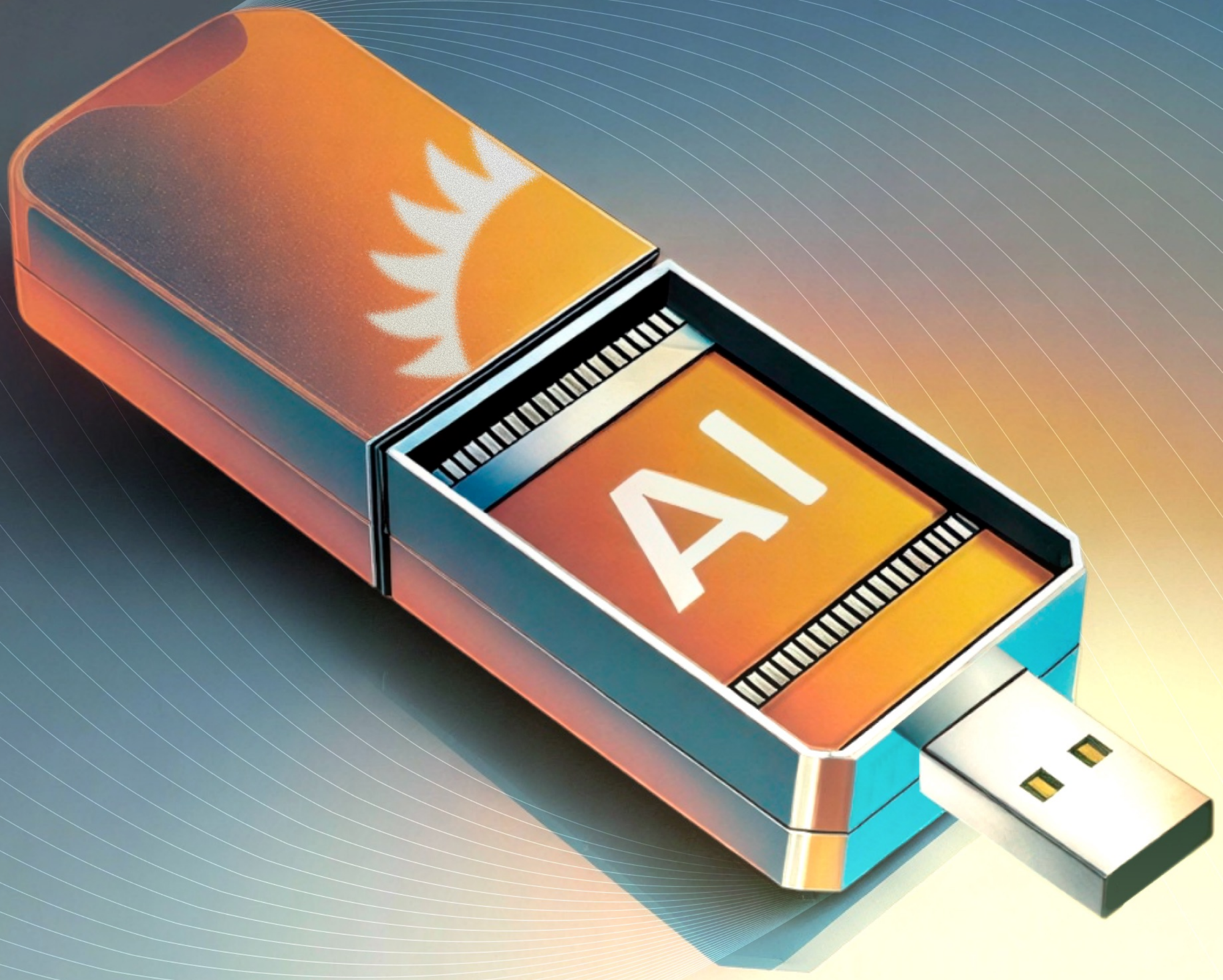
ANDERSEN




# AI-Assisted Operations for Platform Teams

## From Alert to Fix — A Practical Guide for DevOps, SRE, and Platform Engineering Leaders

# Executive Summary

Platform engineering and SRE teams today operate at a level of infrastructure complexity that their current tooling was never designed to handle. The average organization runs workloads across multiple cloud providers, maintains dozens of CI/CD pipelines, tracks incidents in one system, documents runbooks in another, and stores architecture knowledge in a third. When something breaks — and it always breaks — engineers spend more time assembling context than actually fixing the problem.

AI changes this equation. Not by replacing engineers, but by giving them instant access to the full picture: live resource state, logs, metrics, runbooks, tickets, and code — all at once, in natural language.

This guide is written for two audiences: engineering leaders (CTOs, VPs of Engineering) evaluating whether AI-assisted operations is the right investment, and platform leads (Heads of Platform, SRE Leads) who want to understand how to implement it practically and without risk.

