

September 2026

Building systems that last: The pattern language of the connected enterprise

A guide to integration, cloud and agentic AI
through the MuleSoft and Salesforce lens



Contents

03 Your guide to a shared language for better technology decisions

04 Chapter 1: Why software architects talk in patterns

08 Chapter 2: Enterprise integration is hard, but messaging presents a solution

12 Chapter 3: Patterns for surviving the cloud — and why this matters

16 Chapter 4: The newest patterns in the catalog are for agents

20 In closing: Where this leaves you

Your guide to a shared language for better technology decisions

In large organizations, technology challenges can often come down to communication gaps between business and technology teams.

The business wants an order to reach billing without being rekeyed, or a customer record that is consistent on every system. The technical team responds with gateways, brokers and events. Both are describing the same thing, only in different terms. And if you can't translate between them, scope drifts and costs rise.

The software architecture industry solved this problem for itself by creating catalogs of patterns.

Since 1977, designers have maintained a shared catalog of "patterns" that are assigned to recurring problems, proven solutions, and the trade-offs of each decision. These catalogs have been refined through every generation of technology, from object-oriented software to cloud-native integration. A pattern name compresses hours of explanation into a single word or phrase that can be used by anyone to communicate clearly. Every well-built system rests on these foundations.

At NTT DATA, every recommendation we make — whether it's monolith or microservices, synchronous or asynchronous, workflow or agent — traces back to these patterns.

We believe the thinking behind a solution should be just as transparent as the solution itself, and that a client who shares this vocabulary becomes a better partner in every design decision. Although the patterns are vendor-neutral, every organization eventually implements them on a real platform. Throughout this guide, we use MuleSoft and Salesforce to show how these patterns work in practice.

You don't need a technical background to read this guide, and the chapters can be read in any order:

- **Chapter 1** explains what a pattern is and why the discipline exists.
- **Chapter 2** examines why enterprise integration is inherently difficult and why messaging became the dominant style.
- **Chapter 3** covers the patterns that keep cloud systems reliable and shows that each one is ultimately a business decision.
- **Chapter 4** brings the catalog up to date with the patterns now emerging for agentic AI.

Each chapter concludes by exploring what those patterns mean for organizations using — or considering — these platforms, from simpler integration and greater resilience to new AI capabilities.



Wikus Smit

Senior Systems Architect
MuleSoft and Salesforce Practice
NTT DATA Middle East and Africa

Chapter 1: Why software architects talk in patterns

A 50-year-old shared vocabulary that runs today's software

Walk into almost any software architecture review and you'll hear people talking about circuit breakers, message routers, sagas, hexagonal architecture and API gateways. If you're new to it all, it can feel like everyone is speaking a different language.

They're not. They're speaking in **patterns** — a 50-year-old shared vocabulary that experienced designers use to describe problems, solutions and trade-offs quickly. Once you understand that vocabulary, three things happen: You can follow senior conversations, look up proven approaches instead of inventing your own, and challenge decisions when someone reaches for the wrong tool.

The one big idea

A pattern is a named problem plus a proven solution plus an honest list of trade-offs. Think of it as shorthand for a decision that experienced teams have had to make many times before.

Think of cooking. "Boil the rice" stands for a whole approach: rinse the grains, add the right amount of water, bring it to a boil, then lower the heat and let it cook until the water is absorbed. Once you know what "boil the rice" means, you can talk about it to anyone (including a top chef).

"Use a circuit breaker" works the same way: one phrase stands in for pages of reasoning about failure, timeouts and recovery.

The structure of a good pattern

All good pattern descriptions share the same structure:

- A **name**
- The **problem** it solves, expressed in plain language
- The **context**
- The **forces** that make the problem hard to solve
- The **solution** and its **consequences**
- **Examples** from real systems.

If any of the information is missing, the pattern isn't finished.

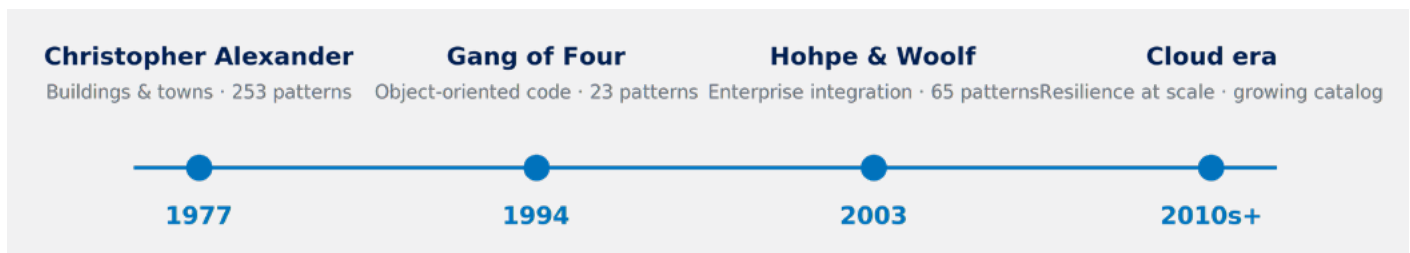


Apply this to your world

Think of the last design conversation you were in that went in circles.
How would the elements of a good pattern have helped to move the conversation forward?

Where the idea came from (and how it evolved)

The pattern movement did not start in software but with an architect looking at buildings.



Fifty years of shared vocabulary – from buildings to the cloud.

1977: Christopher Alexander, building architect

Christopher Alexander, an architect, published *A Pattern Language: Towns, Buildings, Construction*, a book of 253 patterns ranging from the height of a windowsill to the planning of entire towns. Each pattern names a problem (“a room feels dead when light only enters from one side”) and offers a solution (“light on two sides of every room”).

