Gearset

Salesforce observability

Close the visibility gap in your DevOps lifecycle

Contents

Who should read this whitepaper?

Whether you're hands-on with development, responsible for designing scalable architecture, or managing the platform's overall health, this whitepaper is for you.

If it's your responsibility to keep a Salesforce org functional and available — excluding platform-wide outages beyond your control — this paper will provide the technical insights you need.

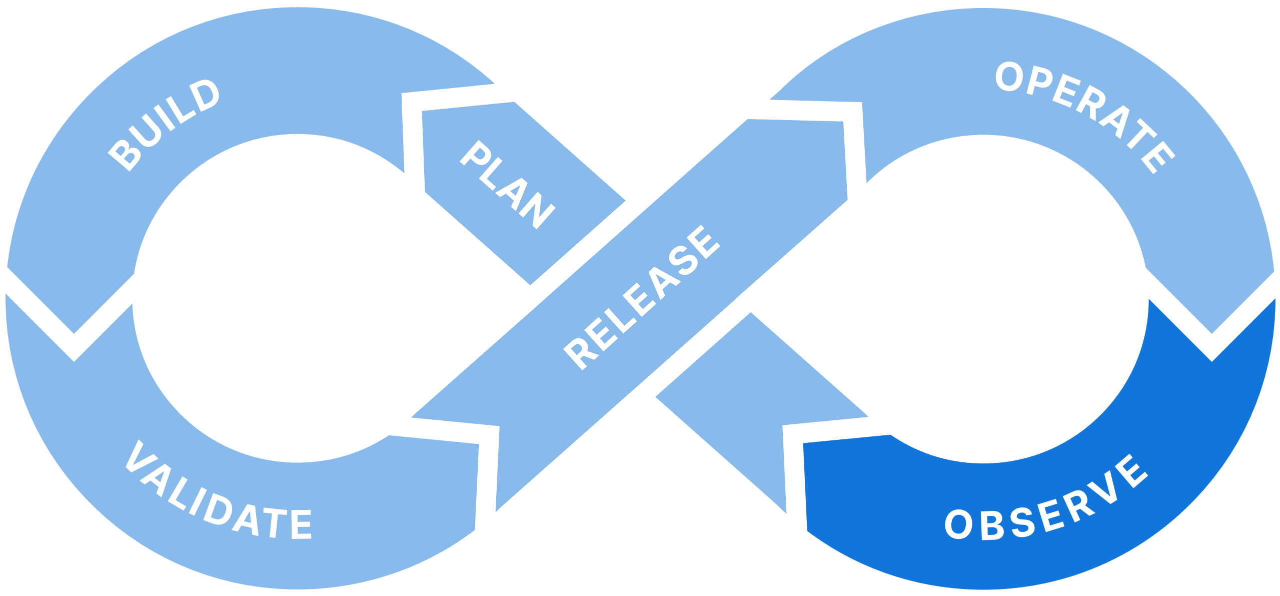
Admins	Gain a clearer understanding of how observability impacts org reliability and what can be done to improve it.
Developers	Learn techniques for better exception handling, logging, and debugging to improve code reliability.
Architects	Discover strategies for designing scalable systems that anticipate failure, adapt to growth, and stay resilient under platform constraints.
Platform managers	Get insights into maintaining system health, minimizing downtime, and ensuring platform-wide stability.

What is observability?

Observability is understanding the state of a system, based on its outputs. In a Salesforce context, it's the ability to assess the health, performance, and behavior of your orgs by analyzing the data they generate.

Observability is more than just setting up logs and alerts. It means proactively monitoring the health of your org, using a variety of metrics, so you can identify how to improve its stability and reliability. Observability minimizes the impact of issues by resolving them without users needing to report manually and maximizes the insights your team needs to debug and optimize the performance of your org.

Observability is an essential part of the DevOps lifecycle, not an afterthought or add-on. It's an ongoing process of surfacing insights that feed into planning and prioritization. These insights improve the quality of builds, reduce the need for reactive firefighting, and help teams deploy at speed without losing control.



Why DevOps needs observability

When performance drops or something breaks in your Salesforce org, you need to know what happened before you can fix it. But without the right insight you're left scrambling — digging through logs, chasing symptoms, and relying on users to tell you what's gone wrong. Observability changes that. It helps you move from reactive to proactive, from guesswork to understanding.

This shift is central to DevOps. Observability creates the feedback loops that DevOps depends on — helping teams detect issues early, fix them faster, and continuously improve how they deliver changes.

Consider the last time something went wrong in production. How quickly were you able to answer the following questions, if at all?

- What exactly has gone wrong — and where?
- Who's been affected?
- How do I fix it?
- Why has it happened?
- How do we prevent it happening again?

If those answers aren't immediately available, it's a sign your observability practices need strengthening. With the right tools and telemetry in place, these questions stop being emergencies — and become part of how you work every day.

Salesforce's observability gap — and why it matters

For a long time, observability has been a standard practice on other development platforms, yet it hasn't been widely implemented for Salesforce. Until now, most Salesforce teams have focused on solving the challenge of building a successful deployment process. But as DevOps adoption matures, observability is the natural next step and it's one that many teams are still missing.

According to the 2025 State of Salesforce DevOps Report, 49% of Salesforce teams say that observability isn't even on their radar. Of the teams without observability tools, 74% say they only find out about issues when users report them. It's a reactive cycle that keeps teams on the back foot.

Now, as Salesforce itself is championing observability and continues to align more closely with development best practices on other platforms, observability is something Salesforce teams can't afford to miss out on.

"We've been talking about observability a lot at Salesforce recently. What it means to me is ***getting access to data and insights*** *— whatever it is that is most relevant to you — and* ***being able to act on that data and those insights****."*

Karen Fidelak,
Senior Director Product Management, Salesforce

Unique challenges for Salesforce observability

Compared with the wider world of software development, most Salesforce teams are behind the curve when it comes to observability. That's not just down to priorities — the platform itself poses unique challenges that can make it harder to implement the kind of visibility that DevOps teams rely on.

Platform as a Service

Being a Platform as a Service (PaaS), Salesforce comes with certain trade-offs that affect how easily teams can observe and troubleshoot issues. The number and configuration of development and QA environments are often limited, and these environments rarely mirror production closely. That makes it harder to reproduce bugs or test under realistic conditions. Log creation and storage is also restricted, making it difficult to gather the right information at the right time.

