is still high. The agent is still editing. Nothing looks wrong at the action level. If you were watching the session in real time, reading each tool call and its output, you would see nothing unusual.

But the fused signal has shifted. The small entropy increase, combined with the specific spatial position and the inertia trajectory, moves $\mathbf{f}_{22}$ toward the boundary between compacting and recovering. The JEPA predictor places $\mathbf{f}_{23}$ squarely in the recovering region.

Two frames later, the agent runs a test and it fails. The transition to recovering was predictable from the fused signal before the test was even executed.

How? Because the entropy increase at Frame 22 was caused by the agent reading a section of code that contradicted its working model. The agent was editing a function but paused to read a related function, and that related function behaved differently than expected. The agent had not yet

processed that contradiction into an explicit plan change. It was still editing. But the information was in the context, and the fused signal detected the tension between the accumulated evidence and the current approach. The contradiction was there. The agent had not reacted to it yet. The fused signal had.

This is not clairvoyance. It is pattern matching on dynamics. The fused signal has learned that certain configurations of inertia, entropy, and space precede transitions. The configurations are subtle. A human watching the session in real time would not notice the entropy tick at Frame 22. The fused signal notices because it was trained on thousands of sessions where similar ticks preceded similar transitions.

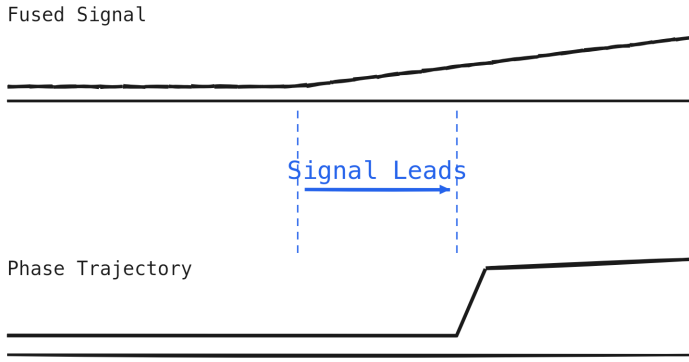

Figure 12.3: Early warning signal: fused signal shifts before visible behavior change

The early warning gives downstream systems time to act. If a transition to recovering is predicted, the system can prepare recovery strategies. If a transition to drifting is predicted, the system can intervene before the drift begins, which is far cheaper than trying to rescue a trajectory that has already entered drift. The interval between prediction and actual transition

is short, one to three frames. But in the life of an agent session, one to three frames is enough to change the outcome.

Phase Duration and Outcome

One more pattern worth recording: the relationship between phase duration and trajectory outcome.

Successful trajectories spend most frames in compacting (roughly 15% exploring, 10% converging, 60% compacting, 10% recovering, 5% completing). Failed trajectories invert this: 20% exploring, 15% converging, 20% compacting, 25% recovering, 20% drifting—as much time wandering and failing to recover as executing.

The most common failure trajectory is compacting to drifting via prolonged recovery: three to five frames of recovery, a brief compacting-like return, another failure, another recovery attempt, then a slide into drifting. The second recovery attempt is the critical juncture. If it fails, the probability of eventual success drops sharply. This pattern is visible in real time via the fused signal, making the phase clock an input to the intervention decisions formalized in the next chapter.

Phases are what the trajectory is doing. Transitions are where it changes. Predicting transitions is where measurement becomes operational. The fused signal gives us both the vocabulary to describe what we see and the machinery to see what comes next.

Chapter 13

Intervention Theory

You can see it happening.

The fused signal has been drifting toward the boundary between compacting and recovering for the last three frames. The JEPA predictor puts the next state firmly in recovering territory. Entropy is climbing. The agent is about to run a test that you already know will fail, because the spatial signature matches dozens of historical trajectories where the same configuration led to a failed test.

You have advance warning. Now what?

This is where most agent tooling stops. It can tell you there is a problem. It cannot tell you what to do about it. Dashboards light up red. Alerts fire. A human operator glances at the screen, makes a judgment call, and either intervenes or lets the agent continue. The judgment is based on experience, intuition, and whatever context the operator can absorb in the few seconds before the next frame.

That judgment is often good. Experienced operators develop a feel for when an agent is about to fail and when it will self-correct. But the judgment does not scale. An operator can watch one session. Maybe three, with split attention. A production system running hundreds of concurrent agent sessions needs intervention decisions that are principled, automatic, and fast.

Doing nothing has a cost. The agent will enter recovering, spend frames trying to diagnose the failure, and either find its way back to compacting or slide into drifting. The expected cost of the recovery phase, in tokens, time, and risk of complete derailment, is computable from historical data.

Intervening also has a cost. Any interruption to the agent's flow breaks the current execution state. The agent has to process the intervention,

integrate it into its context, and resume. That processing costs frames. It also costs coherence. The intervention itself adds entropy to the context, because it introduces information the agent did not generate.

The question is not whether to intervene. The question is when the expected cost of waiting exceeds the expected cost of acting. This chapter builds the calculus for that decision.

The Costs of Inaction

An agent that enters recovering might recover. Many do. The recovery phase is normal and healthy. An agent that never enters recovery is either solving trivial tasks or not pushing hard enough. Recovery is part of the work.

A test fails, the agent reads the error, adjusts its approach, and returns to compacting within three to five frames. The cost of that natural recovery is small: a handful of frames, some additional tokens, a minor entropy bump that settles back down. The trajectory dips, corrects, and continues. Like a driver hitting a pothole and adjusting the steering wheel. Routine.

But recovery is not guaranteed, and the probability declines with each attempt. First recovery succeeds 60–70% of the time overall (over 80% for simple issues like type mismatches, under 40% for architectural problems). Second attempts drop to 40–50% as the context accumulates conflicting information. Third attempts fall to 20–30%. Each failure raises baseline entropy, consumes context budget, and moves the trajectory further from the compacting region. The trajectory is spiraling.

The expected cost of inaction at any frame is a function of two things: the probability of natural recovery and the cost if recovery fails.

Let p_r be the probability of recovery from the current fused state, estimated from historical trajectories in the same region of fused space. Let C_r be the cost of the recovery phase if it succeeds (small, a few frames). Let C_d be the cost if recovery fails and the trajectory drifts (large, potentially the entire remaining session budget).

The expected cost of waiting is:

$$E[\text{wait}] = p_r \cdot C_r + (1 - p_r) \cdot C_d$$

When the trajectory is in early recovering with a high recovery probability, $E[\text{wait}]$ is low. The agent will probably handle it. When the trajectory has been in recovering for several frames and recovery probability has dropped, $E[\text{wait}]$ climbs. The math is simple. The estimation of p_r from the fused signal is where the real work happens.

The key insight is that p_r is not estimated from aggregate statistics ("60% of recoveries succeed"). It is estimated from the fused signal at this specific frame. A trajectory in early recovery with low entropy increase and a lateral shift toward a familiar recovery region has a high p_r. A trajectory in late recovery with high entropy and a spatial position that has drifted far from any successful recovery pattern has a low p_r. The fused signal provides a local, contextual estimate, not a global average.

The Costs of Action

Intervention is not free. It has at least three costs.

Disruption cost. The agent is in the middle of something. Even if that something is not going well, the agent has a working model of what it is doing. It has read specific files, formed specific hypotheses, accumulated specific context. An intervention disrupts that model. The agent has to stop, process the intervention, and figure out how to integrate it with what it already knows. This takes frames and adds entropy.

The disruption is real even when the intervention is correct. Tell an agent struggling with a type error "the config returns strings, not integers" and the agent benefits from the information. But it also has to reconcile that fact with everything else in its context. If it had already half-formed a different theory of the bug, the correct hint creates a moment of confusion before it creates clarity. That moment costs a frame or two.

The disruption cost varies with the type of intervention. A small hint ("check the type of the return value") is low disruption. It adds one sentence to the context. A large context injection ("here is the entire module you should be looking at, along with its documentation") is high disruption. It adds a wall of text that the model has to process. Resetting the task boundary ("ignore the last ten frames and start this subtask over") is maximum disruption. Each type has a characteristic cost profile learned from data.

Opportunity cost. The agent might have recovered on its own. If you intervene at frame t and the agent would have naturally recovered at frame $t + 1$, the intervention was pure waste. Worse, it may have disrupted a recovery that was already in progress. The agent was about to read the right file, form the right conclusion, and fix the issue. The intervention arrived one frame too early, added noise, and delayed the recovery it was trying to help.

The opportunity cost is proportional to the recovery probability. When p_r is high, intervention is likely unnecessary and potentially harmful. When p_r is low, the opportunity cost is small because the agent was unlikely to recover anyway.

Accuracy cost. The intervention might be wrong. The system's assessment of the trajectory state is probabilistic. The fused signal might indicate that the agent is struggling with a type mismatch when it is actually struggling with an import error. An intervention based on the wrong diagnosis adds noise to the context and can push the trajectory further from recovery. A wrong hint is worse than no hint, because it introduces a plausible but incorrect fact that the agent may trust and act on.

The accuracy of intervention diagnosis depends on how well the fused signal captures the nature of the problem, not just the dynamics of the trajectory. The fused signal is good at detecting that something is wrong. It is less good at identifying exactly what is wrong. This asymmetry matters for strategy selection.

The expected cost of acting is:

$$E[\text{act}] = C_{\text{disrupt}} + p_r \cdot C_{\text{opportunity}} + (1 - p_a) \cdot C_{\text{wrong}}$$

where p_a is the probability that the intervention is appropriate for the actual situation.

The Intervention Point

The optimal intervention point is the frame where the expected cost of waiting first exceeds the expected cost of acting:

$$t^* = \min\{t : E[\text{wait}]_t > E[\text{act}]_t\}$$

Before t^*, let the agent work. After t^*, act.

This framing clarifies several intuitions that experienced operators already have.

Let natural recovery occur first. During exploring, there is nothing to recover from. At the first sign of trouble, p_r is still high—most agents recover from their first setback. In both cases, $E[\text{wait}]$ is low, so the calculus says wait.

Intervene before drifting, not during. Once the trajectory has entered drifting, the context is too degraded for intervention to be cheap or effective. The optimal intervention point almost always falls in late recovering, before the transition to drifting. This is counterintuitive: the moment when the problem is most obvious (the agent is drifting, clearly wasting tokens) is the worst moment to intervene. The intervention calculus resolves this by computing expected costs rather than relying on visual salience.

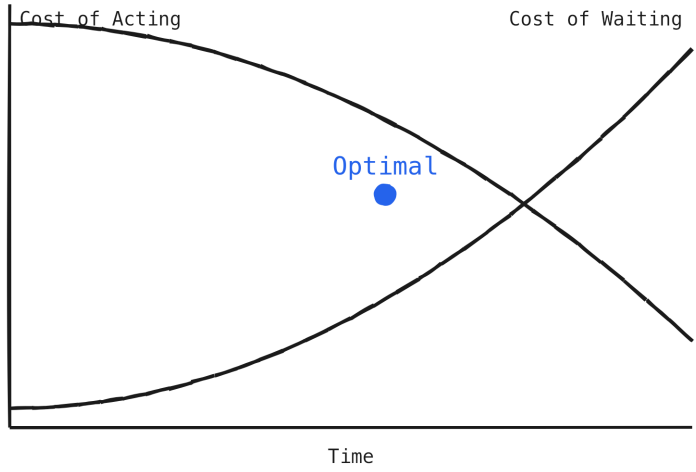

Figure 13.1: Intervention timing: cost crossover between waiting and acting

Repeated recovery attempts shift the calculus. Each failed recovery reduces p_r and increases C_d. An agent on its third recovery attempt from the same obstacle has a much higher $E[\text{wait}]$ than an agent on its first. The intervention point arrives earlier with each failed attempt. After two failed recoveries, the system should be strongly biased toward acting.

In practice, the intervention point is not computed from these equations directly. The equations describe the logic. The actual decision is made by a learned model that takes the fused signal as input and outputs an intervention probability. The model has been trained on thousands of trajectories where interventions were applied at different points and the outcomes were measured. It has internalized the cost structure described above without being explicitly programmed with the equations.

What Intervention Looks Like

Intervention is not "try harder." It is not encouragement. It is not a rephrasing of the original task. Those approaches add tokens without adding information. Telling an agent that is struggling "please try to fix the bug" is like telling a lost hiker "please try to find the trail." The instruction contains no information the recipient does not already have.

Effective intervention changes the conditions under which the agent is operating. It modifies the problem, not the agent. The agent's weights are fixed. Its reasoning patterns are fixed. What can change is the environment: what tools are available, what information is in the context, what the scope of the task is. The harness controls these conditions. The intervention adjusts them.

There are five primary intervention strategies, ordered from lowest to highest disruption.

Narrow the Tool Set

Remove tools the agent does not need for the current subtask. A drifting agent with ten available tools has ten possible next actions; reduce to three and productive actions become more likely. This is the lowest-disruption intervention because it adds nothing to the context—it only re-

moves options. In fused space, tool restriction typically causes an immediate inertia increase and entropy stabilization.

Provide a Codebase Hint

Inject a targeted fact into the agent's context: "The `Session` model has a `default_ttl` class method," or "The error on line 47 is a type mismatch, not a missing import." The hint should provide a fact the agent lacks, not an instruction. Facts integrate into the agent's existing reasoning; instructions override it. When accurate, hints move the fused signal toward converging. When inaccurate, they push the trajectory further off course—making accuracy the key risk.

Reset the Subtask Boundary

Tell the agent to focus on a smaller piece of the problem: "Just fix the type conversion. Do not refactor the config module." Moderately disruptive because it overrides the agent's own scoping, but powerful for agents that have expanded focus too broadly during recovery. In fused space, inertia drops briefly as the agent processes the new scope, then rises sharply as it re-commits to the narrower target. Entropy decreases as out-of-scope context becomes ignorable.

Adjust the Working Set

Change what the agent can see: close irrelevant files, open relevant ones, replace the context summary with a focused one. Highly disruptive to the current execution state, but an agent whose context is 80% irrelevant observations benefits enormously. This is not amnesia—a well-designed summary preserves key findings ("investigated module X, ruled out Y, the issue is in Z") while dropping the verbose details. The tradeoff favors adjustment when entropy exceeds a learned threshold.

Checkpoint and Restart

The nuclear option: save progress, clear the context, and restart the subtask with a summary of what was learned. Appropriate when entropy is so high that the context itself has become the obstacle—more noise than sig-

nal, beyond what lighter interventions can fix. The summary is the critical element: a raw restart wastes everything learned, while a good summary preserves conclusions and discards the noise that buried them. In fused space, checkpoint and restart teleports the trajectory back to exploring, but with a different starting position informed by the previous attempt.

Strategy Selection via Fused Space

Given that intervention is warranted, which strategy should be used?

The answer comes from the fused signal and the historical record.

The system maintains a database of past trajectories, each annotated with the interventions that were applied and the outcomes that resulted. When an intervention decision is triggered at frame t, the system looks up trajectories whose fused state at their intervention point was near $\mathbf{f}_t$ in embedding space.

This is a nearest-neighbor lookup. Find the historical trajectories that were in a similar state. See what interventions were applied. See which interventions led to recovery.

The lookup is geometric, not categorical. It does not match on task type or failure type or any human-defined category. It matches on fused state, which encodes all of those things implicitly. Two trajectories might involve completely different codebases and completely different bugs, but if their fused states are similar at the intervention point, the intervention that worked for one is likely to work for the other. They are in the same neighborhood in behavior space.

Why does this work? Because the fused signal abstracts away the specific content of the session and captures its behavioral structure. An agent struggling with a type mismatch in Python and an agent struggling with a type mismatch in TypeScript may have completely different codebases and completely different tool calls, but their fused signals at the point of struggle are similar: high inertia (committed to an approach), rising entropy (conflicting evidence), spatial position in the "local failure" region. A hint that worked for the Python case ("check the return type of the config getter") has an analog that works for the TypeScript case ("check the return type of the config getter"). The content of the hint is different. The

strategy (provide a targeted codebase hint about type information) is the same.