Alexander’s lasting contribution was the format of the pattern: named, structured, composable units of design knowledge. Almost 50 years later, software architects still use his template almost verbatim.

1994: The Gang of Four

Four programmers — Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, the so-called “Gang of four” — published *Design Patterns: Elements of Reusable Object-Oriented Software*. They cataloged 23 patterns, labeled Singleton, Factory Method, Observer, Strategy and so on, for building software at the level of individual classes and objects. For the first time, two engineers on opposite sides of the world could say, “Let’s use the observer pattern here,” and mean exactly the same thing.

2003: Gregor Hohpe and Bobby Woolf

By the late 1990s, a new class of problem had emerged. Businesses were running customer relationship management (CRM), enterprise resource planning (ERP), human resources, billing, warehouse management and countless other systems, but none of them were designed to communicate with one another.

Hohpe and Woolf published *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, cataloging 65 patterns for moving data between systems through messaging. Their vocabulary — channels, routers, translators, aggregators and splitters — now underpins every modern message broker, event bus and application programming interface (API) gateway.

Their book also initiated the creation of products: A generation of integration platforms was built directly on its vocabulary, including the open source Mule project that became MuleSoft, which began that very year — named after the “donkeywork” of moving data between systems.

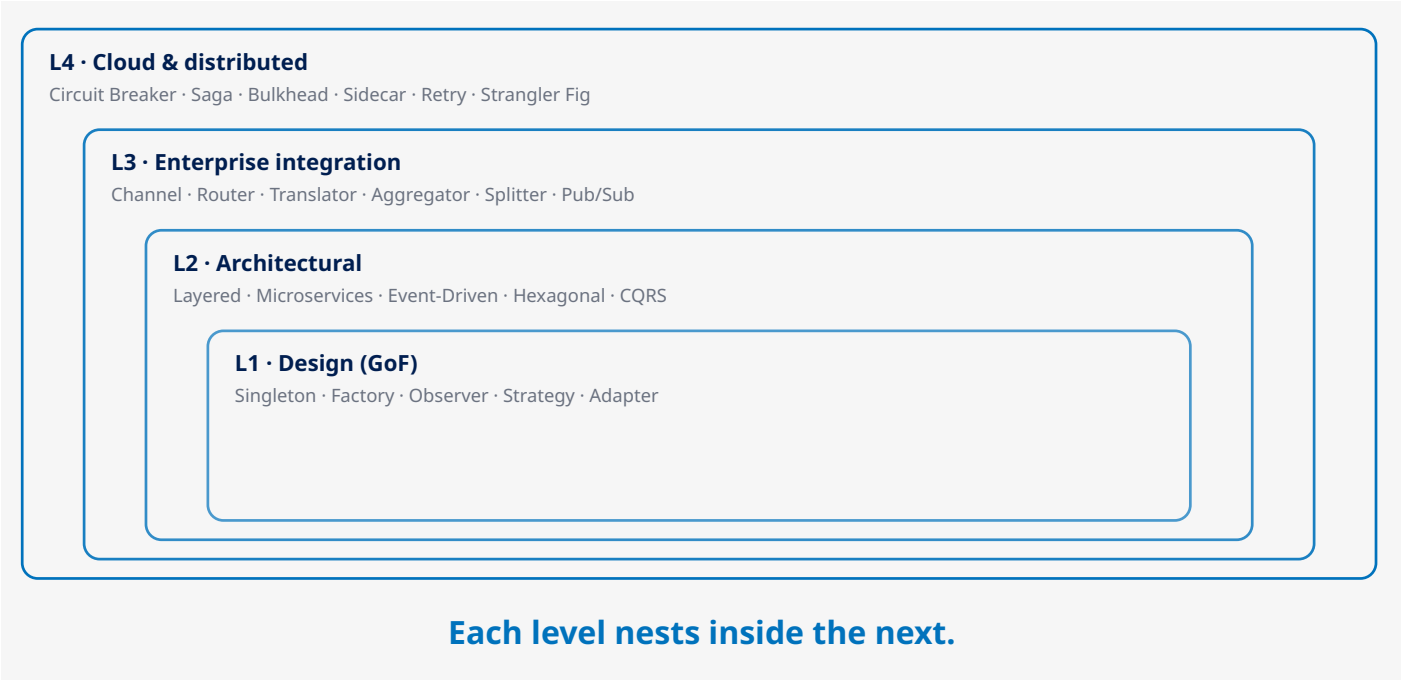
2010s onward: The cloud era

Another set of problems appeared when systems moved to the cloud: networks failed, servers disappeared and dependencies slowed down without warning. The response was a fresh wave of patterns focused on resilience under uncertainty, with names such as Circuit Breaker, Bulkhead, Saga, Sidecar and Strangler Fig. The catalog keeps growing.

The four scales of patterns

Patterns exist at different scales. Some describe how a small part of an application should behave; others describe how many systems should work together across an organization.

A pattern for how one class of code behaves inside your application is different from a pattern for how 12 services collaborate across 3 data centers.



Level	Scope	Examples
L1. Design patterns “How do I let one object notify many others without them knowing each other?”	One codebase, at class level How individual objects collaborate inside a single program (this is the gang of four’s territory)	Singleton, factory method, observer, strategy, decorator, adapter, facade, command
L2. Architectural patterns “Should this be a monolith or many small services?”	One application How the whole application is shaped, including its overall structure, its major components and how they talk	Layered, microservices, event-driven, hexagonal, pipe and filter, Command Query Responsibility Segregation (CQRS), Model-View-Controller (MVC), broker
L3. Enterprise integration patterns “How does an order get from the shop to the warehouse without locking the two together?”	Many applications How separate systems collaborate across an organization: the vocabulary of messaging, routing, transformation, aggregation and delivery guarantees	Message channel, router, translator, aggregator, splitter, publish-subscribe
L4. Cloud and distributed patterns “What does my service do when a downstream dependency is slow?”	Global and unreliable systems How systems survive when things fail at global scale and across unreliable networks and constantly changing infrastructure	Circuit breaker, retry, bulkhead, saga, sidecar, API gateway, strangler fig

These scales may be confusing at first. The key is to remember that each one answers a different kind of question. We use the same word — **pattern** — but we’re talking about very different kinds of design decisions.