Multi-tenant architecture

Salesforce's multi-tenant cloud architecture adds another layer of complexity in the form of governor limits (built-in limits on resource usage per transaction) and restricted logging capabilities. These limitations make it harder to identify, prevent, and remediate issues.

Lack of native tools

Salesforce's first-party tooling doesn't yet cover all the observability needs that come with building increasingly complex applications on the platform. While some monitoring and logging features are available, they aren't always designed to support enterprise-scale development or give teams the depth of insight they need across the DevOps lifecycle.

Difficulty getting buy-in

It can be harder to justify investment in observability than other DevOps solutions. Its benefits — like faster incident resolution, improved stability, and better developer feedback — often sit in the background, while delivering new features tends to take center stage. And with fewer well-established observability practices within the Salesforce ecosystem, teams don't always have a clear path to follow.

What you’re missing without Salesforce observability

Salesforce is a mission-critical platform for businesses, so uptime and reliability are crucial. As a result, organizations often rely on long UAT cycles to thoroughly test before releasing — but this can clash with Agile methodologies and slow down the pace of development.

As Salesforce applications scale, their complexity, scope, and volume of data all increase, making it even harder to see if the application can handle it all. Without strong observability practices in place, teams risk flying blind just when they need visibility the most.

Observability can stop serious disruption in production. In 2024, bugs caused a Salesforce outage at 21% of businesses. Teams with observability tools are 50% more likely to catch critical bugs like these within a day and 48% more likely to fix them just as quickly.

That kind of responsiveness is core to modern DevOps — especially the principle of “shifting left”, which focuses on identifying potential issues earlier in the development lifecycle, long before they reach production.

When salesforce observability is done right, teams can:

1. Reduce critical and costly system downtime
2. Adopt DevOps best practices and increase team capacity
3. Scale alongside business needs

*“**Observability is a cornerstone** of any successful Salesforce DevOps strategy. It’s not just about knowing what’s happening in your systems but **understanding why it’s happening** — so you can fix issues faster, prevent them proactively, and continuously improve how you **deliver value**.”*

Rob Cowell,
DevOps Advocate, Gearset

How to close Salesforce's biggest observability gaps

In this section, we'll walk through three of the most common observability gaps in Salesforce orgs. These aren't theoretical issues; they're the real-world pain points that block debugging, hide regressions, and slow down delivery.

For each one, we'll explore how to uncover the signals that matter, build smarter monitoring practices, and give your team the context they need to move faster with more confidence.

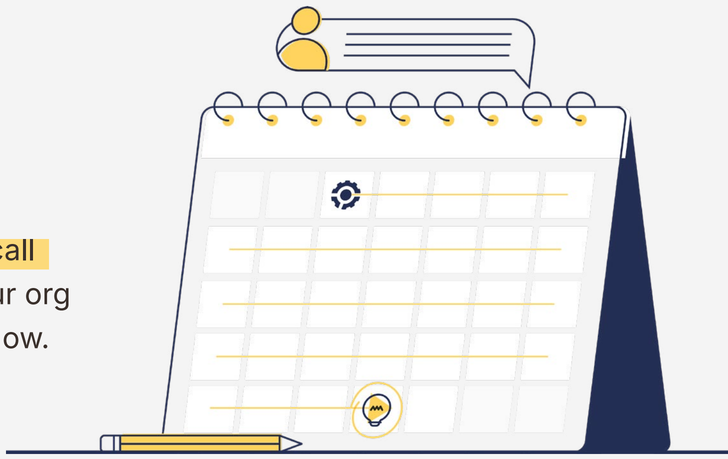
Here's what we'll focus on:

- Unhandled exceptions
- Governor limits
- Storage sprawl

Arrange a tailored demo with our DevOps experts

Kick off your observability process with a 15-minute call to talk through where your current gaps are, what your org needs, and how to build observability into your workflow.

BOOK A DEMO →



Unhandled exceptions

Blind spots	With observability
● Debugging is slow, manual, and often missing context.	● Metadata change tracking speeds up root cause analysis.
● Unhandled errors cause silent transaction failures.	● Proactive logging and real-time error visibility helps catch issues early.

Unhandled exceptions can seriously disrupt business continuity and app stability. When Salesforce hits unexpected errors without proper handling, it aborts the entire transaction and every data change is rolled back. Diagnosing these errors is challenging because the platform doesn't provide much context.

Exceptions	Exceptions are errors that disrupt code execution
Unhandled exceptions	Exceptions that the code doesn't or can't catch

In Apex, unhandled exceptions are those that aren't caught using a `try-catch` block. These blocks allow developers to catch runtime errors and handle them in a controlled way by showing a user-friendly message, triggering a recovery path, or at the very least, preventing the entire transaction from crashing. Without a `try-catch`, the platform has no choice but to fail hard and leave users in the dark.

But just catching an exception isn't enough. If the catch block simply logs the error or does nothing — a pattern known as swallowing exceptions — the system can be left in an inconsistent state without anyone noticing. Data might be committed after something went wrong, making problems harder to spot later.

When unhandled exceptions happen in Apex, you typically get a stack trace — a few lines of code that executed before the error — but no detail about the state of variables or the transaction that led up to it. If you're lucky, the user who experienced the issue might be able to describe what they were doing. But for asynchronous Apex like Batch or Scheduled jobs, there's often no way to know what triggered the failure or what data was involved.

In Flows, unhandled exceptions occur when there's no fault path configured, so the Flow stops and the user sees a generic error. This can seem unhelpful, but it can make unexpected issues easier to catch and fix. If fault paths just log problems without properly handling them, incomplete or invalid data might still be committed.

By default, Salesforce emails exception details to the last user who modified the affected component. Apex emails typically lack useful detail, while Flow emails provide technical information that's difficult to interpret.

Sending exception emails to the last modifying user isn't reliable, as that user might be inactive or a generic account with an unmonitored inbox. While Salesforce lets you choose who receives these emails, the inbox often fills with technical details that users struggle to interpret or act on quickly.

To debug unhandled exceptions effectively, teams need immediate, detailed visibility into what happened at runtime. But Salesforce's default logging makes that hard. Debug logs require a trace flag on a specific user, class, or process. These manually activated flags — diagnostic switches that trigger log capture — last up to 7 days by default (or 24 hours in the Developer Console), or until log storage runs out, whichever comes first. You can configure which types of events are captured, but only from a predefined set.

