FIVE

When Words Break Systems

"Words do not break all at once. They fracture slowly. Silently. One conversation at a time."

The Pattern Beneath the Stories

You have heard three stories from fintech, logistics and healthcare. Different domains with different technologies, developed by different teams, but all have the same pattern.

A word started with one meaning. It gained a second. Then a third. Nobody said no. The meanings drifted. The architecture froze the drift. The system died(eventually and silently).

Now I want to tell you about four patterns. Not three. Four. Because there is a fourth pattern that is more subtle than the others. More invisible. More dangerous. It does not announce itself with a bloated table or a fractured team or an exploding enum. It announces itself with silence. With words that wander between worlds, belonging to no one, owned by no one, modeled by no one.

The first pattern is **Concept Overload**. One word carrying too many concepts. The second is **Concept Collapse**. Different meanings collapsing into the same name. The third is **State Explosion**. One state machine trying to serve multiple interpretations. The fourth, my favorite, is **Wanderer Words**. Concepts that exist in the spaces between models, homeless, unmodeled, waiting to cause trouble.

Each pattern has a different smell. Each pattern needs a different cure. Each pattern is a different way that language breaks systems. But the fourth pattern is the one that most teams never see. And because they never see it, they never fix it. And because they never fix it, the system slowly suffocates.

Pattern One: Concept Overload - The Promotion Story

Let me tell you about Promotion. Not the kind you get at work. The kind that breaks systems.

An e-commerce company with smart people and good technology stack. They had a promotion engine. Everyone was proud of it. The marketing team used it for campaigns. The pricing team used it for discounts. The engineering team used it for performance. The operations team used it for monitoring.

The word Promotion was everywhere. In the code. In the database. In the APIs. In every meeting.

Here is what no one noticed.

When marketing said *Promotion*, they meant a campaign. A time-limited thing with a budget and a target audience and creative assets. They wanted to know how many customers saw the promotion, how many clicked, how many bought. They cared about reach and conversion.

When pricing said *Promotion*, they meant a discount rule. A mathematical thing that adjusted prices. They cared about margin. They wanted to know how much revenue they were giving up. They ran simulations and forecasts.

When engineering said *Promotion*, they meant a database record with a type field and a bunch of conditional logic. They cared about performance. They wanted the thing to run fast and not break.

When operations said *Promotion*, they meant a running process. Something that was active or scheduled or expired. They cared about monitoring and alerts and rollbacks.

//TODO, image

Four teams, one word. The word *Promotion* had become a meeting place for four different concepts. Everyone put their stuff in the meeting place because no one had the authority to say no. The word swelled. It became a container. A junk drawer. A landfill.

The *Promotion* table had seventy-five columns. Most were null. The code had conditionals everywhere.

if promotion.type == "campaign"

if promotion.type == "discount"

if promotion.type == "process"

The challenge here is not about handling types as string on enum! If you get this vibe, so go back again and read the paragraph again.

The conditionals nested. The bugs multiplied. The system slowed. The word did not fracture. It overloaded. It tried to carry more conceptual weight than any single word should carry. And it collapsed under the weight.

This is **Concept Overload**. One word carrying more concepts than it can hold. The word stops being a pointer to a meaning. It becomes a container for whatever anyone wants to put in it.

Be caution. **Concept Overload** happens slowly. One new meaning at a time. Each addition seems reasonable. Of course marketing needs to track campaign budgets. Of course pricing needs discount rules. Of course engineering needs performance metrics. Of course operations needs monitoring.

But no one asks the question that matters. Are these the same concept or just the same word?

They are not the same. They never were. The word just hid that fact.

Pattern Two: Concept Collapse - The Customer Story

Let's see a different company with financial services. The word was *Customer*. The word was everywhere. In the code. In the database. In the APIs. In the conversations. Here is what no one noticed.

Compliance said *Customer* meant a legal entity. Someone who signed a contract. Someone the company had a formal relationship with. They cared about KYC, about regulations, about audit trails.

Sales said *Customer* meant an opportunity. Someone who might buy something. Someone to call and email and nurture. They cared about leads and pipelines and conversion rates.

Support said *Customer* meant a person with a problem. Someone who called or emailed or chatted. Someone who needed help. They cared about ticket counts and resolution times.