When you hear a pattern name, ask at which level it is found. A factory method (L1) and a circuit breaker (L4) are not competing ideas. They operate at different scales and can coexist in one system.



Apply this to your world

Pick one system you work on.

Which patterns is it using right now that nobody has ever explicitly identified?

Read patterns to understand the competing forces

Every pattern balances competing forces. For example, a circuit breaker has to balance two pressures: keep trying in case the failed service recovers, or stop trying so the failure does not spread.

The trade-offs matter just as much: Every pattern gives you something, but it also asks you to give something up, whether that's complexity, latency, coupling or something else. With a circuit breaker, the pattern protects the wider system by stopping calls for a while, but the trade-off is that some requests may fail sooner than the user expects.

Applying a pattern without understanding the competing forces and trade-offs is a beginner's mistake, not a design decision.

A codebase littered with textbook patterns for no reason is a warning sign.

Great software architects use the fewest patterns possible to solve a problem and can defend why each one is there.

The same discipline applies in reviews. It's easy enough to recite pattern names, but real expertise is found in explaining the forces a pattern balances — and knowing when not to use it.



How NTT DATA, MuleSoft and Salesforce address this

"Every project starts from zero."

Does that sound familiar?

MuleSoft's API-led connectivity turns integrations from one-off plumbing into reusable assets: Connect a core system once and that integration becomes an asset you can reuse. Every new integration becomes faster because more of the groundwork has already been done.

Engagements led by NTT DATA's apps and BPS: MEA applications, Salesforce, begin the way this chapter ends: by deliberately naming the patterns already found in your estate and showing how they can be reused.

Starting with patterns gives everyone a shared language before technology choices enter the discussion. In our experience, it's usually the fastest way to move from a whiteboard conversation to a practical roadmap.

Chapter 2: Enterprise integration is hard, but messaging presents a solution

Six systems, 15 connections, a decade of regret — and the one pattern that fixes it

Many large organizations tell the same story: They bought a CRM, then an ERP, then a warehouse system. Next, they acquired a company that came with its own billing platform. Then someone wanted a single view of everything. Ten years later, they have hundreds of integrations, most of them undocumented and several written by contractors who left years ago. And every change starts with a team meeting.

This is called integration debt, and it's one of the biggest reasons large organizations struggle to change.

It's also the problem that a whole generation of software patterns was invented to solve.

The N^2 problem

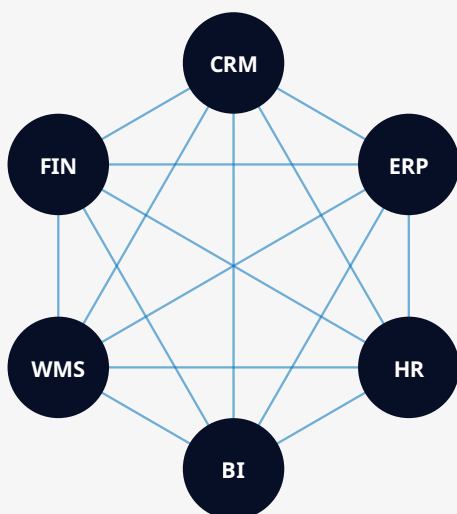
If every system needs to talk to every other system directly, the number of connections grows very quickly: 6 systems need 15 connections, 10 systems need 45, and 20 systems need 190. Every new system means rewiring those that already exist. It quickly becomes spaghetti architecture, and you need integration patterns to untangle it.

This is how integration debt accumulates. Every point-to-point connection solves an immediate problem, but each one also becomes another dependency to understand, maintain and eventually untangle. Over time, the cost of changing the landscape grows faster than the landscape itself.

The fix is deceptively simple: Put a broker in the middle. Six systems, six connections; when you add one more, you add just one more link, too, not five. Every integration platform of the past 20 years — from the enterprise service buses of the 2000s to MuleSoft's Anypoint Platform today — is essentially this idea turned into a platform.

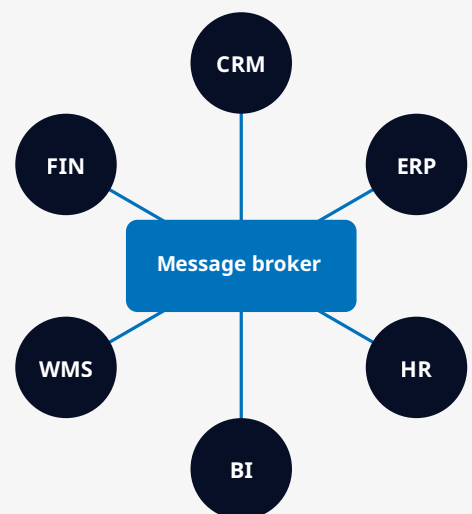
Before: Point-to-point spaghetti

6 systems, 15 connections



After: Hub-and-spoke broker

6 systems, 6 connections



The N^2 problem — a broker turns 15 connections into 6.

Four ways systems can talk

In 2003, Hohpe and Woolf identified four fundamental integration styles. You'll see all four in production somewhere. Each has its place, but one of them became dominant:

1. File transfer: One system writes a file (CSV, XML or JSON) and another reads it. It's as simple as email attachments, and still found in payroll, billing and legacy exports.