That might be fine when you can reproduce an issue in a sandbox. But exceptions are rarely predictable — and the storage limits make it unlikely that useful logs will exist when the issue first occurs. Compounding the problem, Salesforce debug logs only capture data if a trace flag is already active. By the time you know something went wrong, it's often too late to capture the detail you need.

Even when debug logs are available, analyzing them through Salesforce's default tools is difficult. Logs are lengthy, unstructured text streams, lacking consistent field names or formatting. In the Developer Console, lines are often truncated, there's no clear visual indication of transitions between classes or declarative logic, and overly granular trace flags add repetitive noise — making real issues difficult to identify or analyze.

To debug unhandled exceptions effectively, you need proactive strategies that go beyond Salesforce's native error handling.

Proactive exception handling

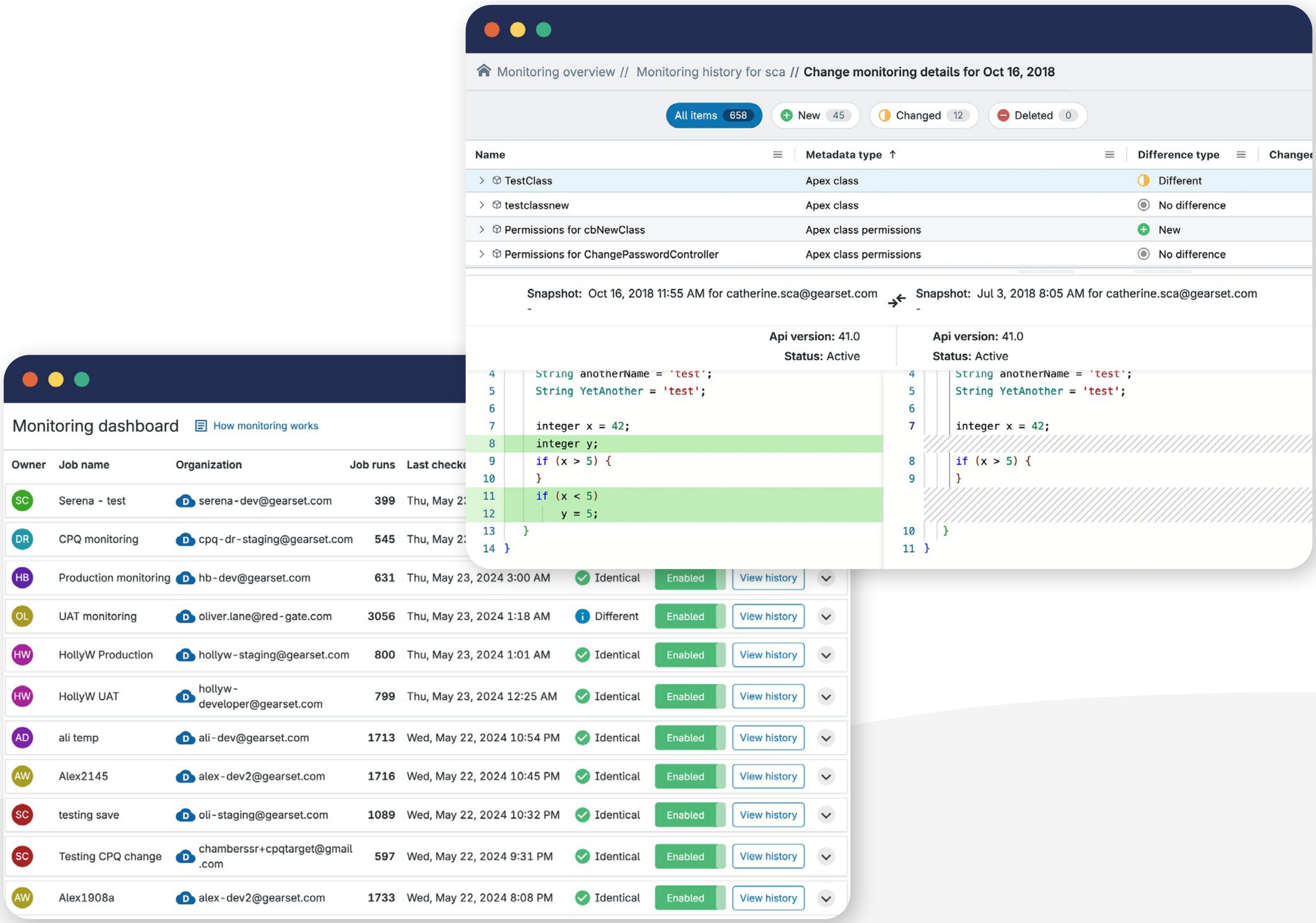
Dealing with unhandled exceptions proactively means building a feedback loop that enables early detection, faster debugging, and minimal disruption.

Track metadata changes across environments

Being able to correlate unexpected behavior with recent changes — by tracking metadata differences and deployment activity across environments — makes it easier to identify likely causes.

When an exception occurs, the first question to ask is whether the action that triggered it was expected — a known use case — or something entirely new. If it's an existing business process there's a chance that this was previously supported and some sort of regression has occurred. In that case, recent change and deployment logs can reveal where the change was introduced, and rolling back may be good enough to restore service — assuming your DevOps process can support this.

If the action itself is new — even if it looks similar to older behavior — the problem may lie in legacy logic that no longer fits today's needs. For example, a validation rule that once limited an Opportunity to three Line Items might have made sense in the past, but changing business requirements could now mean users need to add five. No code change triggered the error — the system simply didn't keep up with real-world use. In these cases, observability tools can surface the conditions that caused the failure, helping teams move quickly from detection to diagnosis to a targeted fix.