Finance said *Customer* meant a payer. Someone who owed money or was owed money. Someone with an account balance. They cared about invoices and collections.

Marketing said Customer meant a persona. Someone with demographics and psychographics and shopping habits. They cared about segments and campaigns and lifetime value.

Five teams. One word. No one had ever sat down and asked what Customer actually meant.

The system had a single *Customer* table. Every team had added columns to it over the years. The table had more than a hundred columns. Most of them optional. Most of them only used by one team. Most of them null for most customers. At that point, it was less of a table and more of a cry for help.

The team called it the monster table. They joked about it. But they were not really joking. They were in pain.

This is different from **Concept Overload**. In Concept Overload, the word swells. It absorbs new meanings. In Concept Collapse, the word fractures. Different meanings live under the same name because no one wanted to pay the cost of separating them. The cost of saying *we have five kinds of Customer and they need five different models.*"

Concept Collapse is harder to see than Concept Overload. With Overload, you can feel the word getting heavy. With Collapse, the word feels normal. Everyone uses it. Everyone understands it their own way. The problem only becomes visible when the teams try to share data. Or when the monster table starts collapsing under its own weight.

The Customer table had a hundred columns. Not because anyone designed it that way. Because the word Customer had collapsed. Five meanings. One table. No survivors.

Pattern Three: State Explosion - The Order Story

You have heard part of this story. The food delivery company. But I left out the worst part.

The system had been running for three years. Three years of adding features. Three years of fixing bugs. Three years of everyone using the same Order table.

The Order table had a status field. Simple. Everyone needs a status field. The status started with three values.

PENDING. ACTIVE. CLOSED.

Life was good.

Then the product team wanted to show customers the status of their order.

ORDERED. PREPARING. OUT_FOR_DELIVERY. DELIVERED.

They added four new values.

Then the restaurant team wanted to track kitchen status.

RECEIVED. COOKING. READY. PICKED_UP.

They added four more.

Then the driver team wanted to track delivery status.

ASSIGNED. PICKED_UP. IN_TRANSIT. ARRIVED.

They added four more.

Then the payment team wanted to track payment status.

AUTHORIZED. CAPTURED. SETTLED. REFUNDED.

They added four more.

Then the support team wanted to track exception status.

DELAYED. CANCELLED. LOST. DISPUTED.

They added four more.

The status field now had twenty-three possible values. The code had conditionals everywhere.

if status in ["DELIVERED", "CANCELLED", "LOST"]

if status in ["PENDING", "AUTHORIZED"]

The conditionals multiplied. The bugs multiplied.

One day, the product team asked for a feature. They wanted to show customers the status of their order in real time. Kitchen preparing. Driver assigned. On the way. Delivered.

Simple request. Every food delivery app has this. It took three months.

Not because the technology was hard and not because the team was slow. Because no one knew what status actually meant anymore. For the customer, status meant where is my food. For the restaurant, status meant what do I need to cook. For the driver, status meant where do I need to go. For payment, status meant when do I charge. For support, status means is this a problem yet.

One field. Twenty-three values, and five interpretations. The system had a single state machine trying to serve multiple interpretations. Each interpretation had its own rules. Its own transitions. Its own lifecycle.

This is **State Explosion**. One state machine trying to serve multiple interpretations until it becomes impossible to understand what any given state actually means.

State Explosion is subtle because each individual state seems reasonable. You can always add one more value to the enum. One more status code. What is the harm? The harm is that each addition makes the state machine more ambiguous. States that mean different things to different people get forced into the same namespace. The word DELIVERED means one thing to the customer and another thing to the driver and another thing to the restaurant. But the system cannot know that. The system just sees a string.

Three months for a status indicator. That is the cost of **State Explosion**.

Pattern Four: Wanderer Words - The Payroll Story

Now let me tell you about a different kind of failure. A more subtle one. A failure that happens not when a word means too many things, but when a word means something that no one is modeling at all.

Let's go into a payroll company with two departments, Payroll and Accounting.

The payroll team processed employee salaries. They had concepts like Employee, Salary, Timesheet, Bonus, Deduction. Their model was clean. Their code was clean.

The accounting team recorded financial transactions. They had concepts like JournalEntry, Account, Debit, Credit, Ledger. Their model was clean. Their code was clean.