The verdict: Simple, universal and reliable — but slow. No real-time actions, so data goes stale.

2. Shared database: Two applications read and write the same database. It's fast and consistent, but every change to the schema breaks everyone at once.

The verdict: Fast, but the most tightly coupled option.

3. Remote procedure call: One system calls another synchronously over the network, typically using REST APIs, SOAP or gRPC. To the developer, it feels like calling a local function, even though another system is doing the work.

The verdict: Familiar and straightforward, but dependent on the other system being available. If the callee is down, you're stuck.

4. Messaging: Systems exchange messages through a broker (a queue or event bus). The sender doesn't wait; the receiver reads when ready. Store-and-forward reliability.

The verdict: Loosely coupled, resilient and the preferred choice for most modern integration.



Apply this to your world

Could your organization enumerate every integration it runs today?

If the answer is no, you already have integration debt; you just haven't priced it yet.

Integration styles in action: Coffee shops don't use two-phase commit

The best way to understand why messaging works so well is to consider what happens once you place an order at a coffee shop.¹ You pay at the cashier, who doesn't wait until your drink is ready before serving the next customer. They write your name on a cup, put the cup in a queue on the espresso machine and take the next order. A barista pulls cups from the queue and makes the drinks. When yours is done, the barista calls your name.

This design lets the coffee-shop process far more customers per hour than if the cashier had to wait. It also lets them add more baristas when it's busy; the queue happily absorbs the load. Your name on the cup is how the barista matches drinks to customers, even when drinks aren't ready in the same sequence in which the orders were placed.

The coffee shop uses three familiar integration patterns:

1. Asynchronous: The cashier doesn't wait for the drink to be made.

2. Competing Consumers: Many baristas can work from the same queue of orders.

3. Correlation Identifier: Your name on the cup matches the finished drink to the right customer.

And when things go wrong? The coffee shop doesn't lock the till until your latte is confirmed. If you can't pay after the drink is made, they discard it (**a write-off**). If the drink is wrong, they remake it (**a retry**). If you're unhappy, they refund you (**a compensating action**).

Real-world businesses have always preferred these pragmatic recovery moves over distributed locks (stopping everything and locking the whole process until every step is confirmed), which is exactly the trade-off that modern messaging systems make.

¹ Gregor Hohpe. [Your Coffee Shop Doesn't Use Two-Phase Commit](#). IEEE Software. March/April 2005.

What messaging actually does

With messaging, a message broker sits between the sender and the receiver. When the sender wants to send something, it hands the message to the broker. The broker stores the message, then delivers it once the receiving system is ready. If the receiver is offline, the message waits. If the receiver is overloaded, the queue absorbs the burst (it holds the extra work to avoid slowing everything down or crashing the receiving system).

This is called store and forward, and it is why messaging works. It decouples the sender from the receiver in **time** (they don't have to be online together), **space** (they don't have to know each other's addresses) and **capacity** (bursts of sending don't crash anything).

This decoupling is the architectural answer to integration debt. Instead of every application knowing about every other application, systems communicate through shared messaging patterns. Individual integrations become easier to change, and the overall landscape becomes easier to evolve.

Messaging comes in two forms:

- 1. **Point-to-point** delivers each message to exactly one receiver — like assigning a task to one person — and is used for commands ("charge this card").
- 2. **Publish-subscribe** delivers a copy to every subscriber — like a magazine subscription — and is used for events ("an order was placed"). If you know Salesforce Platform Events, you're familiar with this pattern inside a CRM.

A small catalog worth knowing

Modern message brokers implement dozens of patterns from the Hohpe and Woolf canon. A handful appear everywhere in modern systems:

Message router	Content-based router	Message filter	Message translator
Like a postal sorting office	Like a hospital triage nurse	Like an email spam filter	Like a human interpreter
Reads incoming messages and forwards them to different channels based on rules	Examines the message content to decide where to send it	Discards messages that don't match specific criteria	Converts a message from one format to another
Splitter	Aggregator	Dead letter channel	Correlation identifier
Like someone unpacking a grocery delivery	Like baggage claim	Like the office for undeliverable mail at the post office	Like your name on the coffee-shop cup
Breaks one composite message into many individual messages	Collects related messages until a set is complete, then publishes one combined message	A special channel where undeliverable messages end up	A unique tag on a message that lets the system match a response to its original request

There's another pattern worth knowing — this one about organizations rather than software. In 1968, Melvin Conway observed that organizations produced designs that copied their own communication structures.² In other words, your architecture ends up mirroring your organizational chart.

Integration is as much a people problem as it is a technology problem, and no broker will save you if two teams of employees refuse to talk.

² Melvin E Conway. [How do committees invent?](#). Datamation. April 1968.

Messaging isn't free

Asynchronous messaging trades one hard problem for another. You lose the simplicity of a synchronous call ("Did it work? Yes or no, right now") and gain resilience, but you also take on new challenges: How do you show progress to a user waiting for their order? How do you handle a message that failed processing? How do you replay events during a debug?

These are solvable problems, but they're still trade-offs. And not everything should be asynchronous: The coffee shop is asynchronous at the drink station but synchronous when you pay. Choose what makes the most sense for your given requirements.



Apply this to your world

Walk through one flow in your IT estate. Which steps are synchronous today that don't need an answer **right now** — and which are asynchronous where the caller genuinely needs that answer?



How NTT DATA, MuleSoft and Salesforce address this

"Every change needs a committee, and nobody can retire anything."

That's what the left side of the N² diagram feels like from the inside.

A broker-led architecture on **Anypoint Platform** — with **Salesforce Platform Events** publishing changes from the CRM in real time — reduces the number of moving parts every time another system is added. Instead of creating new point-to-point integrations, teams plug into existing patterns, making change simpler and easier to scale. In our delivery experience, the reduced cost of change pays for the platform within 6 to 12 months.