Monitoring overview // Monitoring history for sca // **Change monitoring details for Oct 16, 2018**

Name	Metadata type ↑	Difference type	Change
TestClass	Apex class	Different	
testclassnew	Apex class	No difference	
Permissions for cbNewClass	Apex class permissions	New	
Permissions for ChangePasswordController	Apex class permissions	No difference	

Snapshot: Oct 16, 2018 11:55 AM for catherine.sca@gearset.com
-

Api version: 41.0
Status: Active

Snapshot: Jul 3, 2018 8:05 AM for catherine.sca@gearset.com
-

Api version: 41.0
Status: Active

```
    String anotherName = 'test';
    String YetAnother = 'test';

    integer x = 42;
    integer y;
    if (x > 5) {
    }
    if (x < 5)
        y = 5;
  }
}
```

```
    String anotherName = 'test';
    String YetAnother = 'test';

    integer x = 42;
    if (x > 5) {
    }
  }
}
```

Monitoring dashboard

Owner	Job name	Organization	Job runs	Last checke			
SC	Serena - test	serena-dev@gearset.com	399	Thu, May 2			
DR	CPQ monitoring	cpq-dr-staging@gearset.com	545	Thu, May 2			
HB	Production monitoring	hb-dev@gearset.com	631	Thu, May 23, 2024 3:00 AM	Identical	Enabled	View history
OL	UAT monitoring	oliver.lane@red-gate.com	3056	Thu, May 23, 2024 1:18 AM	Different	Enabled	View history
HW	HollyW Production	hollyw-staging@gearset.com	800	Thu, May 23, 2024 1:01 AM	Identical	Enabled	View history
HW	HollyW UAT	hollyw-developer@gearset.com	799	Thu, May 23, 2024 12:25 AM	Identical	Enabled	View history
AD	ali temp	ali-dev@gearset.com	1713	Wed, May 22, 2024 10:54 PM	Identical	Enabled	View history
AW	Alex2145	alex-dev2@gearset.com	1716	Wed, May 22, 2024 10:45 PM	Identical	Enabled	View history
SC	testing save	oli-staging@gearset.com	1089	Wed, May 22, 2024 10:32 PM	Identical	Enabled	View history
SC	Testing CPQ change	chamberssr+cpqtarget@gmail.com	597	Wed, May 22, 2024 9:31 PM	Identical	Enabled	View history
AW	Alex1908a	alex-dev2@gearset.com	1733	Wed, May 22, 2024 8:08 PM	Identical	Enabled	View history

Metadata change monitoring

Stay ahead of failures with real-time change awareness

Gearset's metadata change monitoring brings visibility into day-to-day workflows. It tracks changes across your Salesforce environments, highlighting what's been added, modified, or removed — even outside of the formal deployment process.

Change monitoring gives your team the context they need to connect unexpected behavior to recent configuration or code changes, accelerating root cause analysis and reducing time spent hunting through logs or chasing assumptions.

Try it free for 30 days

Kick off your observability adoption with a **free trial** of Gearset's change monitoring solution.

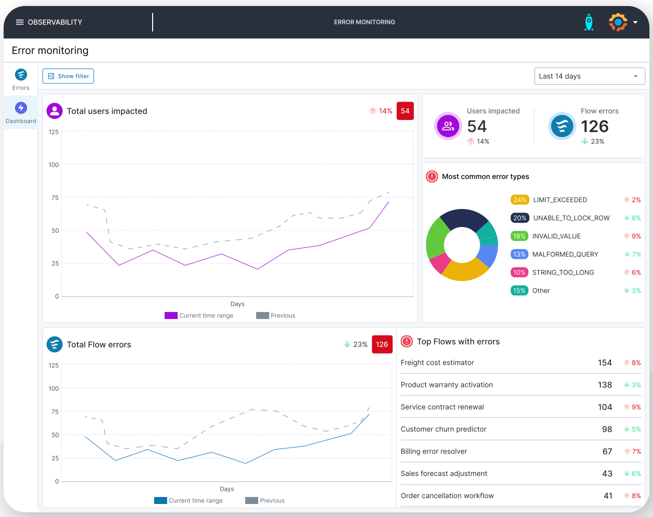
START FREE TRIAL

Surface and trace Flow and Apex errors

When an error occurs in Apex or a Flow, you're often starting with just the line of code where the error occurred and the exception message as your context. For something like a null pointer exception, where several variables could be involved, narrowing down the root cause becomes guesswork without visibility into the transaction's state at runtime.

Salesforce does have some native tools that can help. You can execute the Flow step-by-step in "rollback mode" to test or debug your Flows manually. This means changes made to the database in the debug run will be reverted so that you don't accidentally change any real data.

On the Apex side, tools like the Apex Debug Log Replayer and the free Apex Log Analyzer plugin for Visual Studio Code can offer a clearer view of execution flow and transaction bottlenecks.



Most common error types

24%	LIMIT_EXCEEDED	↑ 2%
20%	UNABLE_TO_LOCK_ROW	↓ 8%
18%	INVALID_VALUE	↑ 9%
13%	MALFORMED_QUERY	↓ 7%
10%	STRING_TOO_LONG	↑ 6%
15%	Other	↓ 3%

Top Flows with errors

Freight cost estimator	154	↑ 8%
Product warranty activation	138	↓ 3%
Service contract renewal	104	↑ 9%
Customer churn predictor	98	↓ 5%
Billing error resolver	67	↑ 7%
Sales forecast adjustment	43	↓ 6%
Order cancellation workflow	41	↑ 8%

Easy error monitoring

Proactively catch Flow and Apex errors and prevent them impacting users

Gearset's observability solution captures Flow and Apex errors in real time and surfaces them in an intuitive dashboard.

Get full visibility into your errors, so you can spot trends and diagnose root causes more easily. Configure custom alerts to catch unexpected spikes in errors and prevent issues from escalating. And use the data to guide strategic decisions and make the biggest impact on org health.

Try it free for 30 days

Kick off your observability adoption with a **free trial** of Gearset's observability solution.

START FREE TRIAL

Use event monitoring for runtime visibility

Salesforce's Event Monitoring, now available in a free tier as of the Winter '25 release, is a step toward making the right runtime signals more visible. It reflects a growing recognition that exception emails alone aren't scalable or sufficient for understanding and resolving failures.