The two systems needed to talk to each other. Payroll needed to tell accounting about salary payments. Accounting needed to record those payments as journal entries.

Here is what happened in every meeting.

A payroll person said: *"We need to book a journal entry for this salary payment."*

An accounting person said: *"What is the account code for the salary expense?"*

The payroll person said: *"That is not our problem. We just pay the employees."*

The accounting person said: *"We cannot record the entry without an account code."*

The meeting stalled. People got frustrated. They blamed each other. They blamed the process. They blamed the software.

But the problem was not the process. The problem was not the software. The problem was the words.

The phrase *payroll journal entry* was a **Wanderer Word**. It did not belong to payroll. Payroll did not own it. Payroll did not understand it. Payroll did not want to own it. But Payroll could not stop using it because that was how they talked to accounting.

The phrase *salary expense account code* was a **Wanderer Word**. It did not belong to accounting. Accounting owned the account codes, but the concept of *salary expense* was

about employees, not about accounts. The word wandered between the two domains, homeless, unmodeled, owned by no one.

Here is the critical insight that most teams miss.

When you have two models, you actually have three. You have Model A. You have Model B. And you have the conversation between them. The conversation has its own language. Its own concepts. Its own rules. Its own constraints.

//TODO need an image

The payroll team had their model. The accounting team had their model. No one had modeled the conversation. The words that appeared in the conversation *journal entry, account code*, *salary expense*, *posting*, *reconciliation* were wanderers. They belonged to neither model. They floated between them. They caused confusion. They caused meetings. They caused bugs.

The team tried to fix the problem by pushing the wanderer words into one of the existing models. They added JournalEntry to the payroll model. Now the payroll team had to learn accounting. The model became bloated. The team became confused.

They tried pushing the wanderer words into the other model. They added SalaryExpense to the accounting model. Now the accounting team had to learn payroll. The same problems.

The wanderer words did not belong to either model. They belonged to the space between the models. The space of the conversation. The space of the integration. The space of the handshake.

That space needs its own model. Explicit. Deliberate. Owned.

The Conversation Model

Here is what the payroll company needed to do. Not shove wanderer words into payroll or accounting. Build a third model. The conversation model.

The conversation model would have its own concepts. *PayrollInstruction. AccountingRecord. AccountMapping. PostingRule. ReconciliationStatus.*

These concepts belong to neither payroll nor accounting. They belong to the integration. They are the language of the handshake.

The payroll team would emit a PayrollInstruction. The instruction would say: "We paid employee 123 salary of $5000."

The conversation model would receive the instruction. It would look up the account mapping. It would apply the posting rule. It would generate an AccountingRecord.

The accounting team would receive the record. It would post it to the ledger.

Now the wanderer words have a home. JournalEntry no longer wanders. It lives in the conversation model. AccountCode no longer wanders. It lives in the conversation model. The payroll team does not need to understand accounting. The accounting team does not need to understand payroll. The conversation model translates.

This is not duplication. This is clarity. The payroll model is clean. The accounting model is clean. The conversation model is clean. Three models, not two. Always three.

Why Teams Miss the Third Model

Most teams never see the third model. They see Model A. They see Model B. They assume that the conversation between them is just... conversation. Just words. Not worthy of modeling. This is a mistake.

The conversation has structure. It has rules. It has constraints. It has lifecycle. It has state. It has the same complexity as any other part of the system. But because it lives between teams, between systems, between departments, no one claims it. It becomes a wanderer. It wanders until it causes chaos.

I have seen this pattern everywhere.

An e-commerce company had a conversation between the cart and the inventory. The wanderer word was *Reservation*. The cart said "*reserve this item*." The inventory said "*what does reserve mean?*" The conversation was never modeled. The word wandered. The system failed.

A healthcare company had a conversation between the scheduling system and the billing system. The wanderer word was *Authorization*. The scheduling system said "*this appointment is authorized*." The billing system said "*authorized by whom*? For what amount? Until when?" The word wandered. The system failed.

A logistics company had a conversation between the warehouse and the carrier. The wanderer word was *Manifest*. The warehouse said "*the manifest is ready*." The carrier said "*ready for what? Pickup? Loading? Dispatch?*" The word wandered. The system failed.