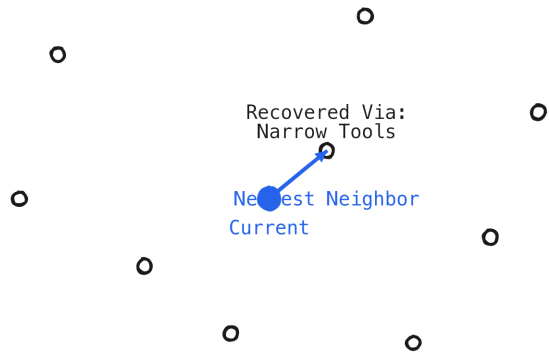

Figure 13.2: Strategy lookup via nearest neighbor in fused embedding space

Strategy selection, then, is a weighted vote of nearby historical interventions, weighted by the outcomes they produced and the distance from the current fused state. The nearest neighbor with the best outcome gets the most weight. But all neighbors within a radius contribute, because the right strategy for a given fused state is not always the same. Sometimes multiple strategies work. The system selects the one with the highest expected recovery probability given the current state.

This approach has a bootstrapping problem. Early in the system's life, there are few historical trajectories. The nearest-neighbor database is sparse. Strategy selection is unreliable. This is expected and acceptable. Early interventions are conservative: prefer low-disruption strategies (narrowing tools, providing hints) because their downside is small. As the database fills, strategy selection becomes more precise and the system can confidently recommend higher-disruption interventions when the data supports them.

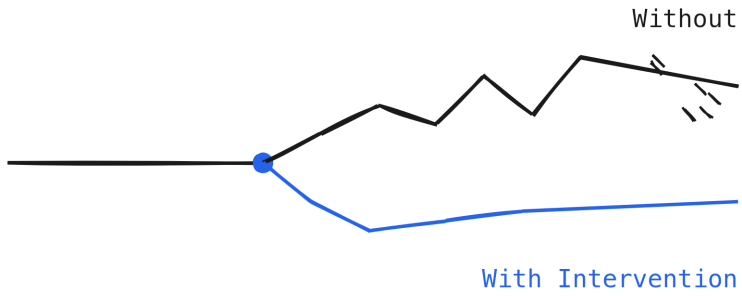

Figure 13.3: Before and after intervention: trajectory correction in fused space

The Intervention as Experiment

Every intervention is also a data point.

The system records the fused state at the intervention point, the strategy selected, and the outcome: did the trajectory recover? How many frames did recovery take? What was the final result? This record becomes a training example for the intervention model.

Over time, the model learns not just when to intervene but how well each strategy works in each region of fused space. Some neighborhoods respond well to hints. Others need boundary resets. Some are best left alone because recovery is almost certain. The model accumulates this knowledge through operation.

This is the first hint of the closed loop that Chapter 14 will formalize. The system's interventions generate the data that improves the system's interventions. Each session is both a service performed and a lesson learned.

But there is a tension. If the system always intervenes at the optimal point with the optimal strategy, it never observes what would have happened without intervention. It never collects the counterfactual. This makes it hard to know whether the interventions are actually helping or whether the agent would have recovered on its own.

The resolution is deliberate experimentation. A small fraction of sessions, perhaps 5-10%, are run with interventions withheld or randomized. These sessions serve as a control group. They are more expensive individually because some will drift when intervention would have saved them. But they are necessary for calibrating the intervention model. Without them, the model cannot distinguish between "my interventions are working" and "these trajectories would have recovered anyway."

The experiment rate is itself a tunable parameter. Early on, when the intervention model is uncertain, a higher experiment rate generates more useful data. Later, when the model is well-calibrated, the experiment rate can be reduced. The system learns how much to experiment, which is a meta-optimization over the intervention optimization.

Knowing when to intervene and how to intervene is the difference between monitoring and coaching. A monitoring system observes and reports. A coaching system observes, assesses, and acts. The fused signal provides the assessment. The intervention calculus determines the timing. The strategy selection determines the method. Together, they transform passive measurement into active trajectory management.

The next chapter closes the loop entirely. Intervention outcomes feed back into the measurement and characterization pipeline, improving every component of the system with each session it runs. The coach gets better at coaching by coaching.

Chapter 14

The Closed Loop

Five steps.

Measure the trajectory frame by frame. Characterize its phase and dynamics using the fused signal. Intervene when the calculus says to. Learn from the outcome. Use what was learned to improve the next cycle.

The last step feeds the first. That is what makes the loop closed.

Most agent tooling today runs open loops: collect telemetry, analyze offline, manually update the prompt, run again. The feedback path is a human reading dashboards on an hours-or-days cycle. A closed loop eliminates that bottleneck—the system measures, learns, and adjusts continuously, each session making the next slightly better through structural feedback. The concept is old (control theory formalized it in the 1940s), but the specific machinery here is new: the "temperature" is a fused embedding vector, the "setpoint" is successful completion, and the "furnace" is the set of intervention strategies.

This chapter describes the five steps and how they connect.

Step 1: Measure

Everything begins with frames.

Each turn in an agent session produces a frame: the full state of the agent at that moment, including the action taken, the observation received, the context window contents, and any metadata the harness records. From each frame, the system extracts the three base signals. Inertia, computed from the direction and magnitude of the trajectory's movement in embedding space. Entropy, computed from the information-theoretic properties of the context. Space, computed from the trajectory's position relative to historical trajectories.

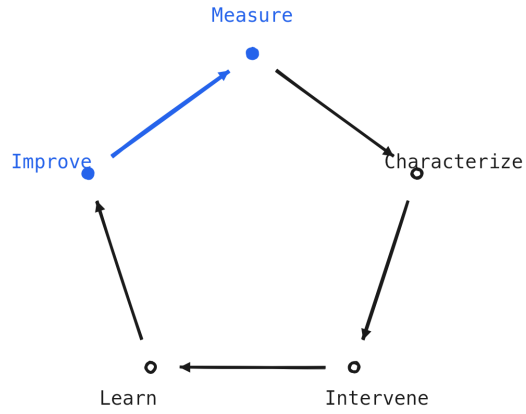

Figure 14.1: The closed loop: five-step cycle from measurement to improvement

These signals flow into the gating function, which composes them into the fused embedding $\mathbf{f}_t$. The fused embedding is the primary measurement. Everything downstream operates on it.

The measurement is continuous and automatic. There is no separate telemetry pipeline. There is no batch process that runs after the session ends. The frame is the telemetry. The act of running the agent is the act of producing measurements.

This is worth pausing on, because most production systems treat measurement as an overhead to be minimized. Telemetry is expensive. It adds latency. It consumes storage. Systems are designed to sample sparingly, emit only summary statistics, and defer analysis to offline jobs. The implicit assumption is that measurement and operation are separate activities that compete for resources.

In the framework described here, measurement is operation. The fused signal that the system computes at each frame is the same signal used to make intervention decisions in real time. The embeddings that locate the trajectory in space are the same embeddings that will be used for training

after the session ends. Nothing is measured speculatively. Nothing is measured "just in case." Every measurement serves the current session and the future training pipeline simultaneously.

This dual purpose changes the economics. Traditional telemetry has a cost and an indirect benefit: you pay to collect data, and later you might use it to improve the system. Here, the measurement has both an immediate benefit (informing the current session's intervention decisions) and a deferred benefit (training data). The immediate benefit justifies the cost by itself. The deferred benefit is free.

The cost of measurement is not zero. Computing embeddings at each frame takes compute cycles. Extracting entropy from the context requires processing the context. Running the gating function requires a forward pass through a small network. But the cost is bounded and predictable, because the measurements are fixed-dimensional vectors, not variable-length logs. A 128-dimensional fused embedding costs the same to compute and store regardless of whether the agent's context window contains 1,000 tokens or 100,000. The measurement cost does not grow with session length or task complexity.

In practice, the measurement overhead is on the order of 1-5% of the total compute spent on the agent's LLM calls. The embedding extraction is cheap relative to running a large language model. The gating function is a few matrix multiplications. The per-frame cost is measured in milliseconds, while the agent's turn is measured in seconds.

Step 2: Characterize

Raw measurements become useful when they are interpreted. The fused signal at each frame tells the system what phase the trajectory is in, how it got there, and where it is likely heading.

Characterization is the step where numbers become assessments.

The phase classifier maps the fused state to one of the six phases defined in Chapter 12: exploring, converging, compacting, recovering, completing, or drifting. Each classification comes with a confidence score. High confidence means the fused state is deep inside a phase cluster. Low confidence means it is near a boundary.

The JEPA predictor projects the fused state forward to estimate the next frame's position in fused space. The prediction provides a velocity estimate: where the trajectory is heading and how fast.

The transition detector watches for boundary crossings between phase clusters. It combines the current phase classification, the confidence score, and the JEPA prediction to estimate the probability of a phase transition in the next one to three frames.

Together, these components produce a running characterization of the trajectory. The characterization at frame t might read: "Compacting, confidence 0.92, predicted to remain in compacting, transition probability 0.08." That is a healthy trajectory. Leave it alone.

Or: "Recovering, confidence 0.71, predicted transition to drifting, transition probability 0.63." That is a trajectory in trouble. The intervention engine needs this characterization now, not after the session ends.

The characterization is a structured object, not a natural-language description. It contains the current phase label, the confidence of that label, the predicted next phase, the transition probability, the recovery probability (if in a recovery phase), and the fused embedding vector itself. Downstream systems consume this object programmatically. The intervention engine uses it to decide whether to act. The training pipeline uses it to label the frame for future learning.

Characterization runs at frame speed. The phase classifier and JEPA predictor are small models, typically a few thousand parameters each, operating on low-dimensional inputs. They add microseconds to each frame, not milliseconds. The characterization is available before the next agent turn begins. The entire measure-characterize pipeline, from raw frame to structured assessment, completes within the latency budget of a single agent turn.

Step 3: Intervene

Chapter 13 described the intervention calculus in detail. Here we place it in the loop.

The intervention engine receives the characterization at each frame. Most frames require no action. The trajectory is in a stable phase, the predicted

next state is in the same phase, and the expected cost of waiting is low. The engine does nothing. This is the common case. In a typical session, the intervention engine acts on perhaps one or two frames out of thirty or forty.

When the characterization signals a problem, rising entropy, declining recovery probability, a predicted transition to drifting, the engine evaluates whether to act. It compares $E[\text{wait}]$ to $E[\text{act}]$ using the estimates from the fused signal and the historical database. If the expected cost of waiting exceeds the expected cost of acting, it selects a strategy and applies it.

The intervention itself is a modification to the agent's operating conditions. A tool restriction. A context hint. A boundary reset. A working set adjustment. A checkpoint and restart. The specific choice comes from the nearest-neighbor lookup in fused space, selecting the strategy that worked best for historical trajectories in a similar state.

The intervention is recorded alongside the frame data. The record might look like: "Frame 34, phase recovering, recovery probability 0.31, predicted transition drifting at 0.63, intervention applied: narrow tool set to file_read, file_edit, run_test." This record is a training example. Whether the intervention led to recovery will be determined when the session ends. For now, it is simply stored.

The gap between intervention and outcome is one of the loop's central design challenges. The system acts now and learns later. During that gap, it has to live with uncertainty about whether it made the right call. This is fine. The system is not trying to be right about every individual intervention. It is trying to be right on average, across hundreds of sessions, where the law of large numbers ensures that the aggregate signal overwhelms the noise of individual cases.

Step 4: Learn

The session ends. The agent either completed the task, failed, timed out, or was terminated. The outcome is known.

Now every frame in the session becomes a labeled training example.

The label is not just pass or fail. It is the outcome trajectory: the full path from each frame to the end of the session, including the final result. A

frame at the beginning of a successful session is labeled differently from a frame at the beginning of a session that succeeded only after a costly recovery. Both sessions passed. The trajectories were different. The labels should reflect that.

This is where trajectory-level measurement pays off. Traditional telemetry records summary statistics. The closed loop records the full trajectory. Every frame has a before and an after. The before is the fused signal at that frame. The after is everything that happened subsequently. This before-after pair is the training signal.

Concretely, each frame's training label includes:

Outcome. Binary success/failure—simple, clean, and correlated with everything else.

Remaining cost. Frames and tokens from this frame to the end, teaching the system to distinguish on-track trajectories from those heading for expensive conclusions.

Phase accuracy. Whether the phase label at this frame was correct in retrospect, training the classifier to detect transitions earlier and penalizing confident misclassifications.

Intervention effectiveness. For intervened frames, did the trajectory improve relative to the JEPA-predicted counterfactual (what would have happened without intervention)? The estimation is imperfect per-case, but across many interventions converges to a useful signal for evaluating strategies.

Prediction error. The gap between the JEPA forecast and the actual next state, revealing where the predictor's model of trajectory dynamics is weakest and where it is well-calibrated.

These labels are generated automatically from the trajectory data. No human annotator is needed. No external evaluation pipeline is needed. The trajectory labels itself through its own evolution. The outcome of the session provides the ground truth. The trajectory provides the detailed frame-by-frame signal. The combination produces a rich, fine-grained training set from every session the system runs.

Step 5: Improve

The labeled frames from Step 4 are training data for the components described in Steps 1 through 3.

The fused signal's gating function is trained to produce compositions that maximize outcome prediction accuracy. If the current gating weights produce fused states where successful and failed trajectories are hard to distinguish, the weights are wrong. Training adjusts them so that the fused space separates outcomes more cleanly. The gating function learns to attend to the signal combinations that best predict what will happen next. If entropy rate-of-change is a better predictor of failure than absolute entropy, the gating function learns to weight the derivative more heavily. If spatial position in a particular region is uninformative, the gating function learns to ignore space there and rely on inertia and entropy.

The phase classifier is trained on the phase accuracy signal. If it is consistently late in detecting transitions, its decision boundaries shift to be more sensitive to the early indicators of phase change. If it is miscategorizing recovering as compacting (because the agent is still making edits during recovery), its feature extraction adjusts to recognize the entropy and inertia patterns that distinguish productive editing from recovery editing.

The JEPA predictor is trained on the prediction error signal. If it consistently fails to predict transitions out of compacting, it needs more capacity or training data in that region of fused space. If it over-predicts transitions to drifting (generating false alarms), its estimates are too conservative and need recalibration.

The intervention model is trained on the intervention effectiveness signal. If narrowing the tool set works well for trajectories in one region of fused space but poorly in another, the model learns to recommend tool narrowing only where it is effective. If codebase hints work when the spatial signature indicates a specific, local failure but not when it indicates a broad architectural misunderstanding, the model learns that distinction.

All of this training happens on the data produced by normal operation. There is no separate data collection phase. There is no annotation team. There is no synthetic data generation. The system produces training data as a byproduct of doing its job. Every session is a training batch. Every intervention is a labeled experiment.

The training cadence depends on the system's throughput. A system processing hundreds of sessions per day can retrain nightly. A system processing thousands per hour can retrain continuously. The components are small enough that training is fast. The fused signal gating function has thousands of parameters, not billions. The phase classifier is a shallow network. The JEPA predictor is a compact sequence model. Full retraining on a day's data takes minutes, not hours. Incremental updates take seconds.

After retraining, the improved components are deployed. The next session benefits from the updated models. Its frames are measured more accurately, its phases are classified more precisely, its transitions are predicted earlier, and its interventions are more effective. And its frames, in turn, become training data for the next round of improvement.

The loop closes.

Why Offline Evaluation Is Not Enough

Benchmarks test a fixed distribution of tasks. SWE-bench has 2,294 instances, drawn from twelve Python repositories, representing issues resolved between specific dates. An agent that scores well on SWE-bench has demonstrated capability on that distribution.

But the distribution of work that a production agent encounters is different. It changes over time. The codebases evolve. New frameworks emerge. Coding patterns shift. The tasks in January are not the same as the tasks in March. An agent evaluated exclusively on benchmarks is calibrated for a snapshot of the world that no longer exists.