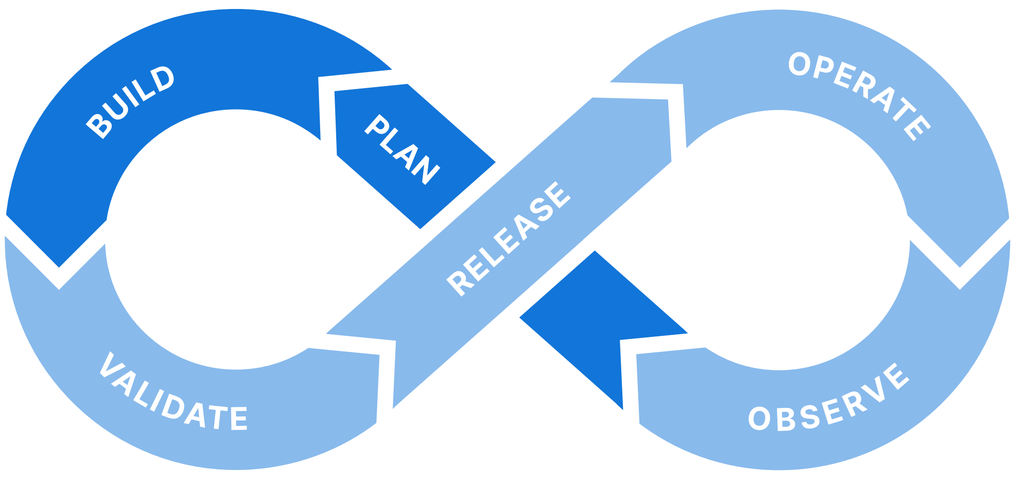
While Event Monitoring can uncover issues that didn't generate an exception email, it remains a reactive tool. Its event logs are pulled manually and only generated during off-peak hours the following day, unless you're using a paid edition that supports Real-Time Event Monitoring. This limits their usefulness for time-sensitive debugging of Apex and Flow failures. For faster resolution, it's still essential to have processes in place for triaging and responding to exception emails as they arrive.

To improve how quickly exceptions are diagnosed, teams can layer broader event monitoring across activities like deployments, metadata changes, and Flow faults. Capturing these critical signals closer to real time — through custom Platform Events — provides the missing context around why failures occur. Structured event data, consistently capturing who initiated an action, what changed, and what the outcome was, can help teams link exceptions back to recent system activity and reduce recovery time.

But while broader event monitoring provides visibility into the overall flow of system activity, it doesn't replace the need for detailed, transaction-level insight. That's where logging comes in.

Embed structured proactive logging

Observability doesn't begin at runtime. It starts in the ***plan*** and ***build*** phases of the DevOps lifecycle, where you decide what to log, where to log it, and how it will be used.



When you're starting from scratch, it's much easier to build observability into your DevOps lifecycle. You can plan what to log as you design your code or declarative functionality, identifying key decision points and outcomes as part of the build.

With existing applications, retrofitting structured logging requires identifying valuable logging points across a mature codebase and layering that functionality in. A more realistic approach is to embed structured logging into new work as it moves through the pipeline, while gradually addressing the technical debt in older areas over time.

A custom logging framework helps standardize this approach and close the visibility gap. By capturing runtime context like user input, decision paths, and API responses, structured logs allow you to reconstruct what led up to the failure — even when the exception itself doesn't contain much detail. This can be especially valuable for automated processes, such as Batch Apex or Scheduled Jobs, where no one is around to describe what went wrong.

Custom logging framework	Structured log
The system you build or install to standardize logging. It defines what to log, how to format it, and where to store it — often using custom objects or platform events. It enables consistent, scalable observability across your org.	The output of a logging framework — a machine-readable log entry (e.g. JSON) with consistent fields like timestamp, userId, and action. Easy to query, analyze, and integrate into alerts or dashboards.

Structured logs also support downstream alerting, analysis, and reporting. Tools like Nebula Logger and Pharos enhance this by capturing events asynchronously using platform events or storing logs in custom objects, making them queryable, persistent beyond standard log storage limits, and enabling rich alerting and reporting via native dashboards or external systems. Developer tools like the Apex Log Analyzer from Certinia — VS Code Marketplace extension — also give you more visibility into the debug logs and a cleaner view of the progress of the transaction. Although, as the Apex Log Analyzer works with log files downloaded locally, it isn't an easy option for shared access or team review — but you can still share the files manually if needed.

Debug logs and Event Monitoring — what's the difference?

Feature	Debug logs	Event Monitoring
Primary use case	Application-level debugging (Apex, Flows, validation rules, triggers).	Platform-level auditing and visibility (logins, API calls, report exports, etc.).
How it's activated	Requires setting a trace flag on a user, class, or process.	Always on (no manual activation required for standard events).
Log generation	Generated in real-time during user interactions or code execution, provided there's an active trace set up and within Salesforce governor limits.	Generated once daily, during off-peak hours.
Retention	Stored temporarily — expires automatically after 7 days (or 24 hours if created in Developer Console). When storage is full, logs stop generating until you manually delete existing ones.	Stored for 30 days by default (longer with paid add-ons).
Level of detail	Configurable and can be very detailed — includes stack traces, variable states, and execution flows.	High-level — metadata about events, but not full runtime context.
Configurability	Customizable log levels (e.g. DEBUG, INFO) for specific components.	Captures a fixed set of event types; not configurable per event.
Data format	Verbose text output (viewed in developer console or logs viewer).	Structured CSV files intended for external analysis.
Best for	Developers troubleshooting application logic.	Admins/security teams analyzing org usage and user behavior.
Limitations	Requires manual setup, prone to log loss if storage is exceeded.	Not real-time, no access to fine-grained execution state.

Governor limits

Blind spots	With observability
Governor limit breaches cause sudden, hard failures without warning.	Usage patterns are monitored to predict and prevent threshold breaches.
Failures are isolated events, difficult to connect to systemic issues.	Consumption trends are visible across transactions, helping teams plan scalable architecture.

Salesforce enforces strict limits on CPU time, queries, and transactions. Without proactive monitoring, these limits can quietly become performance bottlenecks.

Governor limits exist to ensure fair resource usage in the multi-tenant architecture of the platform. The challenge is that governor limit exceptions can't be caught in Apex. When a transaction exceeds a limit, Salesforce terminates it immediately, and no further code is executed — including any `try-catch` blocks or logging routines.

Some of the most common thresholds include:

- **CPU time:** 10,000 milliseconds per synchronous transaction (60,000 ms for asynchronous).
- **SOQL queries:** 100 per synchronous transaction (200 async).
- **DML operations:** 150 statements per transaction.
- **Records returned by SOQL:** Up to 50,000 total.
- **Records processed by DML:** 10,000 rows.
- **Heap size:** 6 MB synchronous, 12 MB asynchronous.
- **Callouts:** 100 per transaction.
- **Queueable chain depth:** Maximum of 50 jobs in a single chain.

Even the most sophisticated logging frameworks won't catch or log governor limit exceptions. That's why fallback mechanisms — like email alerts or the new free tier of Platform Event Monitoring — will always have a role in observability. There's always a risk of hitting those invisible ceilings.