Every integration has wanderer words. Every conversation between models has concepts that belong to neither model. Every handshake needs its own language.

Most teams ignore this. They shove wanderer words into one of the existing models. Or they leave them implicit in code, in comments, in meetings, in Slack threads. The words continue to wander. The confusion continues to compound. The system continues to decay.

Wanderer Words are the most invisible pattern of semantic failure in my experience. Because they do not leave a trace in the code. They leave a trace in the meetings. In the confusion. In the frustration. In the time wasted.

And by the time you notice, the words have been wandering for years.

The Common Thread

Four patterns. Different failures. Different symptoms. Different cures.

Concept Overload: One word doing too much.

Concept Collapse: One word meaning too many things.

State Explosion: One state machine serving too many interpretations.

Wanderer Words: Concepts that live between models, homeless, unmodeled.

But one common thread.

In every case, the architecture was trying to preserve distinctions that the language had already destroyed. Or in the case of **Wanderer Words**, the architecture was pretending that distinctions did not exist when they clearly did.

The promotion engine had sophisticated discount calculation logic. The customer system had elegant domain models. The order system had clean service boundaries. The payroll system had two beautiful models.

None of it mattered. Because the words had already broken. And once words break, architecture cannot save you. Architecture can only freeze the break into place. Make it permanent. Make it expensive to fix.

The fintech company had a beautiful architecture diagram. The logistics company had a reasonable data model. The healthcare company had certified software. The e-commerce company had a state-of-the-art promotion engine. The financial services company had a well-designed customer relationship system. The food delivery company had a scalable microservices architecture. The payroll company had two immaculate domain models.

Every single one of them died. Not because the architecture was wrong. Because the language was broken. The architecture just made the break permanent.

The Naming of the Problem

At the time, I did not have names for these patterns. I called it ambiguity. I called it confusion. I called it the naming problem. I called it integration hell. None of those names captured what was actually happening.

Concept Overload. Concept Collapse. State Explosion. Wanderer Words.

Four patterns. Four kinds of semantic failure. Four ways that language breaks systems.

I am giving you these names now for a reason. Not because you need to memorize them and not just because I expect you to solve them yet. Because when you see them in your own system, you will have a word for the pain. And having a word for the pain is the first step toward fixing it.

The solutions exist. They have names too. **Meaning Split. Semantic Boundary. State/Status Segregation. Explicit Context. Semantic Event. Policy as Language**. You will meet them in later parts of this book. Part II, Part III, Part IV, Part V. Each pattern is a chapter. Each chapter is a tool.

But this chapter is not about tools. This chapter is about seeing. About recognizing that your system is not dying because of bad code. About understanding that the fracture is linguistic before it is technical.

So here is what I want you to take from this chapter.

When you see a word that means too many things, you are looking at **Concept Overload**. There is a pattern for that. We will get to it.

When you see a word that different teams define differently, you are looking at **Concept Collapse**. There is a pattern for that. We will get to it.

When you see a status field with twenty values and no one knows what they mean, you are looking at **State Explosion**. There is a pattern for that. We will get to it.

When you see a conversation between two models that is not modeled, words that wander between teams and systems, you are looking at **Wanderer Words**. There are patterns for that. We will get to them.

For now, just sit with the patterns. Let them resonate with your own experience. You have seen these failures before. You just did not have names for them. Now you do.

What Comes Next

This chapter introduced four patterns of semantic failure. **Concept Overload. Concept Collapse. State Explosion. Wanderer Words.** Four ways that language breaks systems. Four things you can now see in your own codebase.

The next chapter introduces a name for the underlying problem that connects all four patterns. Semantic Debt. The debt that accumulates when words lose their stable meaning. The debt that no one measures. The debt that kills systems slowly. The debt that includes **Concept Overload, Concept Collapse, State Explosion, and Wanderer Words** as its symptoms.

But before you turn the page, I want you to do something.

Look at your own system. Find one word that might be overloaded. Find one word that different teams define differently. Find one status field that has grown out of control. Find one conversation between models that is not modeled.

Just find them. Do not fix them. Not yet. Just see them. Seeing is the first step.

Thank you for taking the time to read this chapter.

If you found it valuable, the complete book is waiting for you at the link below.

✦ Masoud Bahrami ✦

📌 Read Online: https://masoudbahrami.com/ldd