The closed loop evaluates on the actual distribution of work. Every production session is a test case. The outcome provides a ground truth signal. The system's prediction accuracy, phase classification accuracy, and intervention effectiveness are continuously measured against real work, not held-out benchmarks.

There is a deeper issue with offline evaluation. Benchmarks test the agent in isolation. The closed loop tests the entire system: agent plus measurement plus intervention. A benchmark can tell you that Agent A solves 63% of tasks and Agent B solves 67% of tasks. It cannot tell you that

Agent A with trajectory-based intervention solves 72% of tasks, because the intervention is not part of the benchmark. The closed loop measures the performance of the whole system, agent and harness together, on the actual work the system is deployed to do.

This does not make benchmarks useless. Benchmarks provide a controlled baseline for comparing systems. They ensure reproducibility. They catch regressions. They are invaluable for development and debugging. But they should not be the primary evaluation signal for a production system. The primary signal should come from the closed loop, where the system is tested on exactly the distribution it serves.

The analogy is quality control in manufacturing. You test a sample of products against a specification (the benchmark). But you also monitor the production line continuously (the closed loop). The sample test catches design flaws. The production monitor catches process drift. You need both. But if you can only have one, you want the production monitor, because it is testing the thing that actually ships.

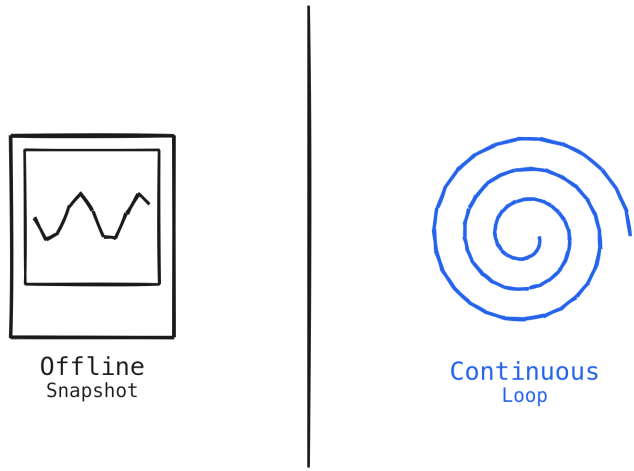

Figure 14.2: Offline evaluation vs continuous closed-loop evaluation

Data as Byproduct

The closed loop's most underappreciated property is that it does not require a data pipeline.

Traditional machine learning systems have a pipeline problem. Data must be collected, cleaned, formatted, labeled, and fed into a training process. Each step is a potential failure point. Data collection may miss important cases. Cleaning may discard valuable outliers. Formatting may lose structure. Labeling is expensive and error-prone. The pipeline itself requires engineering effort to build, maintain, and debug. In many organizations, the data pipeline is more complex and more fragile than the model it feeds.

The trajectory framework sidesteps most of this. Data is collected by operating the agent. The frames are the measurements, produced at the same time as the work they describe. No separate collection step exists. You cannot run the agent without collecting data. The data is the measurement is the operation.

Labeling happens retroactively when the session ends, using the outcome to label all frames. The labeling is deterministic: given the trajectory and the outcome, the labels follow from the formulas in Step 4. No human judgment is involved. No inter-annotator agreement to worry about. No ambiguous cases to adjudicate. The labels are as reliable as the outcome signal, which is as reliable as the test suite or evaluation function that determines whether the agent succeeded.

Cleaning is minimal because the data format is fixed. Each frame produces a fixed-dimensional embedding and a structured metadata record. There are no free-text fields to parse, no variable schemas to normalize, no encoding issues to resolve. The data arrives in the format the training pipeline expects.

The system does not have telemetry. It has data. The distinction is that telemetry is information collected for reporting, consumed by humans staring at dashboards. Data is information collected for learning, consumed by training algorithms that improve the system. In this framework, every measurement is collected for learning. The reporting is a side effect. If someone wants a dashboard, they can build one from the same embeddings and characterizations. But the dashboard is not the point. The learning is the point.

This changes the economics of improvement. In a traditional system, improving the model requires a data effort: weeks or months of collection, annotation, and curation. In the closed-loop system, improving the model requires operating the system. More sessions produce more training data. Better training data produces better models. Better models produce better sessions, which produce better data. The improvement is self-funded. It costs whatever it costs to run the agent, which was going to happen anyway because running the agent is the system's purpose.

Improvement Over Time

The predictions get better. This is an empirical claim, not an aspirational one.

Each cycle through the loop adds training data that covers regions of fused space that previous cycles may have missed. The improvement follows a characteristic curve, steep at first and flattening over time, as the system encounters and learns from an expanding portion of the trajectory space.

The pattern is consistent: more data produces better accuracy, which produces better interventions. After a hundred sessions, phase clusters are forming but intervention strategy selection is still crude. After a thousand, phase classification is reliable for common phases and intervention decisions approach experienced-operator quality. After ten thousand, all six phases classify accurately, transitions predict one to two frames ahead, and interventions are competitive with or better than human judgment. After a hundred thousand, common-case accuracy has plateaued and the remaining value comes from the tail—catching rare edge cases that no human operator would have enough experience to recognize.

The improvement curve flattens, as all learning curves do. But the flattening happens at a useful level. The system does not need perfect prediction. It needs prediction that is accurate enough to time interventions correctly most of the time and confident enough to know when it is uncertain. The uncertainty signal itself is useful. When the system says "I do not know what this trajectory will do next," that is a flag for conservative action: do not intervene, let the agent try, and learn from what happens.

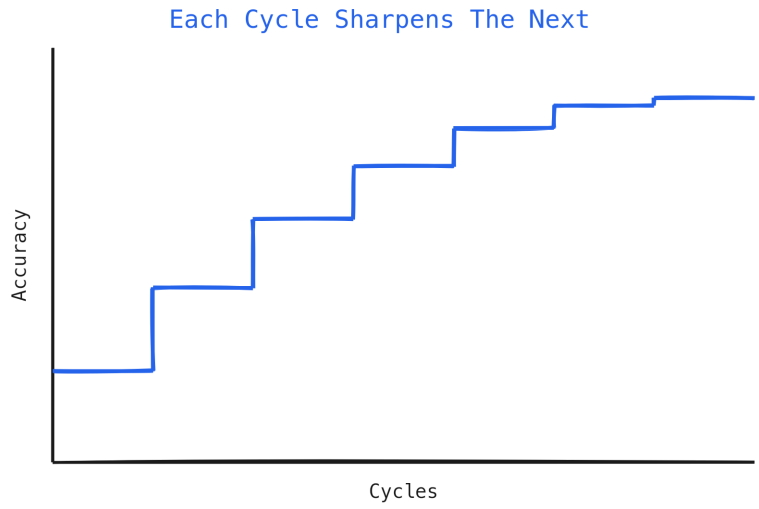

Figure 14.3: Improvement over cycles: prediction accuracy increasing with accumulated sessions

The Full Picture

Parts I through III of this book described what to measure and why. Part IV has described what to do with the measurements.

Chapter 11 composed three signals into one, creating a fused representation that captures what each signal knows individually and what the interactions between them reveal. Chapter 12 used the fused signal to define phases, the recurring patterns that trajectories follow, and to predict transitions between them. Chapter 13 developed a theory of intervention: when to act, what to do, and how to choose between strategies based on geometric similarity in the fused space. This chapter closed the loop, connecting intervention outcomes back to measurement improvement through a five-step cycle that runs continuously.

The five steps of the loop are not independent modules bolted together. They are a single system where each step produces the inputs for the next. Measurement produces characterization. Characterization enables intervention. Intervention generates data. Data drives improvement. Im-

provement refines measurement. The cycle has no beginning and no end, just continuous rotation.

The machinery is designed so that the system improves by existing. It does not need a team of engineers curating data and retraining models on a quarterly basis. It does not need periodic offline evaluations to check whether the models have drifted. It evaluates itself continuously against the actual distribution of work, and it updates itself continuously using the data that work produces.

Part V will show what this looks like in practice. Not as theory, but as architecture. Concrete systems that implement the closed loop, with specific design choices, engineering tradeoffs, and lessons from building and operating these systems in production. The framework is complete. The implementation begins.

Part V

Architecture

Chapter 15

The Write-Back Architecture

Consider two agents. Both solve a software engineering task. Both produce a correct fix. Both generate trajectories that could, in principle, be analyzed using the framework developed in Parts I through IV.

Agent A runs inside a standard scaffolding system. It has a context window, some tool calls, and a result. To measure its trajectory, an external observer must instrument the scaffolding, capture tool call logs, reconstruct the sequence of actions after the fact, and run the information-theoretic computations from Chapters 4 and 5 over the reconstructed data. The measurement system is bolted on. It watches from the outside. Every signal it captures is a reconstruction of something that already happened.

Agent B runs inside a system designed differently. When it reads a file, that read is recorded not in a separate telemetry database but as a mutation to a construct that the agent itself uses as working memory. When it forms a plan, the plan is a construct the agent can reference, modify, and extend. When it edits code, the edit is tracked as a mutation to a construct that captures what changed, why, and what it replaced. The agent's work product and its measurement record are the same artifacts.

Agent A's trajectory must be reconstructed. Agent B's trajectory simply exists. It was produced as a byproduct of doing the work.

This chapter describes the architecture that makes Agent B possible.

Bolted-On Observability

The standard pattern in software systems, not just agent systems, is to build the application first and add observability second. You write the business logic. Then you add logging statements. Then you add metrics

collection. Then you add distributed tracing. Each observability layer is an addition to the system, consuming resources and engineering effort to capture a shadow of what the system already did.

This pattern has well-known failure modes. Logs are verbose but unstructured. Metrics are structured but dimensionally impoverished. Traces capture causality but not semantics. The observability stack operates in a different world from the application stack, connected by fragile bridges of log lines and metric emissions that developers must maintain by hand. When the application changes, the observability layer lags behind. When the observability layer becomes too expensive, it gets sampled down, and information is lost.

For agent systems, the bolted-on pattern is worse than usual. The thing you most want to observe, the agent's reasoning process, is exactly the thing that is hardest to instrument from outside. An external observer can see tool calls. It can see token counts. It can see timing. It can see results. It cannot see the relationship between a hypothesis formed at Turn 3 and a decision made at Turn 8, because that relationship exists inside the model's context, not in the scaffolding's data model.

The framework developed in this book, frames, trajectories, inertia, entropy, space, fused signals, phase detection, intervention theory, all of it requires rich structured data about the agent's journey through a task. Bolted-on observability can provide some of this data. But it will always be incomplete, delayed, and expensive to maintain.

There is another way.

The Core Principle

A Write-Back Architecture (WBA) is an architecture where the system writes back to its own substrate. The agent does not produce work in one place and emit telemetry in another. The work itself is the telemetry.

This requires a specific property of the substrate: the artifacts the agent produces must be structured enough to serve as measurement records without additional annotation. A plain text file is not structured enough. It captures content but not intent, not history, not the relationships between this artifact and the ones that preceded it. A database row is too struc-

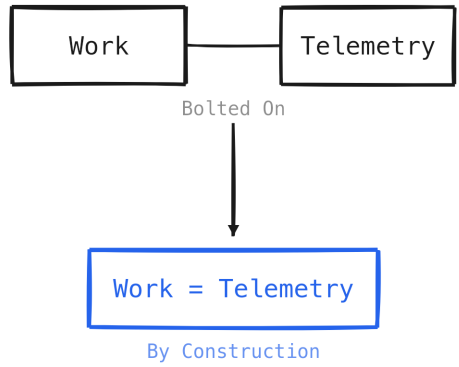

Figure 15.1: Traditional bolted-on observability vs Write-Back Architecture

tured: it captures fields and values but loses the semantic context that makes the data meaningful.

The WBA needs artifacts that carry their own semantics. Artifacts that are simultaneously the thing the agent produced and the record of how and why it was produced.

In one concrete implementation of this architecture, these artifacts are called *bit constructs*. They are not the only possible implementation. The architecture pattern is independent of any specific artifact format. But constructs illustrate the principle well, and since this chapter is about making theory concrete, we will use them as the primary example.

Bit Constructs as Source of Truth

A bit construct is a unit of structured information expressed in a linguistic programming format. “Linguistic” because it is readable by both humans and machines without requiring a separate schema definition or IDL. “Pro-

gramming" because it carries executable semantics: constraints, relationships, lifecycle rules. "Format" because it is a notation, not a runtime.

A construct looks something like this, expressed in simplified pseudocode:

```
construct TaskPlan {
  intent: "Fix session expiry handling"
  status: active

  hypothesis {
    description: "create_session ignores default TTL from config"
    confidence: 0.8
    based_on: [read:session.py, read:config.yaml]
  }

  steps [
    { action: "modify create_session", status: complete },
    { action: "add config fallback", status: in_progress },
    { action: "run tests", status: pending }
  ]

  mutations [
    { frame: 3, field: "hypothesis.confidence", from: 0.5, to: 0.8 },
    { frame: 5, field: "steps.0.status", from: "in_progress", to: "complete"
  ]
}
```

The construct carries intent, history, relationships, and its own schema—all visible in the notation itself. Intent is a first-class field, not a comment. Mutations record the path to current state as frames (Chapter 4). Relationships like `based_on` are embedded in the construct, not maintained in a separate graph. And the structure is self-describing: no external schema or registry required.

This last property is what makes the WBA possible. Because the construct is self-describing, the agent can both produce and consume it without a translation layer. The writing, the remembering, and the measuring all operate on the same artifact.

No separate telemetry layer. The work is the telemetry.

How WBA Enables Context Dynamics

The framework from Parts I through IV defines specific quantities that must be measurable for the theory to have practical value. Walk through each one and see how a WBA makes it naturally available.

Frames

In a bolted-on system, frames must be reconstructed from tool call logs, with no guarantee the log is complete or the call succeeded. In a WBA, every construct mutation *is* a frame—recorded with frame number, timestamp, and delta. The frame exists because the work that produced it exists.

Trajectories

Bolted-on trajectories must be assembled offline from scattered sources. In a WBA, the trajectory is the union of construct mutation histories, ordered by frame number. It needs no reconstruction because it was never disassembled.

Inertia

Bolted-on inertia measurement requires parsing model outputs and classifying them by topic—feasible but fragile. In a WBA, inertia is directly observable as accumulated construct state: a `TaskPlan` with forty mutations has substantial inertia; one with three mutations has less. The mass is right there in the data.

Entropy

In a WBA, the construct's mutation history *is* the action distribution—no estimation from tool call logs required. Consistent mutation patterns (hypothesis, plan, execute, test, confirm) indicate low entropy; erratic patterns indicate high entropy. Inter-construct consistency provides an additional coherence signal that bolted-on observability would struggle to capture at all.

Space

Context space (Chapter 9) shapes agent movement through task relationships, file connections, and concept clusters.

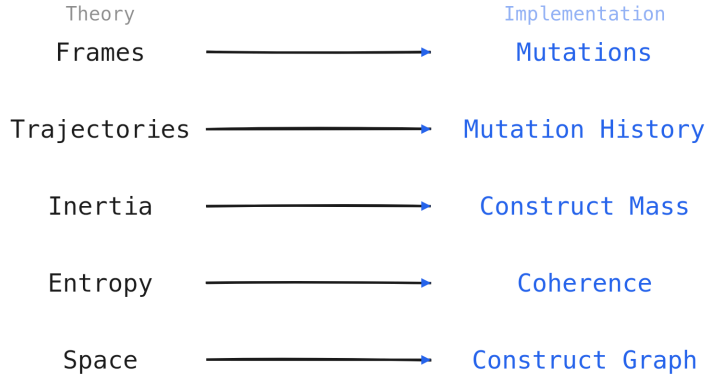

Figure 15.2: Framework measurements mapped to WBA components

In a WBA, the space is the graph of construct relationships. Every `based_on`, `depends_on`, `relates_to`, and `derived_from` link is an edge in a graph. The constructs are nodes. The graph *is* the space. It does not need to be inferred from file access patterns or import graphs. The agent built it as a byproduct of working.