Technology should implement architectural decisions, not make them. That's why we start with patterns and choose platforms afterwards — not the other way around.

Chapter 3: Patterns for surviving the cloud — and why this matters

Every architectural choice is a business choice.

Cloud systems have a distinctive problem: things fail. Not occasionally — constantly. Servers disappear, networks partition and dependencies slow down without warning. Requests time out. A modern architect's job is not to prevent failure — that's impossible — but to survive it gracefully. A new generation of patterns emerged for exactly this.

Seven patterns that keep the lights on

1. Circuit breaker

Wraps calls to an unreliable dependency and trips open when failures cross a threshold.

Once open, further calls fail immediately without hitting the broken service — giving it time to recover and preventing your system from being dragged down with it (just like the electrical circuit breaker in your house.)

2. Retry (with backoff)

Automatically tries a failed call again, usually with an increasing delay between attempts.

Handles transient failures — the ones that would have worked if you'd tried a moment later. Similar to redialing a busy phone number.

3. Bulkhead

Isolates parts of the system into separate pools so a failure in one can't drain resources from the others.

If one dependency starts consuming all your threads or connections, the bulkhead confines the damage. (It's named after the watertight compartments in a ship's hull.)

4. Saga

Manages a business transaction spread across multiple services as a sequence of smaller local transactions.

If any step fails, compensating actions undo the earlier steps.

Think of it as booking a multileg trip: Reserve the flight, then the hotel, then the car — and if the car booking fails, cancel the hotel and the flight.

5. API gateway

A single front door for all external traffic to your back-end services.

Handles authentication, rate limiting, routing and protocol translation so that each individual service doesn't have to. It's like a restaurant waiter who coordinates with the kitchen, the drinks bar and the dessert station on your behalf. The pattern is now so well established that most organizations buy it rather than build it themselves. MuleSoft Omni Gateway (formerly Flex Gateway) is one example.

6. Sidecar

Deploys helper functionality alongside a service, in its own process, rather than inside the service's code.

Logging, monitoring, security and service discovery are all handled by the sidecar, leaving the main service focused on business logic.

7. Strangler fig

Incrementally replaces a legacy system by routing more and more functionality to a new one, until the old system can be safely removed.

It's never a big-bang rewrite; always a gradual replacement. Named after the strangler fig tree that slowly grows around a host in the rainforest until the fig stands on its own, in practice, the fig is usually an integration layer: a MuleSoft facade in front of the legacy system, redirecting one route at a time while revenue keeps flowing.

None of these patterns exists for its own sake. Each represents a business decision about what matters most: availability or cost, consistency or speed, simplicity or resilience. The technology is different; the trade-offs are the same.



Apply this to your world

Name the one dependency that, if it slowed to a crawl today, would take your whole platform with it. That's where your first Circuit Breaker and Bulkhead belong.

What happens without them

The best argument for using patterns is what happens when they're ignored.



\$460 million in 45 minutes

In 45 minutes, Knight Capital lost more than \$460 million.³ On the morning of 1 August 2012, the trading firm deployed a new piece of software to eight servers. Seven servers received the new code; one did not. A configuration flag that used to mean something else now reactivated a dormant piece of code called “Power Peg” — software that had been unused for years but was never removed.

When the market opened, the old code on the eighth server started firing trades. In just three-quarters of an hour, Knight Capital's router sent more than 4 million orders into the market while trying to fill just 212 customer orders — over 397 million shares — and lost more than \$460 million. Their systems had generated 97 automated warning emails before the market even opened, but nobody was reading them. The firm had to be rescued and merged with a competitor.

What was missing: automated deployment checks (to ensure all servers ran the same version), a kill-switch pattern, dead-code removal discipline and a functioning alerting pattern. None of these failures was unprecedented. Each already had a well-understood architectural pattern designed to prevent it. Knight Capital didn't use those patterns.

The monolith and the modern rerun

The distributed monolith: Ten years after the industry started moving away from the monolith, a strange new failure mode appeared. Teams broke their big applications into many microservices but wired them together with synchronous calls, shared databases and tight version dependencies.

The result: all the operational complexity of microservices but none of the independence. Any change requires deploying five services in sequence. One service's downtime cascades through the others. The architecture looks distributed on the whiteboard; in production, it behaves exactly like a monolith — only bigger, slower and with more places to fail.

What was missing was genuine asynchronous messaging between services, ownership boundaries around each service's data, circuit-breaker patterns to isolate failure, and — most importantly — the willingness to ask, “Does this really need to be its own service?”

Neither failure was caused by a lack of technology. Both were the result of architectural choices — and architectural choices are ultimately business choices.