You can think of governor limits like a fuel warning light: it's a good idea to trigger an alert when a transaction uses about 90% of any given limit. That way, you get an early heads-up and a chance to act before the hard ceiling is hit. Otherwise a transaction burning through 99% of its limits can look exactly the same as one that's barely using any.

Designing for governor limit resilience

Governor limit resilience means understanding how your org consumes resources and building in ways to anticipate, detect, and adapt to platform limits before they cause failures.

Design for data usage awareness

Avoiding governor limit breaches requires more than monitoring — it needs predictive awareness. This starts with understanding org-level consumption patterns: API calls, DML operations, storage, platform event deliveries.

Governor limits are designed so that standard transactions should stay within the thresholds. If your application consistently bumps into them, it's a sign the architecture is straining against the platform — often because it's processing more data than it was designed for. The earlier you can spot these stress points, the more options you have for resolving them safely.

While most governor limit breaches result in uncatchable hard failures, certain asynchronous contexts do offer limited recovery mechanisms. For example, in a Batch Apex job that implements the `RaisesPlatformEvents` interface, any unhandled exceptions will emit a `BatchApexErrorEvent`. These events can be subscribed to and used to trigger a remediation workflow — whether that's notifying a team, logging additional detail, or even retrying a process.

Similarly, Queueable Apex now supports the `Finalizer` interface, which allows you to define `TransactionFinalizer` methods. These run in a separate transaction — even if the main queueable fails — and can access the context of the failed job. While these options don't prevent governor limit exceptions, they can soften the impact by providing a structured response when something goes wrong.

Monitor resource usage

While governor limit breaches can't be caught or logged once they occur, structured logging plays a vital role in identifying risks before they escalate. Logging resource usage at key points in a transaction — such as CPU time consumed, heap size, SOQL query counts, and DML operation totals — gives you visibility into how close a process is getting to platform limits. This early insight makes it possible to detect when a business process is starting to strain under growing data volumes or architectural complexity.

For example, logging CPU usage checkpoints during a large batch job can highlight when synchronous processing should be re-architected as asynchronous. Tracking SOQL query counts inside loops can reveal inefficient patterns before they hit hard limits. Monitoring heap size trends can expose memory-heavy operations that risk heap overflow.

By embedding lightweight, structured logging into high-risk areas of the codebase, you can spot scaling challenges long before they appear as production failures — giving you time to rework logic, split processing, or optimize queries proactively.

Test at realistic data volumes

Testing at scale is an important part of a resilient delivery pipeline. Large data volumes are rarely tested thoroughly in typical environments, largely because it's complex and time-consuming to do so. But without realistic volume testing, teams risk discovering too late that a synchronous process needs to be re-architected as asynchronous — often after a governor limit breach has already impacted production.

Full-copy sandboxes are ideal for validating performance and behavior at scale. However, they're often repurposed for UAT, which can make them unsuitable for long-running volume tests. UAT environments tend to be active, shared, and filled with competing test activity — not the best setting for stress testing. A better strategy could be to seed a partial-copy sandbox with representative data volumes for day-to-day testing, preserving your full-copy environment for dedicated volume and regression validation.

Sandbox seeding

Seed sandboxes for realistic testing.

Seed test environments with production-like data sets — without the overhead or risk of cloning full environments every time.

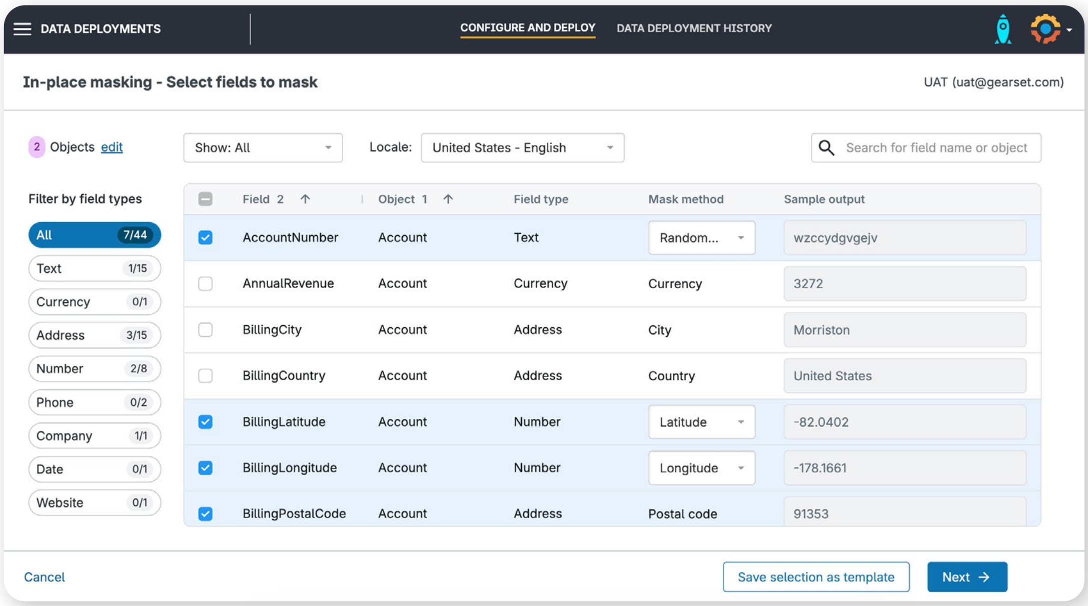
- Seed sandboxes with related records across multiple objects using pre-defined filters and relationship mapping.
- Mask sensitive data during deployment to protect personally identifiable or confidential information while still enabling realistic testing.
- Create reusable deployment templates, so you can quickly re-seed environments as needed or ensure consistency across multiple sandboxes.

Bridge the gap between test automation and real-world usage patterns, and uncover issues like governor limit risks before they reach production.

Try it free for 30 days

Kick off your observability adoption with a **free trial** of Gearset's Sandbox seeding solution.

START FREE TRIAL



In-place masking - Select fields to mask

UAT (uat@gearset.com)

2 Objects edit | Show: All | Locale: United States - English | Search for field name or object

Filter by field types

All	7/44
Text	1/15
Currency	0/1
Address	3/15
Number	2/8
Phone	0/2
Company	1/1
Date	0/1
Website	0/1

	Field 2	Object 1	Field type	Mask method	Sample output
✓	AccountNumber	Account	Text	Random...	wzccydgvgejv
	AnnualRevenue	Account	Currency	Currency	3272
	BillingCity	Account	Address	City	Morriston
	BillingCountry	Account	Address	Country	United States
✓	BillingLatitude	Account	Number	Latitude	-82.0402
✓	BillingLongitude	Account	Number	Longitude	-178.1661
✓	BillingPostalCode	Account	Address	Postal code	91353

Storage sprawl

Blind spots	With observability
• Storage growth is hidden until performance or costs are impacted.	• Data growth trends are tracked early, triggering proactive retention and scaling strategies.
• No easy way to detect fast-growing objects or integration issue.	• Object-level data insights help prioritize archiving, cleanups, and architectural changes.