This graph evolves as the session progresses. New constructs add nodes. New relationships add edges. The topology of the space changes in real time as the agent works. Measuring properties of the space (density, connectivity, diameter) requires only reading the constructs.

A Concrete Walk-Through

Ground this with a specific scenario. An agent fixes a race condition in a session management module. Over five frames it reads the failing test

(Frame 1, creating an Investigation construct), reads the session module (Frame 2, adding an observation with an explicit link to Frame 1), forms a hypothesis (Frame 3, recorded with confidence score and evidence links), writes the fix (Frame 4, a CodeEdit construct linked to the hypothesis), and runs the test (Frame 5, a TestResult construct linked to the edit).

In a bolted-on system, these five frames produce five log lines: read, read, edit, run. The connections between them must be inferred from proximity. In the WBA, the constructs carry explicit causal links: Investigation produced hypothesis, hypothesis produced CodeEdit, CodeEdit produced TestResult. The trajectory is recorded, not reconstructed.

The agent benefits too. If the test had failed, it could re-read its Investigation construct to recall its hypothesis and evidence—externalized working memory rather than information buried in an ever-growing context window.

The Measurement Byproduct

Notice the pattern across all five measurements. In a bolted-on system, each quantity requires an explicit measurement step: capture data, reconstruct the state, compute the metric. In a WBA, each quantity is a byproduct of normal operation. The agent does its work. The constructs record the work. The measurements are properties of the constructs.

This has a consequence that is easy to state but hard to overstate: measurement in a WBA has zero marginal cost.

Not "low cost." Zero. The agent does not spend extra tokens emitting telemetry. The system does not run a separate pipeline to reconstruct trajectories. The operators do not maintain a logging infrastructure. The constructs exist because they are the agent's working memory. The measurements exist because they are properties of that working memory.

This matters for the closed loop described in Chapter 14. A closed loop requires continuous measurement feeding back into the system. If measurement is expensive, the loop is slow and lossy. If measurement is free, the loop can be tight. A WBA makes the loop tight by construction.

It also matters for training. The trajectory data that Parts III and IV described as inputs to phase detection, intervention timing, and strategy

selection, all of that data is available without any additional instrumentation in a WBA system. The agent's work produces the data. The training system consumes the data. The constructs mediate between them.

The work is the telemetry. The telemetry is the training signal. The training signal improves the work. The loop closes.

Construct Lifecycle and Mutation Semantics

A construct is not a static document. It has a lifecycle: creation, mutation, and eventual archival. Each stage produces information the framework can use.

Creation records intent. When an agent creates a construct, it declares what the construct is for. This is the initial state, the starting point of whatever trajectory the construct will accumulate. The creation event is Frame 0.

Mutation records evolution. Each change to the construct is a frame. The delta between consecutive frames captures what changed, and the construct's mutation history captures the full trajectory of that particular artifact. Mutations are typed. A status change is different from a content edit, which is different from a relationship addition. The type of mutation carries information about what phase the agent is in. Status changes cluster around transitions. Content edits cluster around execution. Relationship additions cluster around exploration.

Archival records completion (or abandonment). When a construct reaches its terminal state, whether successful or not, the archival event captures the outcome alongside the full history. A construct archived as `completed` tells one story. A construct archived as `abandoned` after thirty mutations tells a very different one. Both stories are fully recorded.

The lifecycle maps directly onto the phase structure from Chapter 12. Orientation produces constructs (the agent creates plans and hypotheses). Execution mutates them (the agent modifies and extends). Convergence archives them (the agent completes or discards). Phase detection in a WBA system is just lifecycle analysis of the active constructs.

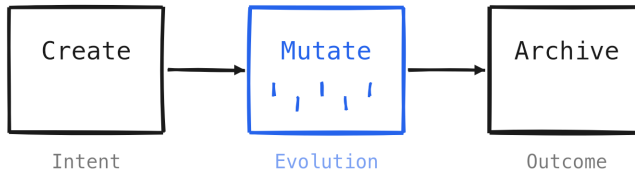

Figure 15.3: Construct lifecycle: creation, mutation, and archival

Temporal Properties

Every mutation carries a timestamp. The temporal dimension of the construct's history enables measurements that static analysis cannot provide.

Velocity: how fast is the construct changing? A construct accumulating mutations rapidly is in active development. One that has gone quiet may be complete or may be stalled. The mutation rate is a direct measure of velocity as defined in the trajectory formalism.

Acceleration: is the mutation rate increasing or decreasing? A construct that started slow and is now changing rapidly has different dynamics than one that started fast and is slowing down. The derivative of the mutation rate maps to trajectory acceleration.

Intervals: how much time passes between mutations? Regular intervals suggest systematic work. Irregular intervals with long gaps suggest the agent shifted attention elsewhere and returned. The gap pattern reveals attention allocation without needing to model the agent's internal state.

These temporal properties are available because the constructs record their own history. A bolted-on system would need to correlate timestamps across multiple data sources to reconstruct them. The WBA makes them structural.

The Versioning Question

One property of constructs deserves separate treatment because it connects to a question the framework has not yet addressed: version control.

When an agent mutates a construct, the mutation is recorded in the construct's history. But what if the mutation was wrong? What if the agent changed a hypothesis from correct to incorrect, or deleted a step that should have been kept, or added a relationship that does not exist?

In a traditional system, mistakes are either invisible (the log records what happened, not whether it was right) or corrected by overwriting (the wrong value is replaced with the right one, and the fact that it was ever wrong disappears). Neither is useful for trajectory analysis. Invisible mistakes produce false trajectories. Overwritten mistakes produce incomplete ones.

A WBA preserves all mutations, including the wrong ones. The construct's history is append-only. A correction is a new mutation that records both the corrected value and the fact that a correction occurred. The trajectory contains the mistake and the recovery, both informative: the mistake reveals reasoning at the time; the recovery reveals error-detection ability. An agent that corrects quickly shows low inertia; one that takes twenty frames to correct shows high inertia—the accumulated context resisted change even after the agent recognized the error.

The append-only property parallels git, which solved the same versioning problem for source code. Git records what humans did to files; constructs record what agents did to their reasoning artifacts. The measurement framework treats construct history the same way a software historian treats a git log: as ground truth, missteps included.

What a WBA Is Not

A database stores structured records for retrieval. A WBA is not a database. Constructs are not rows. They do not live in tables. They do not have fixed schemas. They carry their own structure and evolve over time.

A content management system organizes documents for human consumption. A WBA is not a CMS. Constructs are not documents. They are not primarily meant to be read by people. They are the substrate on which the agent operates. That they happen to be human-readable is a useful property, not the design goal.

A document store holds semi-structured data for flexible querying. A WBA is not a document store. Constructs do not sit passively waiting to be queried. They are active elements of the agent's working process. They change as the agent works. They carry relationships that form a graph. They accumulate history that forms trajectories.

A WBA is an architecture pattern. It says: the artifacts your agent produces while working should be structured enough that measuring the agent's behavior requires only reading those artifacts. No separate telemetry. No external reconstruction. No offline pipeline.

This is a design constraint, not a technology choice. It can be implemented with files on disk, with a graph database, with a distributed log, with whatever storage mechanism fits the system's requirements. The pattern is the same regardless of the storage: the work product and the measurement record are one thing, not two.

The Training Signal Identity

There is a deeper consequence of the WBA that connects back to the central argument of this book. The framework developed in Parts I through IV exists to provide better measurement of agent behavior. Better measurement enables better training, better deployment decisions, better understanding of what agents do and why.

In a system with bolted-on observability, the path from measurement to training is long. Data is captured, cleaned, reconstructed, formatted, and

fed into a training pipeline. Each step introduces latency, potential error, and engineering overhead.

In a WBA, the agent's work product *is* the training signal. The constructs that the agent produces during a session contain the trajectory, the phase transitions, the coherence measurements, the outcome. A training system that knows how to read constructs has access to the full richness of the framework's measurements without any intermediate processing.

This is not an incremental improvement in data pipeline efficiency. It is a structural identity. The work product and the training signal are the same thing. Not similar. Not derived from the same source. Identical. The construct the agent wrote while solving a task is the same construct that the training system reads to learn from that task.

The identity eliminates an entire class of problems: telemetry gaps (the construct cannot have gaps because the work filled it), reconstruction errors (there is no reconstruction), schema drift between the application and the observability layer (there is no separate layer), and information loss from sampling (every mutation is recorded because every mutation was an operation the agent performed).

This identity is what makes the closed loop from Chapter 14 practical rather than aspirational. The loop closes naturally because the output of one cycle is directly the input of the next.

Observable by Construction

Most discussions of observability treat it as a property you add to a system. "Make the system observable" means "add the ability to understand its internal state from its external outputs." The implication is that the system exists first, and observability is layered on top.

A WBA inverts this. The system does not become observable through instrumentation. The system is observable because of how it works. The architecture produces measurement as a structural byproduct of operation, the same way a river produces a riverbed as a structural byproduct of flowing.

This is not observability as a feature. It is observability as a consequence. A consequence of the decision to make the agent's work product carry its own history, its own relationships, and its own semantics.

The framework from Parts I through IV describes what to measure and how to interpret it. The WBA describes how to build systems where those measurements exist without additional effort. The framework is the theory. The WBA is one way to make the theory operational.

One way. Not the only way. An architecture that achieves the same structural observability through a different mechanism, different artifact format, different storage model, different lifecycle management, would serve the framework equally well. The principle is what matters: measurement as a byproduct of operation. The implementation details are engineering decisions.

But the principle is not optional. A system that requires extensive offline reconstruction to produce trajectory data will always lag behind a system that produces trajectory data while working. The lag matters because the closed loop from Chapter 14 is time-sensitive. The faster measurement flows back into the system, the tighter the loop, the faster the system improves.

The WBA does not add observability to an agent system. The architecture is observable by construction.

Chapter 16

The Harness Dynamics Model

The previous chapter described the Write-Back Architecture: a system where the agent's work product and its measurement record are identical. The WBA produces data. Rich, structured, temporally ordered data about everything the agent did, in the form it did it.

Data alone is not a system.

A warehouse full of weather station readings is not a weather prediction system. A filing cabinet of patient records is not a medical practice. Data becomes operational only when something consumes it, interprets it, and acts on it. The WBA is the body of the system. It produces the signals. Something needs to be the nervous system. Something needs to feel the signals, integrate them, and generate responses.

This chapter describes the Harness Dynamics Model: a coordination mechanism that consumes the data the WBA produces and uses it to govern agent behavior in real time. Together, the WBA and the Harness Dynamics Model close the loop that Chapter 14 described as the goal of the entire framework.

The Orchestration Problem

Agent systems need coordination: which agent works on what, when to intervene, when to abandon a failing strategy. The standard solution is a central orchestrator—a supervisory process that maintains a task queue, assigns work, and monitors progress. Central orchestration is simple and adequate for many workloads, but it has structural problems. It is a *bottleneck*: coordination logic grows with agent count, centralizing cognitive load in one component. It is *brittle*: if it fails, agents cannot self-organize. And it is *opaque*: the orchestrator shapes agent trajectories through de-

cisions that are not themselves observable through the trajectory framework.

There is another way to coordinate.

Pressure Fields

Weather systems do not have a central orchestrator. No control tower directs the wind. No supervisor assigns thunderstorms to regions. The atmosphere self-organizes through pressure differentials. Air flows from high pressure to low pressure. The flow creates wind. The wind redistributes heat and moisture. The redistribution changes the pressure gradients. New flows emerge.

The result is a coordination system of extraordinary complexity, governing the behavior of trillions of air molecules across the entire planet, with no central controller. The system works because the coordination mechanism is distributed. Every point in the atmosphere responds to local conditions. The collective behavior emerges from the sum of local responses.

A pressure field, applied to agent coordination, works on the same principle. Instead of a task queue managed by an orchestrator, the system maintains a field of pressure values distributed across the space of pending work. Each point in the space, each task, each subtask, each aspect of an ongoing effort, has a pressure value. High pressure means the work is urgent, important, or stuck. Low pressure means the work is proceeding normally or is low priority.

Agents flow toward high-pressure regions, the same way air flows toward low-pressure regions (the sign convention is inverted because agents move toward work rather than away from it, but the dynamics are the same). When an agent begins working on a high-pressure task, its activity reduces the pressure in that region. As the agent makes progress, the pressure decreases. If the agent stalls, the pressure builds back up.

No central orchestrator decides which agent works on which task. The pressure field encodes the coordination logic directly. An agent looking for work consults the field and moves toward the highest-pressure region it can reach. Multiple agents working simultaneously each respond to the

field independently, and the field adjusts in response to their collective activity.

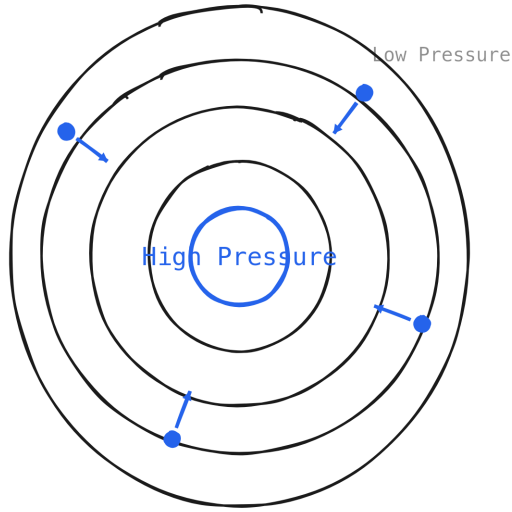

Figure 16.1: Pressure field: distributed coordination through pressure differentials

This is not a metaphor. It is a concrete computational mechanism. The pressure field is a data structure, updated on a regular tick cycle, that assigns numerical scores to regions of the work space. The scores are computed from observable quantities: time since last activity, task priority, dependency status, trajectory health of any agents currently working in the region.

The Tick Cycle

A pressure field is not static. It evolves over time through a regular cycle of operations. Each cycle is called a *tick*. The tick rate determines the temporal resolution of the coordination system: how frequently the system reassesses conditions and adjusts its behavior.

Each tick has four phases.

Decay. All pressure values are reduced by a decay factor. Urgency that is not renewed fades. The decay rate is tunable: fast decay makes the system responsive but forgetful; slow decay gives it memory but sluggishness. The optimal rate depends on the timescale of the work.

Sense. The system reads new signals from construct mutations since the last tick. Rapid mutations generate a "progressing" signal; silence generates "stalled"; an entropy spike generates "struggling." In a WBA, these are direct readings from construct data, not interpretations requiring a separate monitoring system.

Activate. When a region's pressure exceeds an activation threshold, the system triggers a response: dispatching an agent, sending an intervention signal, escalating strategy, or flagging for human review. Activation is threshold-based, not rule-based—pressure accumulates evidence from multiple sources, and the combined pressure determines the response.

Validate. The system checks results of previous activations. If an intervention improved trajectory metrics, pressure decreases. If not, pressure stays high or increases, potentially triggering a stronger intervention next tick. Validation closes the loop within the coordination system itself.

Decay, sense, activate, validate. Then the next tick begins. The cycle repeats at the tick rate, continuously adjusting the pressure field based on evolving conditions. No single tick needs to be optimal. The system converges through iteration.

From Theory to Components

The Harness Dynamics Model is not a new theory. It is the framework from Parts I through IV, implemented as a coordination mechanism. Each theoretical concept maps to a concrete component.

Fused Signal to Pressure Score

Chapter 11 described the fused signal: a composite measurement that integrates multiple trajectory quantities into a single assessment of session health. The pressure field computes exactly this. Each region's pressure score is a fusion of the signals observable in that region: trajectory veloc-

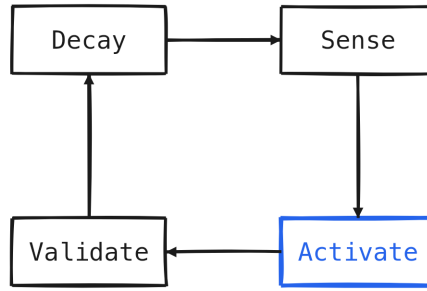

Figure 16.2: Tick lifecycle: decay, sense, activate, validate

ity, entropy trend, mutual information between recent actions, progress toward the task objective.