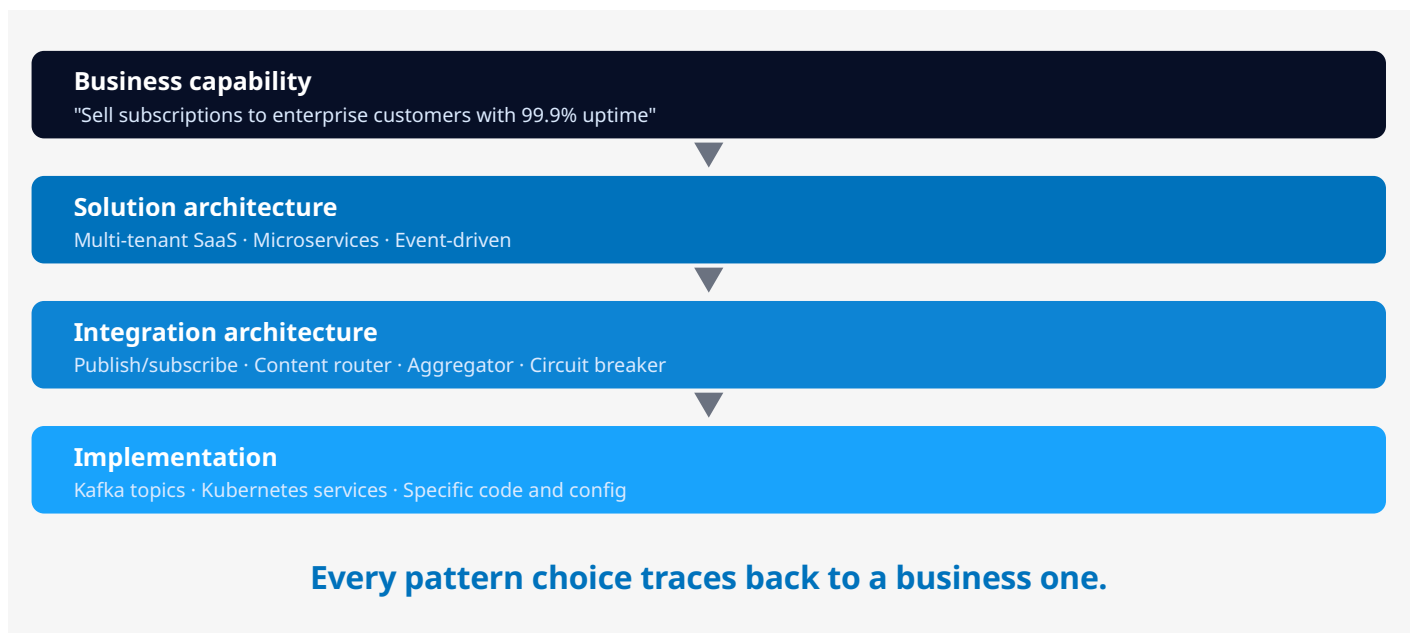
³ US Securities and Exchange Commission. [SEC Charges Knight Capital With Violations of Market Access Rule](#). 16 October 2013.

Every pattern choice is a business choice

Architects don't make technical decisions in isolation. Every major pattern choice reflects a business decision, whether anyone realizes it or not.

Read from top to bottom:

- **Business capability:** "Sell subscriptions to enterprise customers with 99.9% uptime." This is where executives think.
- **Solution architecture:** Multitenant or single-tenant? Cloud or on-premises? Monolith or microservices? These structural choices flow directly from the business intent.
- **Integration architecture:** How the pieces communicate: Pub/sub, routers, translators, sagas
- **Implementation:** Specific technologies, code, configuration



The single clearest example is the choice between multitenant and single-tenant software as a service:

- **Multitenant**
One shared application serves many customer organizations, at a lower cost per customer, faster updates and standardized features. Suits self-serve, high-volume, price-sensitive customers. Think apartment living: Shared plumbing, one landlord and standardized units.
- **Single-tenant**
A dedicated instance per customer organization, with higher cost, slower updates, room for customization and stronger isolation. Suits enterprises customers with strict security or bespoke needs. It's more like a standalone house: your own plumbing, your own rules and your own bills.

The commercial decision (who we sell to and at what price) makes the technical decision (which tenancy pattern) for you.

Every architecture choice traces back to a business choice.

The same principle applies to another common debate: "Asynchronous messaging is always better than synchronous calls." Asynchronous decouples systems but also makes flows harder to reason about, debug and test. If a step is small, fast and needs a definite answer before anything can proceed, a synchronous call is clearer.

Three myths that are worth busting

1. “You should use as many patterns as possible.”

The opposite is true. Every pattern adds complexity, indirection and cost. Great architects use the fewest patterns that solve the problem and can defend why each one is there.

2. “Microservices are always better than a monolith.”

Microservices solve organizational problems — letting many teams work independently — and specific scaling problems. For a small team on an early-stage product, a well-structured monolith is faster, simpler, cheaper and easier to reason about.

3. “The cloud makes systems resilient.”

It doesn't. Cloud platforms give you building blocks, but it's how you assemble them that makes you resilient. A cloud application without circuit breakers, retries or bulkheads is still fragile; it just fails on someone else's infrastructure.



Apply this to your world

Take your most recent architecture decision. Can you trace it, link by link, to the business decision that caused it? If the chain breaks halfway, that's where you'll find the design risk.

Where this leaves us

Patterns don't eliminate failure; they determine how failure behaves. When systems fail gracefully, customer organizations often don't notice. When they fail badly, every business promise built on top of them becomes harder to keep.

How NTT DATA, MuleSoft and Salesforce address this

“We signed a 99.9% uptime commitment. Can the architecture keep it?”

Most resilience patterns discussed in this chapter are available as platform capabilities rather than custom engineering.

MuleSoft Omni Gateway provides configurable policies such as rate limiting, spike control and circuit breaking, while **MuleSoft CloudHub 2.0** provides isolation and scaling for integration workloads. **Salesforce Hyperforce** applies many of the same resilience principles within the CRM platform, allowing teams to focus on business logic instead of infrastructure plumbing.

Learn more: [Mulesoft Omni Gateway](#)

At NTT DATA, that's the sequence we follow: business promises first, architectural patterns second, platform choices last.

Chapter 4: The newest patterns in the catalog are for agents

Fifty years after Alexander, the pattern movement has reached AI.

Picture a support “agent” that a team deployed straight into production. Asked to resolve a billing dispute, it calls the CRM, gets an ambiguous answer, decides to check again, gets the same answer and checks again, eventually spending hours in a loop, burning the API budget and rate limits while resolving nothing. What was missing: an iteration budget, a spending cap and an escalation checkpoint. An agent without a stopping condition is unattended, not autonomous.