Observability isn't just about catching runtime failures — it also plays a vital role in managing long-term risks like data bloat. While Salesforce enforces strict governor limits at runtime to protect system performance, storage usage tends to fly under the radar. But if left unchecked, storage usage can quietly grow out of control, leading to escalating costs, slower performance, and an increased risk of hitting org limits.

Storage growth might not cause a governor limit breach directly, but it's often the precursor. A process that once ran smoothly can start to strain — a query that used to return a few hundred records might now return tens of thousands. A synchronous job that was safe last year could suddenly run into heap size or CPU timeouts. The heap is the memory storage allocated to store objects and variables during code execution, while CPU time is the total processing time allowed for a transaction. As your data grows, so does the chance of hitting those limits.

Making storage sprawl visible

It isn't easy to monitor object-level storage in Salesforce, but getting visibility into your data growth is essential for keeping your org stable and scalable

Turn data growth into a signal

Governor limits don't cap how much data you can store — but they do control how much work your code can do. As your data grows, so does the effort required to retrieve, process, and respond to it. That raises the likelihood of hitting critical thresholds.

These limits are designed to protect platform stability, but they assume a certain scale of usage. If your data volume grows faster than your architecture adapts, your org can slowly become more fragile.

Tracking how your data grows over time gives you a better chance of spotting when a process needs to move from synchronous to asynchronous — or when an architectural change is needed. Even a manual process for capturing and trending object-level storage usage is better than having no visibility at all.

Uncover object-level storage gaps

Unfortunately, Salesforce doesn't offer a native API to retrieve object-level storage usage. Top-level storage totals are available via Apex and the REST API, but the object-level breakdown (as shown in the **Setup → Storage Usage** page) isn't accessible programmatically, other than using screen scraping. It's clunky, but for now, it's one of the only ways to get that level of insight.

Salesforce *does* expose current limit states through a REST API endpoint. By polling this regularly, you can build dashboards and trigger alerts that flag anomalies or thresholds that are about to be hit. Over time, this visibility helps establish baselines for expected integration behavior — so when a connected system or inbound integration suddenly pushes more data than usual, teams can investigate before limits are breached.

Knowing how much data exists in your org, how fast it's growing, and how close you are to predefined architectural thresholds gives product and engineering teams time to respond. Whether you're catching a query that's creeping up toward the 50k-row limit or identifying a batch process that's brushing against heap size, early visibility is the key to long-term stability.

Data dashboard

See storage trends in real-time.

Gearset's Data dashboard — included with all Backup licences — brings long-term visibility into how your Salesforce data is evolving. Unlike the static snapshot in the Salesforce Setup UI, the dashboard gives you a historical view of storage trends across your org.

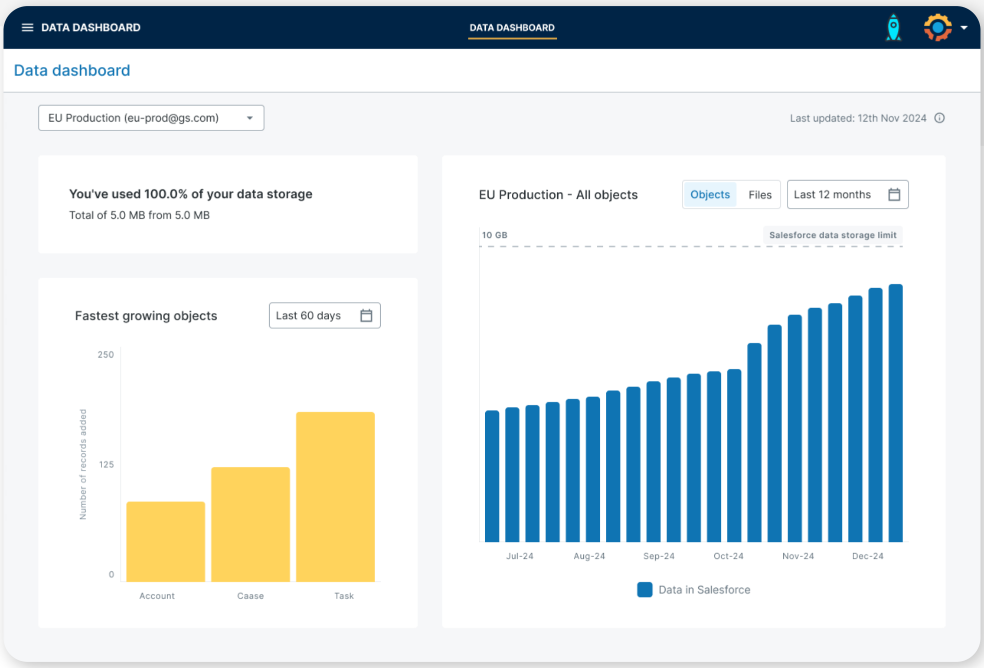
- Identify fast-growing objects
- Catch unexpected data spikes early
- Forecast when you might hit storage or volume-related thresholds

With this insight, teams can make smarter decisions about data retention — what to archive, what to delete, and what to keep. Proactive storage management reduces long-term performance overhead and helps maintain a healthier, more scalable Salesforce environment.

Try it free for 30 days

Kick off your observability adoption with a **free trial** of Gearset's backup solution.

START FREE TRIAL



You've used 100.0% of your data storage
Total of 5.0 MB from 5.0 MB

Implementing observability

Observability is a fundamental principle of DevOps done right and the result of deliberate design decisions made throughout the development lifecycle, capturing meaningful signals about system behavior in ways that are actionable, contextual, and scalable.