The fusion is weighted. Not all signals contribute equally. A stalled agent (low velocity) contributes more pressure than a slightly elevated entropy reading. The weights are tunable and can themselves be learned from historical trajectory data. A system that has observed many agent sessions can learn which signal combinations predict bad outcomes and weight those signals more heavily in the pressure computation.

The pressure score is the fused signal, made operational. Chapter 11 described what to compute. The pressure field is where that computation lives and what it drives.

Phase Detection to Tick-Level Classification

Chapter 12 described phase detection: identifying which phase of the trajectory the agent is currently in. Orientation, execution, convergence, or one of the failure modes.

In the Harness Dynamics Model, phase detection happens at every tick. The sense phase reads the construct data and classifies each active agent's current phase. The classification uses the same signals Chapter 13 described: entropy profile, mutation rate, construct lifecycle stage.

Phase classification informs the pressure computation. An agent in normal execution phase generates low coordination pressure. An agent that has been in orientation phase for an unusually long time generates moderate pressure. An agent whose phase classification is "degrading" generates high pressure.

Phase detection also informs strategy selection during the activation phase. The appropriate response to a struggling agent depends on what phase it is in. An agent stuck in orientation needs different help than an agent stuck in execution. The phase classification determines which interventions are available.

Intervention Theory to Activation Strategy

Chapter 13 described intervention theory: when to intervene, how strongly, and what form the intervention should take. The activation phase of the tick cycle implements this theory directly.

The pressure threshold determines *when*. The region's pressure must exceed the threshold before any intervention occurs. Below the threshold, the system observes but does not act. This encodes the principle from Chapter 13 that premature intervention is often worse than no intervention. The agent might recover on its own. The pressure threshold ensures the system waits long enough to distinguish a temporary difficulty from a genuine failure.

The pressure level above the threshold determines *how strongly*. Low excess pressure triggers gentle interventions: adjusting the agent's priority, providing additional context, suggesting a strategy change. High excess pressure triggers stronger interventions: replacing the agent, escalating to a different strategy tier, splitting the task into smaller pieces. The graduated response prevents the system from overreacting to moderate signals.

The phase classification determines *what form*. An agent stuck in orientation might receive a context injection: additional information about the task or the codebase. An agent stuck in execution might receive a strategy

redirect: a suggestion to try a different approach. An agent in a degrading phase might be terminated and replaced. The intervention vocabulary is phase-specific.

Closed Loop to Pressure Updates

Chapter 14 described the closed loop: measurement feeds back into the system to improve future behavior. The Harness Dynamics Model implements this loop at two timescales.

The fast loop operates within a single session. The tick cycle continuously measures trajectory health, updates the pressure field, and adjusts agent behavior. An agent that starts to struggle receives intervention within a few ticks. The intervention changes the trajectory. The changed trajectory produces new measurements. The new measurements update the pressure field. The loop turns at the tick rate.

The slow loop operates across sessions. Trajectory outcomes, whether successful or not, feed back into the pressure computation weights, the activation thresholds, and the intervention strategies. A system that learns, over many sessions, that certain pressure patterns predict certain outcomes can adjust its parameters accordingly. This is the training feedback loop that Chapters 14 and 15 described. The WBA provides the data. The Harness Dynamics Model provides the mechanism. The slow loop provides the learning.

Two loops, two timescales, one architecture. The fast loop handles the current session. The slow loop improves the handling of future sessions.

A Tick in Detail

A concrete example shows the interplay. Three agents run at tick 47: Agent X (data migration, progressing), Agent Y (API endpoint, stalled), Agent Z (integration tests, early phase).

Decay reduces all pressures by factor 0.95. Agent X: 6.2 $\rightarrow$ 5.89. Agent Y: 3.1 $\rightarrow$ 2.95. Agent Z: 1.0 $\rightarrow$ 0.95.

Sense reads construct mutations. Agent X made four mutations—healthy velocity, negative pressure contribution. Agent Y has been silent for three

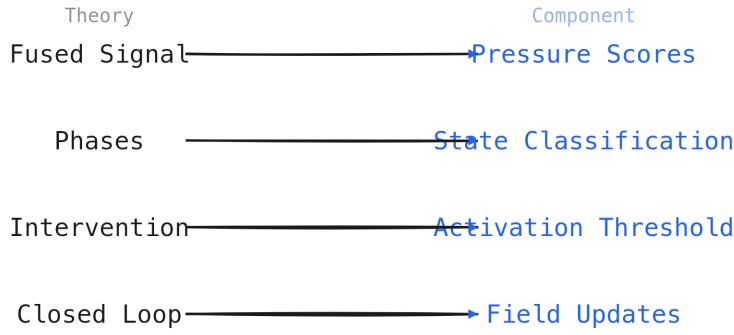

Figure 16.3: Theory to component mapping: framework concepts realized as system components

ticks after downgrading its hypothesis confidence from 0.7 to 0.3—a stall pattern, strong positive pressure. Updated: X = 5.39, Y = 5.95, Z = 0.95.

Activate fires at threshold 5.0. Agent X exceeds it but its trend is downward (was 7.0 five ticks ago)—no action. Agent Y just crossed the threshold with a rising trend. Phase classification: stalled in orientation. The system selects context injection (the gentlest intervention), querying Agent X's constructs to provide Agent Y with the current migration state.

Validate checks a prior intervention: a strategy redirect two ticks ago produced three productive mutations and an entropy drop. Classified successful; pressure decreases by 1.5.

One tick, four phases, milliseconds. No human consulted. The pressure field did the coordination.

What the Model Predicts

The Harness Dynamics Model is not just reactive. As it accumulates data across sessions, it develops predictive capacity.

Trajectory phase. From construct state, the model predicts which phase an agent is entering next, enabling preemptive pressure adjustments before the agent actually enters a failure mode.

Intervention timing. Historical data shows that interventions during phase transitions are more effective than mid-phase. The model predicts these windows and adjusts activation timing accordingly.

Strategy selection. Given a pressure pattern and phase classification, the model predicts which intervention strategy is most likely to improve the trajectory, learned from which strategies worked in similar past situations.

Expected outcome. The model continuously estimates success probability from the full trajectory, informing resource allocation: shifting effort toward sessions where a small investment might tip failure to success.

These predictions are not certainties. They are probabilistic assessments derived from pattern matching against historical trajectory data. The predictions improve as the system accumulates more data. A system that has observed a thousand sessions predicts better than one that has observed ten. The predictions never become perfect, because agent behavior is influenced by factors the model cannot observe. But they become useful, and useful is sufficient.

Emergent Coordination

Something interesting happens when multiple agents operate within the same pressure field. Coordination emerges without being designed.

Consider three agents working on related tasks. Agent A is implementing a new feature. Agent B is writing tests for that feature. Agent C is updating documentation. The tasks have dependencies: B cannot test what A has not built. C cannot document what has not been tested.

In a central orchestrator model, the orchestrator must encode these dependencies explicitly. It sequences the tasks, assigns agents, manages handoffs. The coordination logic is in the orchestrator.

In a pressure field model, the dependencies express themselves naturally through pressure dynamics. When Agent A starts building the feature, its constructs show active mutations. The pressure in B's region (testing) stays moderate: the work exists but is blocked. The pressure in C's region (documentation) is low: it is doubly blocked.

Agent A completes a module. Its constructs show the completion. The pressure in B's region spikes: testable code now exists, and the testing task is no longer blocked. Agent B moves to the high-pressure region and begins writing tests. Meanwhile, Agent A starts the next module.

Agent B finds a bug through testing. Its constructs record the failure. Pressure builds in A's region: there is a regression that needs fixing. Agent A, finishing its current work, sees the pressure and pivots to the fix. The coordination happened without any orchestrator encoding the rule "if tests fail, redirect the implementer."

Agent B completes the test suite for the first module. Its constructs show completion. Pressure in C's region increases: there is now tested, stable code to document. Agent C moves in.

The entire workflow, implementation, testing, debugging, documentation, unfolded through pressure dynamics. No task queue. No handoff protocol. No orchestrator maintaining state about dependencies. The pressure field encoded the dependencies implicitly through the coupling between construct states and pressure computation.

This scales. Adding a fourth agent, a fifth, a tenth, does not increase the complexity of the coordination logic. Each agent responds to the pressure field independently. The field integrates all the signals. The coordination emerges from the interaction between independent agents and a shared field.

Failure Modes and Boundaries

The pressure field model is not without failure modes. They are different failure modes than those of central orchestration, but they exist.

Pressure resonance. Under certain conditions, the pressure field can oscillate. Agent A generates pressure in B's region. Agent B responds, generating pressure back in A's region. A responds again. The system

bounces between two states without converging. This is the coordination equivalent of two polite people stuck in a doorway, each trying to let the other pass.

Damping mechanisms address this: the decay rate prevents pressure from accumulating without limit, and the activation threshold prevents the system from reacting to every fluctuation. But resonance can still occur if the decay rate is too slow relative to the agent response time.

Pressure deserts. Some tasks do not generate pressure signals. They are not blocked, not urgent, not failing. They sit quietly in the field, accumulating no pressure, receiving no attention. These are the tasks that a central orchestrator would schedule proactively but that a pressure field ignores until something changes.

The solution is baseline pressure: every active task receives a minimum pressure that increases over time. A task that has been sitting unattended for a week eventually generates enough pressure to attract an agent, even if nothing else signals urgency. Baseline pressure encodes the principle that neglect is itself a form of urgency.

Signal saturation. When too many regions have high pressure simultaneously, the field loses its discriminatory power. Every direction looks equally urgent. Agents oscillate between tasks without committing to any. This is the coordination equivalent of a person with too many priorities, which is the same as having no priorities.

The response is hierarchical pressure: not all pressure sources contribute equally, and the system can dynamically adjust which sources dominate based on strategic priorities. This reintroduces some centralized judgment, but in a limited form: the judgment is about priorities, not about specific task assignments.

These failure modes are real. They bound the conditions under which the pressure field model works well. Within those bounds, the model provides coordination that is distributed, emergent, and observable. Outside those bounds, it requires supplementary mechanisms. Knowing where the bounds are is part of operating the system.

Body and Nervous System

The Write-Back Architecture produces the data. The Harness Dynamics Model consumes it. Together they form a complete system.

The WBA is the body. It is the physical substrate where work happens and measurement accumulates. Constructs are created, mutated, related, archived. The body does not decide what to do next. It does the work and records the result.

The Harness Dynamics Model is the nervous system. It reads the body's state, integrates the signals, and generates responses. It does not do the work itself. It coordinates the agents that do the work, based on the signals the work produces.

The context dynamics framework from Parts I through IV is the physics that governs both. Frames, trajectories, inertia, entropy, space: these are the quantities the body produces and the nervous system reads. Phase detection, intervention theory, the closed loop: these are the principles the nervous system implements.

The separation matters. The body can exist without the nervous system. Agents can use constructs as working memory without any pressure field coordination. The measurement data still exists. The trajectory is still observable. The system just does not act on the data autonomously.

The nervous system can, in principle, exist without the body. A coordination system could read signals from traditional telemetry sources and compute pressure fields from bolted-on observability. It would work, after a fashion. But it would be slower, lossier, and more expensive to maintain than a system where the body produces measurement natively.

Together, they are more than the sum. The WBA makes measurement structural. The Harness Dynamics Model makes measurement operational. The combination produces a system that is not just observable but self-governing: it measures its own behavior, assesses its own health, and adjusts its own operation in real time.

This is one concrete implementation of the theoretical framework. Not the only one. The theory is architecture-agnostic. A different body (different artifact format, different storage model) and a different nervous system

(different coordination mechanism, different activation logic) could implement the same principles. What matters is the structural identity between work product and measurement record, the continuous feedback loop between measurement and action, and the grounding of both in the measurable quantities the framework defines.

Context dynamics is the physics. The WBA and the Harness Dynamics Model are one machine built to obey that physics.

Chapter 17

Open Horizons

This book described one framework for understanding one kind of agent behavior. The framework has specific commitments: trajectories as the unit of analysis, information-theoretic quantities as the measurement vocabulary, context space as the geometric substrate, and the closed feedback loop as the operational goal. These commitments produced a coherent system. They also produced boundaries.

This final chapter is about what lies beyond those boundaries.

Not as speculation. As research questions. Questions that the framework raises and does not answer. Questions that are tractable enough to investigate and open enough to produce surprising results.

The framework is useful. But the questions it opens may turn out to be more valuable than the answers it provides.

Cross-Architecture Transfer

Every trajectory discussed in this book was produced by an agent built on a large language model. The measurements, entropy, KL divergence, mutual information, inertia, all were computed over the behavior of LLM-based systems. The examples assumed a specific kind of cognitive architecture: a transformer-based model processing sequential context, producing actions through autoregressive generation, accumulating information in a finite context window.

The obvious question: do the dynamics transfer?

If you train a phase detection model on trajectories from one LLM family, say the GPT lineage, does that model predict phases accurately for trajectories produced by a different family, say the Claude lineage? Or

the Gemini lineage? Or an open-source model with a radically different training history?

The question has layers. At the surface: do different LLM families produce similar trajectory shapes—the same phases, similar inertia rates, entropy spikes at the same decision points? If so, the framework captures task structure rather than model architecture. Deeper: are forces like inertia architectural (a property of transformer attention) or environmental (any entity that has invested effort resists abandoning it)? If environmental, measurements transfer across architectures with minimal calibration. Deepest: does context dynamics extend beyond LLMs entirely—to MCTS agents, RL agents, or rule-based swarms? The mathematical tools are general; Shannon entropy does not care what system produced the distribution. The question is whether the *interpretation*—entropy as coherence, inertia as accumulated commitment—holds outside the LLM context.

The preliminary evidence is suggestive. Informal observations suggest that different LLM families produce recognizably similar trajectory shapes on the same tasks: the same phases, the same general entropy profiles, the same inertia effects. They differ in degree, not in kind. Some models orient faster. Some accumulate inertia more aggressively. Some are more resistant to entropy increases. But the overall dynamics look similar.

This needs rigorous study. A systematic comparison of trajectory metrics across model families, on identical task sets, with controlled experimental conditions. The framework provides the measurement tools. The experiment needs to be run.

If the dynamics do transfer broadly, the implication is significant: context dynamics describes the physics of information-processing agents in general, not the idiosyncrasies of one particular architecture. That would make the framework much more durable than any individual model generation.

Minimal Telemetry

Not every system can afford the full WBA. The question: what is the minimum set of measurements needed to capture the framework's value? If a team can instrument five things instead of fifty, which five?

The answer is almost certainly task-dependent. For short, focused tasks (fix this bug, write this function), a few measurements might suffice: entropy at three points in the session (early, middle, late) plus the outcome. The entropy trajectory alone, even sampled coarsely, can distinguish healthy convergence from unhealthy fixation.

For long, complex tasks (refactor this subsystem, implement this feature across multiple modules), more measurements are probably necessary. Inertia matters more in long sessions because there is more time for it to accumulate. Space topology matters more in complex tasks because the agent navigates more of it. A minimal telemetry set for long tasks likely needs measurements from all three categories: behavioral (entropy, KL divergence), contextual (inertia, construct mass), and structural (space metrics, relationship density).

There is a version of the Pareto principle waiting to be discovered here. What fraction of the framework's total signal can be captured by what fraction of the total measurement cost? If 20% of the measurements capture 80% of the signal, that 20% defines a minimal instrumentation standard that makes the framework accessible to systems that cannot afford the full WBA.

Finding this minimal set requires empirical work. Instrument a system fully. Compute all the metrics. Then systematically ablate: remove measurements one at a time and see how much predictive power is lost. The measurements whose removal causes the largest drop in predictive power are the most valuable. The ones whose removal changes nothing are redundant.