That story, in a dozen variations, is why the newest section of the pattern catalog exists. Every wave of computing eventually gets its patterns — the first three chapters traced that lineage from objects to cloud integration. Now, agentic AI systems, where a large language model (LLM) decides what to do next, are getting theirs. Anthropic⁴, Google Cloud, IBM and others have all published catalogs in the past two years; strip away the branding, and the same designs appear in all of them.

The first decision is a business decision

Just as chapter 3’s choice between multitenant and single-tenant was really a commercial one, the first agentic question (workflow or agent?) isn’t technical either:

- A **workflow** orchestrates LLM calls through predefined code paths — the developer fixes the sequence and the model fills in the steps.
- An **agent** lets the model direct its own process — choosing tools, deciding whether to continue and declaring itself done.

Workflows are predictable, testable and relatively inexpensive. The same input produces broadly the same behavior, each step can be tested independently, and the possible failure modes are well-understood.

Agents are different. They choose their own path, which means they are more flexible but also more expensive — in tokens, latency and, above all, risk. That flexibility is exactly what makes them valuable, but it also makes their behavior harder to predict and govern. Who bears the cost of an error? Can the outcome be audited? Those are business questions, and they should determine the architecture before a line of code is written.

The advice you’ll find in every serious catalog is this: **Use the simplest shape that solves the problem.** Most production “agents” are, on inspection, workflows — and better for it.



⁴ Anthropic. [Building effective agents](#). 19 December 2024.

The anchor pattern: ReAct

If there's one pattern every practitioner learns first, it's **ReAct** (Reason + Act, from a 2022 research paper that most agent frameworks implement).⁵

The shape is a loop: The model **reasons** ("I need the customer's current plan before quoting an upgrade"), takes an action (calls the billing API), **reads the observation** (the response) and **reasons again** until it decides the task is done.

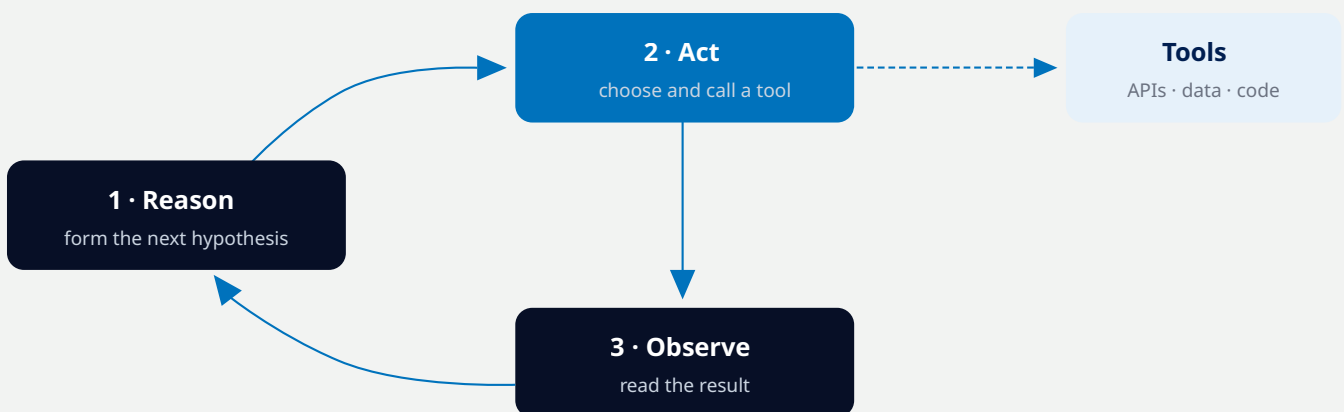
Think of a detective solving a case. They don't see all the evidence at once and immediately reach the right conclusion. Instead, they form a hypothesis, interview a witness, check a record, revise their thinking and repeat the process until the picture becomes clear. ReAct works the same way: The intelligence comes from repeatedly alternating between reasoning and gathering new information. Everything that follows builds on that idea: How do you turn a ReAct loop into a reliable system?

The newest AI patterns don't replace the old ones. Rather, they build on them. Just as enterprise integration introduced patterns for connecting systems, and cloud computing introduced patterns for surviving failure, agentic AI introduces patterns for governing autonomous behavior. Different technology, same discipline.



Apply this to your world

For the agent use case your team is discussing right now, does the model actually need to choose its own path or do you already know the path and just want the model to fill in the steps?



Loop until the model decides the task is done — or a guardrail stops it.

The ReAct - reason, act, observe, repeat.

⁵ Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan and Yuan Cao. [ReAct: Synergizing Reasoning and Acting in Language Models](#). Submitted 6 October 2022 (version 1); last revised 10 March 2023 (version 3).

The workflow patterns: Fixed shapes with predictable behavior

Five shapes recur in every catalog. They are the integration patterns of chapter 2, wearing new clothes:

- 1. Prompt chaining:** A fixed sequence of calls, each feeding the next: outline, then draft, then translate. Like a relay race: The baton passes in one direction, and you can add a judge between legs. It's Pipe and Filter, reborn.
- 2. Routing:** Classify the request, then dispatch to a specialized handler. It's chapter 2's hospital triage nurse, except now the nurse is an AI model.
- 3. Parallelization:** Run several calls at once and combine: split the task, or run it multiple times and take the majority. Think of a jury: 12 independent verdicts beat one confident guess.
- 4. Orchestrator-workers:** A lead model breaks the task into subtasks at runtime, delegates and synthesizes. Similar to general building contractors: They don't lay bricks but decide who does.
- 5. Evaluator-optimizer:** One model generates, a second critiques against explicit criteria, then the first revises. Every writer with a good editor knows this loop, and that it doubles the cost.