Core principles for a Salesforce observability framework

Start designing for visibility	Start designing for visibility early in your DevOps lifecycle. Integrate observability during the plan and build stages, embedding structured logging, error handling, and runtime context tracking into your code and configuration.
Capture meaningful, structured context	A log line is only useful if it tells you *why* something happened. Capturing inputs, outputs, and execution context turns logs into actionable diagnostics. Move beyond plain debug logs by adopting structured logging frameworks that make it easy to search, analyze, and trigger alerts from the data your org generates.
Treat exceptions as signals	Errors aren't just problems to patch, they're signals about how your system behaves under pressure. Surface failures clearly, categorize them by business process, and connect them back to what the user or system was trying to do at the time.
Track behavior over time	Observability isn't just about failures — it's about trends. Logging CPU time, heap usage, or SOQL rows over time builds a profile of what "normal" looks like, so you can catch slow degradations early.

Connect issues to change

When something breaks, look at what changed. Observability tools should help you connect errors to recent deployments or config updates, so you’re not left guessing which commit introduced a problem.

Plan for scale and failure

You can’t catch governor limit breaches — but you can see them coming. Logging usage, understanding limits, and using async recovery options like finalizers or Batch error events gives you more control when things go wrong.

Observability gaps: a checklist

Even the best maintained Salesforce orgs can have hidden blind spots. Without the right observability measures in place, issues can go unnoticed until they start causing real problems.

How do you know where your blind spots are? Here’s a quick observability health check to find out:

Change awareness

- [ ] Are you monitoring metadata changes across environments?
- [ ] Can you correlate recent deployments or config changes with new issues?

Exception handling

- [ ] Do you have a workflow for catching and triaging unhandled exceptions?
- [ ] Are caught exceptions logged and categorized by business process?
- [ ] Do Flow errors surface in tools your developers use — not just email inboxes?

Logging coverage & structure

- [ ] Are all critical events logged — not just errors?
- [ ] Do your logs capture input, output, execution context, and system state?
- [ ] Is your logging structured and queryable for easy analysis?
- [ ] Do developers consider what needs logging or monitoring as part of planning and design?

Governor limit awareness

- [ ] Can you see where your application is approaching platform limits like CPU time, heap size, or DML rows?
- [ ] Are you testing performance at realistic data volumes, not just happy-path test cases?

Storage visibility

- [ ] Do you know which objects are growing fastest in your org?
- [ ] Can you forecast future data trends and their potential impact on performance?
- [ ] Do you have a process for archiving or cleaning up data before it becomes a problem?

Next steps: integrating observability

Observability transforms Salesforce teams from reactive problem-solvers to proactive, high-performing teams. When your platform is emitting meaningful, structured data about what it's doing and how it's behaving, you get a feedback loop that actually helps you build better software.

You can't fix what you can't see — and without proper observability, you're left reacting to issues long after they've impacted users. Whether it's governor limit breaches, silent performance degradation, or bugs that only show up in edge cases, the cost of flying blind is real: downtime, frustrated users, and firefighting that slows everything down.

That lack of visibility breaks the feedback loop that good DevOps depends on. Without clear signals from production, it's harder to learn from issues, harder to ship with confidence, and harder to know whether changes are improving things or making them worse. The techniques in this whitepaper are designed to close that loop by layering multiple practices that work together to make your system more observable, debuggable, and resilient.

The good news? Visibility gaps don't have to be your default. Salesforce *can* be observable — and it starts by embedding logging, error handling, and execution context into how you build. The payoff is a faster, more confident DevOps process with fewer surprises and much better data to make decisions.

At Gearset, we work with hundreds of teams who are building observability into their pipelines every day — and we'd love to help you do the same.

Arrange a tailored demo with our DevOps experts

Kick off your observability process with a 15-minute call to talk through where your current gaps are, what your org needs, and how to build observability into your workflow.

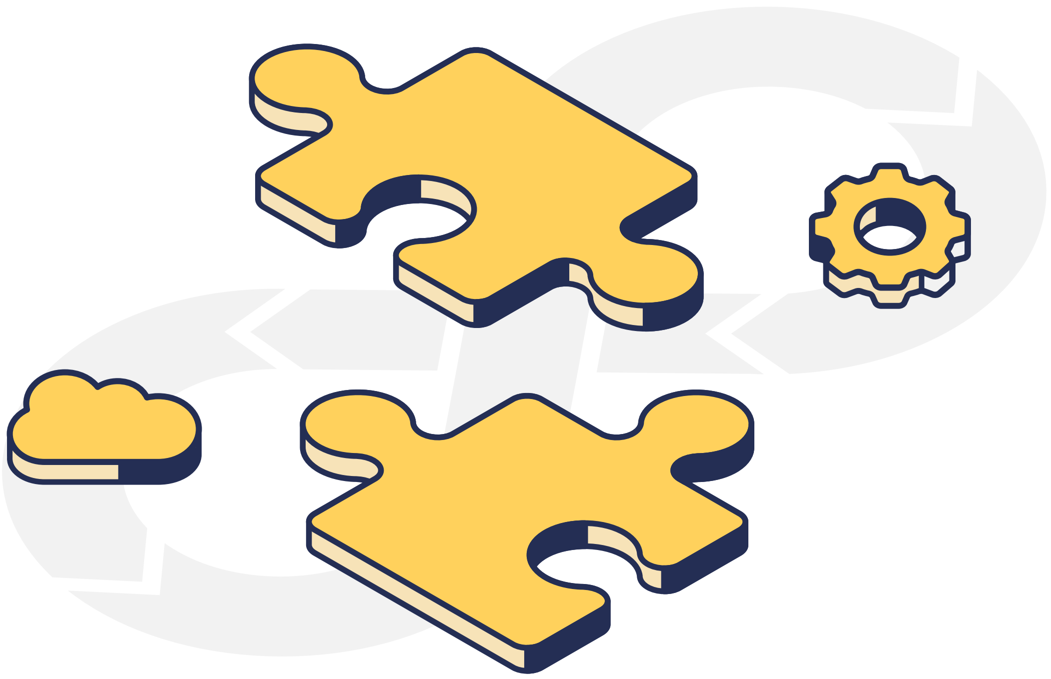
BOOK A DEMO →

About Gearset

Gearset is the complete Salesforce DevOps platform, enabling teams to implement best practices throughout the entire DevOps lifecycle.

With powerful solutions for metadata and CPQ deployments, CI/CD, testing, code scanning, sandbox seeding, backups, archiving and observability, Gearset offers teams unparalleled visibility and control over their Salesforce process.

More than 3,000 enterprises, including McKesson and IBM, use Gearset to accelerate development, improve release quality, enhance security, and make Salesforce deliver.



Gearset