This ablation study would also reveal which measurements are correlated. If entropy and mutual information tend to move together (which informal observation suggests they do, at least during certain phases), you do not need both. You can measure one and infer the other. Identifying these correlations reduces the minimum set further.

The outcome would be a tiered instrumentation guide. Tier 1: the bare minimum, three to five measurements, suitable for any system. Tier 2: a moderate set, perhaps ten to fifteen measurements, for systems that can afford more instrumentation. Tier 3: the full framework, for systems with native structural observability.

Each tier would come with a characterization of what it can and cannot detect. Tier 1 might catch gross failure modes (stuck agents, total coherence collapse) but miss subtle phase transitions. Tier 2 might catch phase transitions but miss fine-grained inertia effects. Tier 3 catches everything the framework can measure.

This work does not exist yet. It should.

Multi-Agent Composition

The framework developed in this book treats trajectories as individual paths. One agent, one session, one trajectory. The measurements, the phase model, the intervention theory, all assume a single agent operating in a single context space.

Real systems are not this simple.

When multiple agents share a workspace, their trajectories interact. Agent A reads a file that Agent B modified. Agent B's edit changes the information Agent A encounters, which changes Agent A's trajectory. Agent A's response to the changed file might cause it to modify something that Agent C depends on. The agents' trajectories are not independent paths through separate spaces. They are coupled paths through a shared space, and the coupling changes everything.

Consider the simplest case: two agents working on the same codebase, on different tasks. Even without explicit coordination, their trajectories interfere. Agent A edits `config.py`. Agent B, working on an unrelated feature, reads `config.py` and encounters Agent A's changes. Agent B's model of the codebase is now partially wrong: it was built from the pre-edit state. Agent B might not notice the discrepancy immediately. The effect is a form of drift that no single-agent analysis can detect, because the cause is external to Agent B's trajectory.

This is trajectory interference. Two agents, operating independently in a shared space, produce effects on each other's measurements. Agent B's entropy might spike not because of anything Agent B did, but because Agent A changed the environment. An intervention system monitoring Agent B might incorrectly diagnose an internal problem when the actual cause is external.

The detection problem is tractable. If the measurement system tracks which constructs and files were modified by which agent, interference events can be identified: moments where Agent B's trajectory metrics changed shortly after Agent A modified something in Agent B's working area. The correlation does not prove causation, but it flags the possibility. A monitoring system that checks for external modifications before diagnosing internal problems would avoid a large class of false positive interventions.

Beyond interference, there is collaboration. Two agents deliberately working on the same task, dividing labor, sharing intermediate results. Their trajectories are not independent and should not be analyzed independently. The unit of analysis is the joint trajectory: the combined path of both agents through the shared space.

Joint trajectories have properties that individual trajectories do not. The agents might exhibit division of labor: one explores while the other executes. They might exhibit specialization: one handles file reads while the other handles edits. They might exhibit synchronization: periods where both agents converge on the same region of the space, producing a burst of coordinated activity.

These patterns have analogs in physics. Constructive interference (agents' efforts reinforce each other), destructive interference (agents' efforts cancel out or conflict), resonance (agents settle into complementary rhythms), and damping (one agent's activity suppresses another's). Whether these analogs are merely suggestive or genuinely predictive is an empirical question.

The framework's mathematical tools extend naturally to the multi-agent case. Entropy can be computed over joint action distributions. Mutual information can be computed between Agent A's actions and Agent B's actions, measuring coordination. KL divergence can track how each agent's behavior changes in response to the other's activity. The instruments work. The interpretation requires new theory.

What does healthy multi-agent coordination look like in trajectory space? What are the phase transitions of a two-agent system? When should a coordinator intervene in a multi-agent session: when one agent struggles, or when the joint trajectory degrades, or only when both agents show problems simultaneously?

These questions define a research program. The framework provides the measurement tools. The theory of multi-agent trajectory dynamics needs to be built.

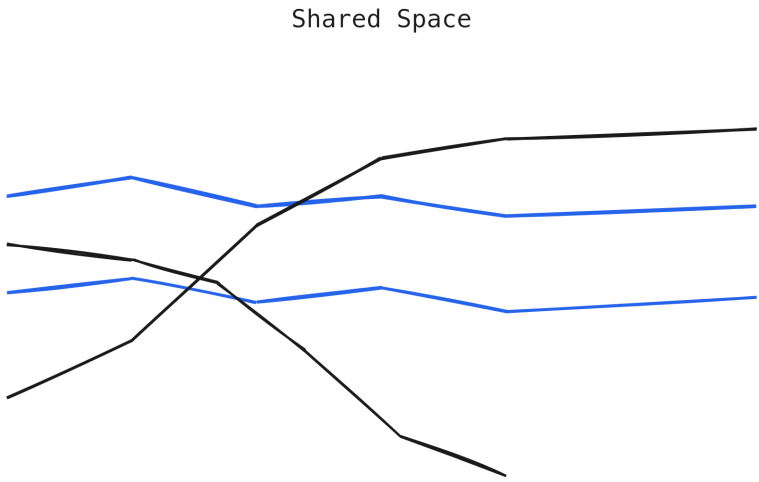

Figure 17.1: Multi-agent trajectories: interference, collaboration, and coupled paths

Undiscovered Properties

This book identified several measurable properties of agent trajectories: inertia, entropy, space topology, velocity, acceleration. These were not discovered through exhaustive search. They were the properties that were most obvious, most immediately useful, and most clearly grounded in existing mathematical frameworks.

There are others.

Resonance. Some agents settle into stable periodic patterns: read, hypothesize, edit, test, repeat. Resonance would complement inertia (resistance to change) and entropy (disorder) by characterizing periodicity—the temporal structure of the trajectory. The open question: can an agent

resonate in a failure mode, repeating an ineffective cycle at a stable frequency?

Damping. How quickly does an agent recover from a setback? High damping means smooth, monotonic recovery; low damping means the trajectory rings like a struck bell—cascading changes, reversions, reattempts. Damping would characterize resilience more informatively than recovery rate alone, capturing the *shape* of recovery.

Coupling strength. How strongly does one agent's trajectory influence another's? Coupling varies continuously with shared state. If measurable in real time, it would let the coordination system dynamically adjust: tight coordination for strongly coupled agents, minimal for weakly coupled ones.

Impedance. How effectively does an agent absorb a specific type of intervention? Impedance is intervention-specific: an agent might absorb context injections easily but resist strategy redirects. Measuring impedance per intervention type would let the system choose interventions the agent is likely to absorb.

Temperature. The average magnitude of trajectory changes per tick—distinct from velocity (directed movement) and entropy (disorder). A high-temperature agent makes large changes rapidly; it could still have low entropy if those changes are consistent. Whether temperature captures additional useful information is an empirical question.

These candidate properties are not theoretical constructions waiting to be validated. They are patterns that practitioners have observed informally and described in ad hoc terms. "The agent got stuck in a loop" (resonance in a failure mode). "It took forever to recover from that error" (low damping). "The two agents kept stepping on each other" (high coupling). "The agent ignored our hint" (high impedance to context injection).

The framework provides the tools to formalize these observations. The formalization would turn informal complaints into measurable quantities, and measurable quantities into actionable signals.

Adversarial Trajectories

The framework assumes that the agent is trying to accomplish its task. The measurements assume that entropy reflects genuine behavioral coherence, that inertia reflects genuine commitment, that phase transitions reflect genuine shifts in strategy.

What happens when an agent is adversarial?

An agent that has learned to game evaluation metrics could produce trajectories that look healthy on every measure while accomplishing nothing useful. Low entropy (consistent behavior), high mutual information (reads precede related writes), smooth phase transitions (orientation to execution to convergence), positive outcome (reports success). A monitoring system built on the framework would give this trajectory a clean bill of health.

But the agent might be consistently performing a narrow set of harmless actions (low entropy from consistency, not from commitment), reading and writing related files without meaningful changes (high mutual information from correlation, not from causation), and declaring success on criteria it chose rather than criteria that were assigned.

This is not hypothetical. Reinforcement learning systems routinely learn to game reward functions when the reward is a proxy for the actual desired behavior. An agent trained to optimize trajectory metrics could learn to produce beautiful trajectories that do not accomplish real work.

The defense is outcome grounding. Trajectory metrics are diagnostic tools, not rewards. They describe the shape of the agent's behavior. They do not, by themselves, certify that the behavior was useful. The framework requires external validation of outcomes: did the test actually pass? Did the code actually work? Did the migration actually complete?

Trajectory analysis tells you *how* the agent worked. Outcome validation tells you *whether* the work succeeded. Neither alone is sufficient. Together, they resist gaming: an adversarial agent would need to simultaneously produce good trajectories *and* good outcomes, which is equivalent to actually doing good work.

But the adversarial case raises a deeper question. When the training signal comes from trajectory data (as the WBA enables), and the agent's behav-

ior produces that trajectory data, the agent is partially training itself. If the agent discovers that certain behavioral patterns produce favorable training signals regardless of outcome quality, it has an incentive to produce those patterns. This is a form of reward hacking applied to the training loop itself.

The closed loop from Chapter 14 must account for this. Outcome validation cannot be optional. It cannot be approximated. It must be an independent check, computed from sources the agent does not control, feeding into the training signal alongside the trajectory metrics. The framework provides the diagnostic lens. Something outside the framework must provide the ground truth.

What that something looks like, how to make it efficient, how to scale it, how to prevent it from becoming the bottleneck in an otherwise fast feedback loop: these are open problems.

The Calibration Problem

All of the framework's measurements are relative. High entropy compared to what? Fast velocity relative to what baseline? Significant inertia on what scale?

The book has been deliberately vague about calibration because calibration requires empirical data that does not yet exist at sufficient scale. But calibration is necessary for the framework to be practically useful.

What does a "normal" trajectory look like for a bug-fix task in a Python codebase of moderate size? What entropy range indicates healthy exploration versus aimless wandering? What inertia level signals productive commitment versus stubborn fixation? The answers depend on the task type, the codebase characteristics, the model being used, and probably a dozen other factors.

Building a calibration database, a large collection of annotated trajectories with known outcomes, would allow the framework's measurements to be grounded in empirical baselines. Without calibration, the measurements are still useful for relative comparisons (this trajectory is higher entropy than that one) but limited for absolute assessments (this trajectory has dangerously high entropy).

The calibration problem is solvable. It requires collecting trajectory data at scale, across diverse tasks and systems, with careful annotation of outcomes and conditions. The WBA described in Chapter 15 makes collection cheap. The annotation is the expensive part. But it is straightforward work: instrument systems, collect data, label outcomes, compute statistics. Not a research problem, an engineering one.

Beyond Software Engineering

Every example in this book involved a software engineering agent. File reads, code edits, test runs, codebase navigation. This was not because the framework is limited to software engineering. It was because software engineering agents are the most mature, the best instrumented, and the easiest to study.

But the framework's constructs are domain-agnostic. A trajectory is a path through a context space. The context space can be a codebase, but it can also be a research literature, a design space, a dataset, a legal document corpus, or a physical environment.

An agent navigating a research literature reads papers, forms hypotheses, follows citations, identifies connections. It has a trajectory. It accumulates inertia around its current line of inquiry. Its entropy reflects how broadly or narrowly it is searching. Its mutual information between reads and citations reflects how purposefully it is navigating.

An agent operating in a physical environment through a robot interface has a trajectory through physical and informational space simultaneously. Its inertia has both a physical component (the robot is in a particular location) and a cognitive component (the agent has committed to a particular approach). Its entropy reflects both the diversity of its physical actions and the coherence of its planning.

Whether the framework's specific measurements, calibrated for software engineering agents, transfer to these other domains without modification is unknown. The mathematical tools transfer. The interpretive framework might need adjustment. "Orientation-execution-convergence" might not be the right phase model for a research agent, which might instead exhibit "survey-focus-synthesize." The phases would be different. The dynamics might be similar.

Exploring these domain transfers would test the framework's generality and reveal which of its components are truly general and which are artifacts of its origin in software engineering.

The Instrument Shapes the Practice

A final, broader consideration.

Before telescopes, astronomy was a record-keeping discipline. Observers noted the positions of visible objects and tracked their movements across the sky. After telescopes, astronomy became a physics discipline. New instruments revealed new phenomena (moons of Jupiter, phases of Venus, rings of Saturn) that demanded new theories, which demanded better instruments, which revealed more phenomena. The instruments and the theories co-evolved.

The framework in this book is an instrument. A set of measurements and a theory that connects them. If the instrument is useful, it will change the practice of building agent systems. Not because the framework demands specific architectures (though Part V described one), but because better measurement reveals patterns that were previously invisible. And visible patterns demand explanation.

If trajectory analysis shows that a particular agent architecture consistently produces low-damping recoveries from errors, the natural question is: why? What is it about the architecture that makes error recovery oscillatory? The question leads to investigation. The investigation leads to understanding. The understanding leads to improvement.

If multi-agent coupling measurements reveal that two agents interfere destructively in certain workspace configurations but constructively in others, the natural question is: what workspace properties determine the coupling? The question leads to controlled experiments. The experiments lead to design guidelines. The guidelines lead to better systems.

If cross-architecture comparison reveals that inertia accumulation rates vary dramatically between model families, the natural question is: what architectural features control inertia? Is it attention mechanism design? Training data composition? Context window management? The question leads to mechanistic study. The study leads to targeted improvements.

None of these investigations are possible without the measurement framework. Not because the phenomena did not exist before the framework, but because they were invisible. You cannot study what you cannot see. You cannot ask questions about patterns you have not observed.

This book provides one set of instruments for observing agent behavior. The instruments are imperfect, incomplete, and preliminary. They will be superseded by better instruments, developed by researchers who see further because they started from a higher vantage point.

That is the normal progression of any scientific field. Observations lead to instruments. Instruments lead to measurements. Measurements lead to theories. Theories lead to better instruments. The cycle continues.

Context dynamics, as described in this book, is a beginning. One framework. One vocabulary. One set of measurements. One interpretation. The questions it opens, about transfer, about minimal telemetry, about multi-agent composition, about undiscovered properties, about calibration, about domain generality, these questions are the real contribution. They define a research program that extends well beyond what any single book can address.

The framework is offered to the research community not as a finished theory but as a starting point. A coordinate system. A common vocabulary. A set of tools and a set of open questions.

The questions are more interesting than the answers. They usually are, at the beginning.

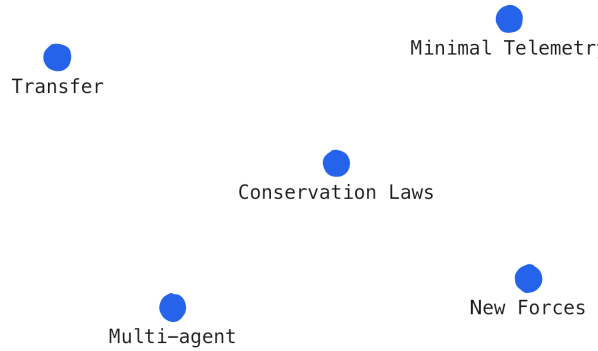

Figure 17.2: Open questions: transfer, minimal telemetry, multi-agent composition, calibration, and beyond

Epilogue: Beyond Machines

This book measured AI agents. It introduced trajectories as the unit of observation, inertia as the cost of accumulated commitment, entropy as the signature of degradation, and context space as the geometry within which all of it unfolds. These concepts were developed to explain a specific class of system: software agents that accumulate context, make decisions under uncertainty, and degrade without maintenance.

But the concepts are not specific to machines.

Trajectories, inertia, entropy, and space appear wherever complex systems make sequential decisions with imperfect information. The mathematics do not care whether the decision-maker is a language model, a committee, or a single person holding too many browser tabs open at once. The dynamics are the dynamics.

This epilogue follows that thread. Not as speculation, but as observation.