The agent patterns: Autonomy — with seatbelts

- **Tool use:** The model gets a described set of tools (APIs, databases and code) and chooses when to call them. A workshop is only as safe as its tool labels. In practice, tool design matters more than prompt cleverness. A tool called "search_customer_by_id", with a clear description and forgiving parameters, will be used correctly far more often than a vague tool wrapped in an elaborate prompt. Treat tool definitions like a public API — because to the model, that's exactly what they are. It's also why integration platforms abruptly became AI infrastructure: A catalog of well-governed APIs — what a MuleSoft estate has been accumulating for years — is, from the agent's point of view, a toolbox.

- **Reflection:** The agent reviews its own output before finalizing. Essentially, it's reading the angry email again before pressing send. Cheap and high value.
- **Planning:** Draft an explicit plan first, then execute and revise it. Think of an itinerary you can inspect before the trip; behavior becomes reviewable before it touches anything real.
- **Multiagent collaboration:** Researcher, coder and reviewer agents pass work along. Powerful, fashionable and most often applied prematurely. A multiagent system is a distributed system — tightly coupled, it's chapter 3's distributed monolith, except now it hallucinates.
- **Human-in-the-loop:** A person approves at defined checkpoints — before the email sends, before money moves. It's the co-signature on the check. For high-stakes actions, the approval step is the design working as intended. Salesforce Agentforce turns that checkpoint into a platform capability rather than relying on prompt instructions alone.



3 questions separate a durable agent design from a demo

1. Who pays for the error?

If the answer is "the customer," you need a human-in-the-loop.

2. Can you audit the outcome?

If not, add planning and reflection so the behavior is inspectable.

3. What stops it?

Every loop needs a budget — iterations, spending and time — and an escalation path for when the budget runs out.

What happens when agent patterns are missing

Again, nothing illustrates the importance of these patterns better than their absence.

The customer-facing chatbot that invented a policy

In 2024, a tribunal ordered Air Canada to compensate a customer after the airline's website chatbot incorrectly told a customer they could claim a bereavement refund after completing their journey. The airline had argued the bot was "responsible for its own actions" and lost⁶.

What was missing: grounding in approved sources, an evaluator checking answers against policy and a human-in-the-loop for commitments. The lesson we all learned from this is that an organization owns what its agent says.

Guardrails are the circuit breakers of this era

Cloud patterns exist because networks fail; agentic patterns exist because models fail. The goal is the same in both cases: Contain failure before it spreads. In cloud systems, that means retries, bulkheads and circuit breakers. In agentic systems, it means budgets, scoped permissions and human checkpoints.

None of these patterns appeared from nowhere. They're the same architectural ideas we've already explored — routing, retries, bulkheads, gateways and human checkpoints — applied to a new kind of system. The vocabulary has evolved but the underlying discipline hasn't.



Apply this to your world

If your "agent" ran unattended for a week, what is the worst thing it could plausibly do?
Your answer is your guardrail backlog.

How NTT DATA, MuleSoft and Salesforce address this

"Where do we put agents — and what stops them?"

Those are the right questions, and today they can be answered with platform capabilities rather than custom engineering.

Salesforce Agentforce provides the agent layer, including scoped actions, built-in guardrails and human escalation. MuleSoft supplies the governed tool layer through MCP support, allowing existing APIs to be exposed safely to agents while extending existing authentication, rate limiting and observability policies.

As agents multiply, MuleSoft Agent Fabric provides the governance layer, helping organizations orchestrate and govern agents and MCP servers across the business.

Learn more: [Mulesoft Agent Fabric](#)

A decade of integration investment becomes your foundation for agentic AI instead of a rebuild. Organizations with mature integration estates often discover they're much closer to enterprise AI than they first thought. Their integration platform doesn't become obsolete; it becomes the foundation agents build on.

At NTT DATA, we help you identify that foundation and extend it rather than replacing it.

⁶ The Guardian. [Air Canada ordered to pay customer who was misled by airline's chatbot](#). 16 February 2024.

In closing: Where this leaves you

The discipline of building durable systems by naming the problem, understanding the trade-offs and choosing the pattern that best fits the context is more than 50 years old. It has outlasted every platform, programming language and technology wave it has encountered.

It is also how NTT DATA works.

When we recommend an integration architecture, a modernization path or an agent deployment, the reasoning in these pages is the reasoning in the room: the same patterns, the same trade-offs, the same insistence that every technical decision must trace back to the business decision that motivates it. If you can't explain why a decision was made, it's difficult to know whether it will still be the right decision a year from now.

The goal of this guide wasn't simply to explain architectural patterns but also to give you a shared vocabulary for discussing them. Because once business and technology teams are speaking the same language, better decisions usually follow.

If you recognized parts of your own environment in this guide — the spaghetti integrations, the distributed monolith or the agent that never knew when to stop — that's a good thing. It means you can name the problem, and once you can name it, you can start solving it.

What comes next is usually straightforward: Which patterns fit your context? Which can wait? And where will they deliver the greatest business value? We'd welcome that conversation.



Start the conversation

Every organization accumulates architectural patterns — some deliberately, others by accident. A patterns-first architecture review helps you understand which ones are truly serving your business and where change will have the greatest impact.

NTT DATA's apps and BPS: MEA applications, Salesforce practice works with organizations across industries to identify those patterns, prioritize change and turn architectural decisions into measurable business results.

[Explore our partnership with Salesforce](#)

Visit nttdata.com to learn more.

NTT DATA is a \$30+ billion business and technology services leader in AI and digital infrastructure. We accelerate client success and positively impact society through responsible innovation. As a Global Top Employer, we have experts in more than 70 countries. NTT DATA is part of NTT Group.