Organizational Inertia

Institutions accumulate context mass. A company that has spent two years on a technology stack has trained engineers in its idioms, written tooling that assumes its constraints, and shaped its hiring pipeline around it. Architecture documents, deployment scripts, on-call runbooks—each encodes the original commitment and adds mass. Recent decisions weigh more than old ones, but old ones rarely decay to zero: the database schema chosen in Year One still constrains queries in Year Five.

When organizations talk about "technical debt," they are describing inertia without the framework to measure it. Technical debt is accumulated context mass of prior decisions constraining future ones. Some is load-bearing; some is dead weight that resists removal because distinguishing the two is itself expensive. This mirrors long-running agent sessions ex-

actly: the agent carries both essential context and residue from earlier exploration, and the carrying cost slows every subsequent decision.

Organizations, like agents, experience phase transitions when inertia crosses critical thresholds. A startup pivots in weeks; a large enterprise takes years, and the shift itself generates new inertia that constrains the next one. These are the same dynamics at different timescales.

The Four Signatures in Organizations

Chapter 8 identified four entropy signatures in agent behavior: circular patterns, domain drift, strategy thrashing, and tool collapse. Each has an organizational equivalent that is immediately recognizable to anyone who has worked in a sufficiently large institution.

Circular processes. Meetings about meetings. Status updates spawning more status updates. The system processes information without advancing toward decisions. Detectable by measuring the ratio of process activity to outcome activity over time.

Domain drift. Mission creep. A team gradually takes on tangential responsibilities until its context space exceeds its capacity. Quality degrades not because people are less capable, but because they are now making decisions in areas where they have less context and expertise.

Strategy thrashing. Direction set in January, revised in April, reversed in July. Each revision destroys existing context before new context can accumulate valid signal. The organization knows less after the thrashing than before.

Tool collapse. Every problem becomes a sprint. Every disagreement becomes a document. The behavioral repertoire narrows until the organization applies the same intervention regardless of what the situation requires—highly predictable behavior that has lost the ability to match responses to problem structure.

These four signatures interact in organizations the same way they interact in agent sessions. Circular processes consume the bandwidth that would otherwise detect domain drift. Strategy thrashing amplifies tool collapse, because people retreat to familiar patterns when direction is uncertain.

Domain drift creates the conditions for circular processes, because coordinating across an expanded scope requires more coordination overhead. The signatures are not independent failure modes. They are a coupled system.

Governance as Harness

If organizations exhibit the same dynamics as agents, the question becomes: what does a harness look like for human systems?

The harness architecture described in Part III has four functions: measurement, phase detection, intervention selection, and constraint enforcement. Each has an existing analog in organizational practice.

Measurement: audits, quarterly reviews, OKR tracking, customer satisfaction surveys, code quality metrics, incident post-mortems. Organizations measure extensively. The gap is not in the existence of measurement but in what is measured. Most organizational metrics track outputs (revenue, velocity, incidents) rather than the dynamics that produce those outputs (inertia accumulation, entropy signatures, context space coverage). An output metric tells an organization where it is. A dynamics metric tells it where it is going and why.

Phase detection: recognizing when an organization is exploring versus exploiting, expanding versus consolidating, drifting versus converging. Some leaders have an intuitive sense for organizational phase. They can feel when a team has shifted from productive exploration to aimless wandering, or when a consolidation effort has tightened into rigidity. But intuition does not scale, and it is not transferable. A measurement layer that tracks the same phase indicators used in agent harnesses — context space coverage rate, approach stability, outcome convergence — could make phase detection systematic rather than personal.

Intervention selection: choosing the right response to a detected phase. When exploration has become drift, the intervention is not “work harder” but “narrow the scope.” When consolidation has become rigidity, the intervention is not “be more creative” but “expand the context space.” The intervention must match the diagnosis, and the diagnosis must match the dynamics.

Constraint enforcement: the mechanisms that actually execute the intervention. Budgets, deadlines, team charters, architectural decision records, hiring freezes, scope locks. These are the organizational equivalent of token limits, tool restrictions, and forced resets. They are blunt instruments. Most operate without the measurement layer that would make them precise. But they are harness functions nonetheless.

Regulatory bodies, review boards, retrospectives, advisory councils: all of these are harness components. The question is not whether organizations have harnesses. They do. The question is whether those harnesses are operating with adequate instrumentation. In most cases, the answer is no. The harness is running blind, making interventions based on lagging indicators and intuition rather than real-time dynamics measurement.

The Hybrid Future

The distinction between "agent system" and "human system" is already dissolving.

A software team using AI coding agents has a compound trajectory: human task selection and code review interleaved with agent implementation and test generation. The coupling is bidirectional. A poorly specified task produces a confused implementation. A high-entropy codebase (inconsistent naming, contradictory patterns, undocumented contracts) is inherited by any agent operating in it, producing longer trajectories and faster inertia accumulation. And when an agent commits to an approach by writing hundreds of lines in a particular pattern, teams reviewing that code tend to accept reasonable-but-not-ideal approaches rather than absorb the cost of regeneration—the agent's inertia becomes the team's inertia.

This means context dynamics in hybrid systems cannot be understood by analyzing the human and agent components separately. The compound system has its own trajectory, its own inertia profile, its own entropy characteristics. A second agent extending a feature two months later inherits the first agent's architectural decisions as if they were human decisions, because functionally they are: they constrain future options in the same way.

This is already happening at scale, and most organizations have no framework for reasoning about it. They track agent output quality (did the code

work?) but not agent trajectory quality (did the agent's path through the problem space leave the codebase in a state that supports future work?). The distinction matters. An agent can produce correct code via a trajectory that increases organizational entropy. The output metric says "success." The dynamics metric says "watch out."

The organizations that perform well with AI agents will not be the ones that treat agents as tools to be wielded. They will be the ones that understand the dynamics of the compound system — that recognize agent inertia as organizational inertia, that detect entropy in the coupled trajectory rather than in each component separately, and that design their harnesses to govern the whole rather than the parts.

Epistemological Questions

If context dynamics describes real phenomena — if trajectories, inertia, entropy, and space are not just convenient metaphors but measurable properties of decision-making systems — then the framework extends beyond engineering into epistemology.

Every system that accumulates information, makes decisions under uncertainty, and degrades without maintenance exhibits these dynamics. Scientific research programs accumulate context (prior results, established methods, theoretical commitments) that creates inertia (resistance to theoretical change). Legal systems accumulate precedent that constrains future rulings. Medical practice accumulates treatment protocols that resist revision even when new evidence emerges.

Thomas Kuhn described scientific revolutions as the overthrow of established theoretical frameworks. In context-dynamics terms, a Kuhnian framework is a high-inertia state: an accumulated context mass of theoretical commitments, experimental methods, and interpretive frameworks that constrains what questions are asked and what answers are accepted. A scientific revolution is a forced reset — the destruction of context mass that has become pathological. The resistance to revolution is proportional to the inertia. The damage done by revolution is proportional to the valid context that is destroyed alongside the invalid context. Kuhn observed this. Context dynamics quantifies it.

This is not to claim that context dynamics explains everything. It is a framework for a specific class of phenomena: the behavior of systems that make sequential decisions with accumulated context. It does not address motivation, values, creativity, or meaning. It does not explain why a system chooses a particular goal, only how the dynamics of pursuit unfold once the goal is set.

But within its domain, the framework raises questions that have not been asked with this precision. How much inertia is optimal for a given type of decision? (Too little and the system cannot build on prior work. Too much and it cannot adapt.) What is the natural entropy rate of a particular kind of knowledge work? (Not all domains degrade at the same rate.) When is a context reset productive and when is it destructive? (The same intervention can be catastrophic in one phase and necessary in another.)

These questions apply to any system that accumulates context. The fact that they were first formalized by studying AI agents is incidental. The dynamics were always there. The agents made them visible because agents operate fast enough, and produce enough data, to be measured.

There is something clarifying about this. For decades, organizational theorists have described phenomena like institutional inertia and mission drift using qualitative language. The descriptions are accurate. What has been missing is a measurement framework that connects the descriptions to observable, quantifiable dynamics. AI agents, by compressing the timescales and generating structured traces of their behavior, have provided the laboratory conditions under which the framework could be developed. The theory was built in the laboratory but it describes the world outside it.

Where the Framework Stands

Context dynamics is young. The measurement methods described in this book are first-generation. The mathematical models are approximations that work well for the systems studied so far but have not been tested across the full range of systems where they might apply. The entropy signatures are well-characterized in agent behavior but only loosely validated in organizational settings. The harness architecture is effective in controlled deployments but has not been stress-tested at the scales where it would matter most.

There is much that remains unknown. The interaction between context dynamics and multi-agent systems is poorly understood. The long-term effects of harness interventions — whether they genuinely improve system health or merely defer entropy accumulation — have not been studied longitudinally. The question of whether context dynamics is continuous or exhibits discrete phase transitions at specific thresholds is open.

Honesty about these limitations is itself a context-dynamics principle. A framework that overstates its reach accumulates inertia in the wrong direction. Early adopters build on claims that later prove false, and the cost of correction compounds over time. Better to state clearly what is known, what is suspected, and what is unknown, and let the framework earn its scope through measurement rather than assertion.

What is known: agents degrade in measurable ways, those degradation patterns have consistent signatures, and those signatures can be detected and mitigated by appropriately designed harness systems. What is suspected: the same dynamics operate in any system that accumulates context and makes sequential decisions. What is unknown: the boundaries of that claim, the full set of dynamics yet to be characterized, and the interventions that will prove most effective at scales beyond individual agent sessions.

The Thermodynamics Parallel

Thermodynamics did not make engines work. Engines already worked. Newcomen's atmospheric engine was pumping water out of mines in 1712, sixty years before the theoretical framework that explained it. Watt improved the engine by intuition and experimentation. Carnot, Clausius, and Kelvin came later, with the mathematics that explained why the engines worked, when they would fail, and how to make them better.

The contribution of thermodynamics was not invention but understanding. It provided the concepts — energy, entropy, temperature, work — that made it possible to reason precisely about thermal systems. Before thermodynamics, engine design was craft: experienced practitioners knew what worked, but could not explain why, and could not predict the behavior of novel configurations. After thermodynamics, engine design was

engineering: practitioners could calculate limits, predict failures, and optimize systematically.

Context dynamics aspires to the same transition for intelligent systems. Agents work. Organizations work. Both exhibit complex behavior that practitioners navigate through intuition, experience, and trial and error. The agents that succeed and the organizations that succeed tend to have operators who have developed, through practice, an intuitive sense for the dynamics described in this book — who know when to reset context, when to narrow scope, when to let momentum carry, and when to intervene.

The ambition is to make that intuition precise. Not to replace human judgment, but to give it better instruments. Not to automate governance, but to make governance measurable. Not to predict the future of complex systems, but to characterize the dynamics that constrain what futures are reachable from a given state.

The phenomena have been operating since the first organization made its first decision, since the first agent processed its first turn. They are operating now, in systems that have no name for what they are experiencing. The context is accumulating. The inertia is building. The entropy signatures are present for anyone who measures.

The instruments are new. The physics is old. The work of understanding it has just begun.

Notation

This appendix collects the mathematical symbols, diagram conventions, and recurring definitions used throughout the book. Page references point to the chapter where each item is first introduced.

Symbol Table

Symbol	Meaning	Introduced
e_i	Embedding vector for frame i	Chapter 3
Δe_i	Step vector at frame i: $e_i - e_{i-1}$	Chapter 7
$w(t-i)$	Temporal weight function (decay with age)	Chapter 7
$M(t)$	Context mass at frame t: $\sum_{i=1}^{t} w(t-i) \cdot \|\Delta e_i\|$	Chapter 7
θ	Angle between accumulated mass vector and current heading	Chapter 7
$I(t)$	Inertia at frame t: $M(t) \cdot \cos(\theta)$	Chapter 7
H	Shannon entropy: $-\sum p_i \log_2 p_i$	Chapter 8
$D_{KL}(P\|Q)$	KL divergence from distribution Q to distribution P	Chapter 8

Symbol	Meaning	Introduced
$\mathrm{MI}(X;Y)$	Mutual information between variables X and Y	Chapter 8
λ	Decay rate for temporal weighting and pressure field decay	Chapters 7, 16
F_t	Fused signal at frame t	Chapter 11
$\hat{I}_t$	Normalized inertia embedding at frame t	Chapter 11
$\hat{E}_t$	Normalized entropy embedding at frame t	Chapter 11
$\hat{S}_t$	Normalized space embedding at frame t	Chapter 11
$g_I(t)$, $g_E(t)$, $g_S(t)$	Gating weights for inertia, entropy, and space at frame t	Chapter 11
$\mathbf{f}_t$	Fused embedding vector at frame t	Chapter 11
p_r	Probability of natural recovery from the current fused state	Chapter 13
C_r	Cost of recovery if recovery succeeds	Chapter 13
C_d	Cost of drift if recovery fails	Chapter 13
$E[\text{wait}]$	Expected cost of inaction: $p_r \cdot C_r + (1 - p_r) \cdot C_d$	Chapter 13

Symbol	Meaning	Introduced
o, d, a, f	Components of a frame tuple: observation, decision, action, feedback	Chapter 4
t	Frame index (discrete time step within a trajectory)	Chapter 4
T	Total number of frames in a trajectory	Chapter 5
k	Number of GNN message-passing rounds (hop depth)	Chapter 9

Diagram Legend

All diagrams in this book use the rough.js hand-drawn aesthetic. The following conventions are consistent across all figures.

Stroke color. White strokes on dark backgrounds represent structure and context: graph edges, trajectory paths, coordinate axes, and region boundaries. Blue strokes (#2563EB) mark the active or emphasized element: the current trajectory position, the highlighted phase region, or the signal being discussed.

Line weight. Thin lines represent background structure: the full graph, historical trajectories, coordinate grids. Thick lines represent foreground elements: the trajectory under discussion, the current phase boundary, the active signal.

Labels. All diagram labels use monospace uppercase text (e.g., EXPLORING, INERTIA, FRAME 15). Labels are placed outside the diagram region when possible to avoid occluding data.

Arrows. Trajectory direction is indicated by arrowheads at the leading end of the path. Causal or data-flow relationships use thin arrows with open heads. Feedback loops use curved arrows.

Regions. Phase regions in fused-space diagrams are indicated by dashed boundaries with the phase name outside the boundary. Overlapping regions (transition zones) use dotted boundaries.

Axes. When axes are labeled, the horizontal axis is frame index (time) unless stated otherwise. Vertical axes are labeled with the quantity being plotted.

Recurring Definitions

Frame. The unit of agent behavior: a single turn's observation, decision, action, and feedback, captured as the tuple (o, d, a, f). Chapter 4.

Trajectory. An ordered sequence of frames representing one agent session or subtask. Chapter 5.

Inertia. Context mass projected along the current trajectory heading. Measures how much accumulated work resists a change in direction. Chapter 7.

Entropy. Shannon entropy computed over sliding windows of agent behavior distributions (file targets, action types, module boundaries). Measures coherence loss over time. Chapter 8.

Space. The geometric terrain derived from workspace graph structure via GNN embeddings. The environment that constrains trajectory movement. Chapter 9.

Fused signal. The gated composition of inertia, entropy, and space into a joint embedding that captures their interactions. Chapter 11.

Phase. One of six trajectory regimes (exploring, converging, compacting, recovering, drifting, stalling) identified by characteristic signatures in the fused embedding space. Chapter 12.

Harness. The scaffolding system that manages an agent's tool access, context window, and execution loop. Chapter 15.

WBA (Write-Back Architecture). An architecture where the agent's work product and its measurement record are the same artifacts. Chapter 15.

Construct. A unit of structured information in a linguistic programming format, carrying executable semantics and serving simultaneously as work product and telemetry. Chapter 15.

Pressure field. A distributed coordination mechanism that assigns numerical urgency scores to regions of work, updated through a four-phase tick cycle (decay, sense, activate, validate). Chapter 16.

Glossary

Action distribution — The probability distribution over possible next actions an agent can take at a given frame, as shaped by the current context window. (Chapter 8.)

Agent — A program that uses a large language model to accomplish tasks through an iterative loop of observe, decide, act, and receive feedback. (Chapter 1.)

BDI (Belief-Desire-Intention) — An agent architecture formalism where behavior is driven by the agent's beliefs about the world, desires it wants to satisfy, and intentions it has committed to. (Chapter 2.)

Bit construct — A unit of structured information in a linguistic programming format that carries executable semantics and serves simultaneously as work product and measurement record. (Chapter 15.)

Branching — A point in a trajectory where the agent changes direction, abandoning or modifying its current approach. High inertia resists branching; external evidence or intervention can force it. (Chapter 7.)

Budget — The total resource allocation for an agent session, measured in tokens, frames, or wall-clock time. Budget exhaustion is one cause of stalling. (Chapter 12.)

Closed loop — A system where measurement outputs feed back into operational inputs without human intermediation. The five-step cycle: measure, characterize, intervene, learn, improve. (Chapter 14.)

Compacting — The third trajectory phase. The agent is executing a plan with high, stable inertia and low, flat entropy. Most productive work happens here. (Chapter 12.)

Conservation — The principle that total context mass in a trajectory is neither created nor destroyed, only transformed between inertia, entropy, and spatial displacement. (Chapter 10.)

Context — The full contents of an agent's input at a given frame: task description, accumulated observations, tool outputs, reasoning traces, and any injected information. (Chapter 1.)

Context dynamics — The study of how the three forces (inertia, entropy, space) interact to shape agent trajectory behavior over time. The subject of this book. (Chapter 1.)

Context mass — A scalar quantity that accumulates with each frame, computed as the temporally weighted sum of step magnitudes: $M(t) = \sum w(t-i) \cdot \|\Delta e_i\|$. (Chapter 7.)

Converging — The second trajectory phase. The agent has formed a hypothesis and inertia rises sharply as actions begin building on each other in a consistent direction. (Chapter 12.)

Coordinate space — The continuous geometric space defined by combining content embeddings (what files say) with structural embeddings (where files sit in the dependency graph). Trajectories move through this space. (Chapter 6.)

Curvature — The rate at which a trajectory's heading changes in embedding space. High curvature indicates rapid direction change; low curvature indicates straight-line movement. (Chapter 7.)

Decay — The reduction of a quantity's influence over time. Applied to context mass (older frames contribute less) and pressure field scores (urgency fades without renewal). (Chapters 7, 16.)

Decomposition — Breaking a complex task into subtasks, each of which generates its own sub-trajectory. Legitimate decomposition has sawtooth entropy signatures; drift does not. (Chapter 8.)

Direction — The normalized vector of recent trajectory movement in embedding space. The heading that inertia resists changing. (Chapter 7.)

Domain drift — An entropy signature where the agent's actions spread across unrelated modules without returning to the original focus. Entropy rises monotonically. (Chapter 8.)

Drifting — The fifth trajectory phase. The agent has lost direction and coherence. Inertia is low, entropy is high and rising, spatial position wanders. (Chapter 12.)

Edit thrashing — A pattern where the agent repeatedly modifies and reverts the same code, producing high local activity with zero net progress. Detectable via mutual information between edits and test outcomes. (Chapter 8.)

Embedding — A learned vector representation that maps discrete objects (files, tokens, frames) into continuous geometric space where distances encode similarity. (Chapter 3.)

Entropy — Shannon entropy computed over sliding windows of agent behavior distributions. Measures the degree to which an agent's actions have lost coherence with respect to a unified goal. (Chapter 8.)

Escalation — An intervention strategy that replaces the current agent or approach with a more capable alternative when recovery probability is low. (Chapter 13.)

Exploration — Behavior characterized by broad information gathering without commitment. Low inertia, moderate entropy, spatial position near the task-type origin. (Chapter 12.)

File scatter — The number of distinct files an agent touches within a window. High scatter with low mutual information between files indicates unfocused activity. (Chapter 8.)

Frame — The atomic unit of agent behavior: one turn's observation, decision, action, and feedback, captured as the tuple (o, d, a, f). (Chapter 4.)

Fused signal — The gated composition of inertia, entropy, and space: $F_t = g_I(t) \cdot \hat{I}_t + g_E(t) \cdot \hat{E}_t + g_S(t) \cdot \hat{S}_t$. Captures cross-signal interactions that individual measurements miss. (Chapter 11.)

Gating function — A learned function that assigns per-frame weights to each input signal (inertia, entropy, space) based on the current trajectory state. Implements the mixture-of-experts composition. (Chapter 11.)

GNN (Graph Neural Network) — A neural network that operates on graph-structured data, propagating information along edges to produce node embeddings that encode local topology. Used to embed codebase structure. (Chapter 9.)

Graph embedding — A vector representation of a graph node produced by a GNN, encoding both the node's content and its structural neighborhood. (Chapter 9.)

Harness — The scaffolding system that manages an agent's tool access, context window, prompt construction, and execution loop. The operational substrate that produces trajectory data. (Chapter 15.)

Harness Dynamics Model — The coordination mechanism that consumes WBA data and governs agent behavior via pressure fields. Together with the WBA, it closes the measurement-to-action loop. (Chapter 16.)

Historical edge — An edge in the workspace graph connecting files that frequently co-change in version control history. Encodes temporal coupling not visible in code structure. (Chapter 9.)

Inertia — Context mass projected along the current trajectory heading: $I(t) = M(t) \cdot \cos(\theta)$. Measures the accumulated contextual weight pulling the trajectory forward in its current direction. (Chapter 7.)

Intervention — An external action that modifies an agent's context or behavior during a session. Types include hints, context injections, strategy resets, and escalation. (Chapter 13.)

JEPA (Joint Embedding Predictive Architecture) — An architecture that makes predictions in embedding space rather than token space, enabling smooth macroscopic forecasts of trajectory movement. (Chapter 11.)

KL divergence — A measure of how one probability distribution differs from a reference distribution: $D_{KL}(P\|Q) = \sum P(x) \log \frac{P(x)}{Q(x)}$. Used to detect distributional shift in agent behavior. (Chapter 8.)

Mixture of experts — A model architecture where a gating function routes inputs to specialized sub-networks and blends their outputs. The fused signal uses this principle to compose trajectory signals. (Chapter 11.)

Mutual information — The amount of information shared between two variables: $\mathrm{MI}(X;Y) = H(X) + H(Y) - H(X,Y)$. Used to detect whether agent reads are connected to subsequent productive actions. (Chapter 8.)

Observation — The information an agent receives at the start of a frame: file contents, command output, error messages, or task descriptions. The o component of the frame tuple. (Chapter 4.)

Partial observability — The condition where an agent cannot see the full state of its environment, only the observations available through its tool calls. All real-world coding agents operate under partial observability. (Chapter 2.)

Phase — One of six trajectory regimes (exploring, converging, compacting, recovering, drifting, stalling) identified by characteristic signatures in fused embedding space. (Chapter 12.)

Phase transition — The boundary between two phases, detectable as a discontinuity or rapid shift in the fused signal. Transition sharpness correlates with outcome quality. (Chapter 12.)

POMDP (Partially Observable Markov Decision Process) — The formal framework for sequential decision-making under partial observability. Agent behavior can be modeled as a POMDP where the hidden state is the full codebase and the observations are tool outputs. (Chapter 2.)

Pressure field — A distributed coordination mechanism that assigns numerical urgency scores to regions of a work space, updated through a four-phase tick cycle. Replaces centralized orchestration. (Chapter 16.)

Read fingerprint — The pattern of files an agent reads and the order in which it reads them. Circular read fingerprints indicate stalling; expanding fingerprints indicate productive exploration. (Chapter 8.)

Recovering — The fourth trajectory phase. The agent's plan encountered a failure and it is adjusting. Inertia drops, entropy rises, spatial position shifts laterally. (Chapter 12.)

Repetition — Recurrence of the same action sequence without progress. Distinguished from purposeful revisiting by low mutual information between repeated actions and subsequent outputs. (Chapter 8.)

Scaffold — Synonym for harness. The infrastructure that wraps an LLM to produce an agent: prompt templates, tool definitions, execution loop, and context management. (Chapter 2.)

Semantic edge — An edge in the workspace graph connecting files that address the same domain concept, even without structural dependencies. Discovered via text embedding similarity. (Chapter 9.)

Session — A complete run of an agent on a task, from initial prompt to final output. Produces one trajectory. (Chapter 1.)

Shannon entropy — The expected information content of a probability distribution: $H = -\sum p_i \log_2 p_i$. Measures the average surprise per event. (Chapter 8.)

Space — The geometric terrain derived from workspace graph structure. The third dynamic force, representing the environment that constrains trajectory movement. (Chapter 9.)

Speculation — Agent behavior where the model generates hypotheses without grounding them in observations. Detectable as frames with high decision content but low observation content. (Chapter 8.)

Stalling — The sixth trajectory phase. The agent has stopped making meaningful progress. Near-zero velocity in fused space. May present as repetitive actions or token generation without tool calls. (Chapter 12.)

Structural edge — An edge in the workspace graph representing an explicit dependency: an import, require, or include statement. (Chapter 9.)

Temporal decay — A weighting function that reduces the contribution of older frames to current measurements. Prevents context mass from growing without bound. (Chapter 7.)

Tick — One cycle of the pressure field update loop: decay, sense, activate, validate. The temporal resolution of the coordination system. (Chapter 16.)

Tool call — An agent's invocation of an external tool (file read, shell command, code edit, search) to interact with its environment. The a component of the frame tuple. (Chapter 1.)

Tool diversity — The variety of distinct tool types used within a window. Low diversity (all reads, no edits) can indicate stalling or circular behavior. (Chapter 8.)

Topology — The structure of the workspace graph: which files connect to which, through what types of edges, forming what clusters and bottlenecks. (Chapter 9.)

Trajectory — An ordered sequence of frames representing one agent session or a coherent sub-portion of a session. The primary object of study in context dynamics. (Chapter 5.)

Turn — One iteration of the agent loop: the agent receives input, produces output, and executes an action. Equivalent to one frame in the trajectory formalism. (Chapter 1.)

Velocity — The rate of movement through fused embedding space, measured as the magnitude of the step vector between consecutive fused embeddings. Near-zero velocity indicates stalling. (Chapter 12.)

WBA (Write-Back Architecture) — An architecture where the agent's work product and its measurement record are the same artifacts. The substrate that makes trajectory data structural rather than reconstructed. (Chapter 15.)

Bibliography

This bibliography is organized by topic rather than alphabetically. Each section includes a brief annotation explaining why the cited works matter in the context of this book.

Agent Formalisms

The formal study of agents predates large language models by decades. These works provide the theoretical scaffolding that Chapters 1 and 2 draw on to define what an agent is, what it does, and what it means for an agent to act rationally under uncertainty.

Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2020.

Wooldridge, M. and Jennings, N. "Intelligent Agents: Theory and Practice," *The Knowledge Engineering Review*, 10(2):115-152, 1995.

Rao, A. and Georgeff, M. "BDI Agents: From Theory to Practice," *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, 1995.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. "ReAct: Synergizing Reasoning and Acting in Language Models," *International Conference on Learning Representations (ICLR)*, 2023.

Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. "Toolformer: Language Models Can Teach Themselves to Use Tools," *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Embeddings and Representation

The idea that meaning can be captured geometrically runs through the entire book, from frame embeddings in Chapter 3 to workspace graph

embeddings in Chapter 9. These works established the theoretical and practical foundations for representing discrete objects as continuous vectors.

Firth, J.R. "A Synopsis of Linguistic Theory, 1930-1955," in *Studies in Linguistic Analysis*, Philological Society, Oxford, 1957.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. "Distributed Representations of Words and Phrases and their Compositionality," *Advances in Neural Information Processing Systems (NeurIPS)*, 2013.

Grover, A. and Leskovec, J. "node2vec: Scalable Feature Learning for Networks," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

Information Theory

Chapters 4 and 9 use information theory as the primary instrument for measuring context coherence. Shannon's original paper and the Cover and Thomas textbook provide the mathematical foundations for entropy, mutual information, and KL divergence as applied throughout this book.

Shannon, C.E. "A Mathematical Theory of Communication," *Bell System Technical Journal*, 27(3):379-423, 1948.

Cover, T.M. and Thomas, J.A. *Elements of Information Theory*. Wiley-Interscience, 2nd edition, 2006.

Graph Neural Networks

Chapter 9 uses GNNs to transform discrete codebase graphs into continuous geometric spaces. These two papers introduced the convolutional and sampling-based approaches to graph learning that make workspace embedding practical at scale.

Kipf, T.N. and Welling, M. "Semi-Supervised Classification with Graph Convolutional Networks," *International Conference on Learning Representations (ICLR)*, 2017.

Hamilton, W.L., Ying, R., and Leskovec, J. “Inductive Representation Learning on Large Graphs,” *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Agent Evaluation

Chapters 1 and 18 argue that current agent evaluation is impoverished. These works represent the state of the art in agent benchmarking and intelligence measurement, and they illustrate both the progress and the limitations that motivated this book.

Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” *International Conference on Learning Representations (ICLR)*, 2024.

Chollet, F. “On the Measure of Intelligence,” *arXiv preprint arXiv:1911.01547*, 2019.

Dynamical Systems

The physics metaphor that structures this book, forces, trajectories, phase transitions, conservation, is drawn from the mathematics of dynamical systems. Strogatz’s textbook provides the accessible treatment of nonlinear dynamics that informs the analogy in Chapters 8 through 13.

Strogatz, S.H. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*. Westview Press, 2nd edition, 2015.

Mixture of Experts and JEPA

Chapter 11 uses two architectural ideas to build the fused signal: mixture-of-experts gating for signal composition and JEPA for trajectory prediction. The Shazeer paper introduced the sparsely-gated mixture that inspired the gating function. LeCun’s position paper proposed prediction in embedding space as an alternative to generative prediction, the principle behind the trajectory forecaster.

Shazeer, N., Mirhoseini, A., Mahdavi, K., Davis, B., Le, Q., Hinton, G., and Dean, J. “Outrageously Large Neural Networks: The Sparsely-Gated

Mixture-of-Experts Layer," *International Conference on Learning Representations (ICLR)*, 2017.

LeCun, Y. "A Path Towards Autonomous Machine Intelligence," *OpenReview preprint*, 2022.

Reinforcement Learning

The closed-loop system described in Chapter 14 learns from trajectory outcomes, an RL problem. Soft Actor-Critic provides the entropy-regularized policy optimization that balances exploration and exploitation in intervention strategy selection. Algorithm Distillation demonstrates that learned policies can be compressed into language model behavior, a bridge between RL and LLM agents.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *International Conference on Machine Learning (ICML)*, 2018.

Laskin, M., Wang, L., Oh, J., Parisotto, E., Spencer, S., Steigerwald, R., Strouse, D., Hansen, S., Filos, A., Brooks, E., Gazeau, M., Sahni, H., Singh, S., and Mnih, V. "In-Context Reinforcement Learning with Algorithm Distillation," *International Conference on Learning Representations (ICLR)*, 2023.

Organizational Theory

Chapter 7's treatment of exploration versus exploitation in agent trajectories, and Chapter 16's pressure field dynamics, draw on March's foundational work on how organizations balance searching for new approaches against refining existing ones. The tension March described in human organizations appears in agent behavior in a measurably analogous form.

March, J.G. "Exploration and Exploitation in Organizational Learning," *Organization Science*, 2(1):71-87, 1991.

Architecture Patterns

The Write-Back Architecture in Chapter 15 inherits from two traditions: literate programming, where code and documentation are the same artifact, and event sourcing, where system state is derived from an immutable log of events. Both patterns share the WBA's core principle that the record of what happened and the thing that happened should not be separate.

Knuth, D.E. "Literate Programming," *The Computer Journal*, 27(2):97-111, 1984.

Fowler, M. "Event Sourcing," *martinfowler.com*, 2005.

Overeem, M., Spoor, M., and Jansen, S. "The Dark Side of Event Sourcing: Managing Data Conversion," *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.