---

## By the end of this document you will understand:

- Why the fragmented toolchain is the root cause of slow incident response and high operational overhead
- How to implement it in phases, starting with zero disruption to your existing stack
- What the most common mistakes are, and how to avoid them
- What AI-assisted operations actually means in a DevOps context — and what it does not mean
- How three composite use cases map to real engineering workflows

# Context: The Platform Engineering Moment

Platform engineering has moved from niche discipline to organizational imperative. Gartner projects that by 2026, 80% of large engineering organizations will have established platform engineering teams — up from under 30% in 2022. The driver is not ideology: it is economics. Infrastructure complexity has grown faster than team headcount, and the resulting operational overhead has become measurable in outage frequency, MTTR, and engineer attrition.

---

## Three structural shifts define the current moment:

### 1. The multi-cloud default

Organizations no longer choose between AWS and Azure — they run both, plus on-premise, plus managed services from a dozen vendors. The average enterprise runs 2.6 public cloud providers (Flexera, 2024). Each has its own APIs, monitoring paradigms, and operational interfaces.

### 2. The toolchain explosion

A typical platform team maintains integrations with cloud consoles, a CI/CD platform (GitHub Actions, GitLab CI, or Jenkins), an IaC tool (Terraform, Pulumi), a monitoring stack (Datadog, Grafana, CloudWatch), a knowledge base (Confluence), and a ticket tracker (Jira or ServiceNow). These systems do not talk to each other. Context lives in human heads.

### 3. The talent constraint

Hiring senior SREs and platform engineers is expensive and slow. The demand for cloud infrastructure expertise continues to outpace supply. Organizations cannot hire their way out of operational complexity — they need to make existing engineers more effective.

AI-assisted operations is a response to all three. It is not a replacement for engineers — it is a multiplier for them.

# The Problem: Context Lives Everywhere, Context Is Available Nowhere

The most expensive moment in platform operations is not the outage. It is the first 20 minutes after the alert fires — when engineers are scrambling to understand what changed, what is affected, and where to look.

## In a typical incident war room, the following happens simultaneously:

This is not a process failure. It is a structural problem: the context needed to diagnose the issue is distributed across five systems with no shared interface.

- One engineer checks CloudWatch or Datadog for metrics
- Another pulls recent deployment history from GitLab or GitHub
- A third searches Confluence for the relevant runbook — often finding it outdated or missing
- Someone opens Jira to check if a similar issue was reported before
- A Slack thread accumulates competing hypotheses

> **“Mean Time to Resolution (MTTR) in the industry averages 4+ hours for PI incidents. Studies consistently show that 60–70% of that time is spent on investigation and diagnosis, not the fix itself”**

The same fragmentation creates a second, less visible problem: tribal knowledge. When your most experienced SRE knows that the payment service always lags after a certain kind of Terraform apply, that knowledge exists only in their head. When they are unavailable — or when they leave — it disappears. Runbooks capture some of this, but they age quickly and are rarely kept current under operational pressure.

CI/CD adds another layer. Debugging a failed pipeline today requires an engineer to open the pipeline UI, read raw logs, cross-reference the repository state, and manually trace which step failed and why. For complex pipelines with 40+ stages, this can take an hour of focused work — work that a connected AI assistant can do in seconds.

# Why Existing Tools Don't Solve This

The obvious response to the context problem is: buy more tooling. Observability platforms, AIOps tools, runbook automation — the market is full of solutions. The problem is that each of them solves one layer of the stack in isolation.

| Tool | What it covers — and what it misses |
|---|---|
| **Observability platforms (Datadog, Grafana)** | Excellent at surfacing anomalies and metrics. Do not know your runbooks, your Jira history, or your IaC state. Cannot tell you whether this alert was caused by last week's Terraform change. |
| **AIOps tools** | Correlate alerts and reduce noise. Operate on telemetry data only. Do not understand your architecture docs, your code, or your incident history stored in Jira. |
| **Runbook automation** | Executes predefined remediation steps. Breaks when the situation does not match the runbook. Cannot adapt to context it was not programmed to handle. |
| **ChatOps integrations (Slackbots, etc.)** | Aggregate notifications in one channel. Do not diagnose. Do not access live resource state. Do not synthesize context from multiple sources. |
| **Internal wikis (Confluence)** | Store accumulated knowledge. Become outdated. Are searched manually. Have no awareness of current infrastructure state. |

The gap is not in any individual tool. It is in the space between tools — the integration layer that no single vendor owns. An AI assistant with access to your full toolchain fills exactly that gap. It does not replace your observability stack or your CI/CD platform. It sits above them, connecting them into a unified interface.

# Key Insights: What AI-Assisted Operations Actually Changes

Based on practical implementation patterns across DevOps and platform engineering teams, four insights define what makes AI-assisted operations effective versus performative.

## Insight 1:

### Live context beats documentation

The standard AI assistant for DevOps works on static documents — runbooks, architecture diagrams, past tickets. This is useful, but it breaks down the moment the live state diverges from what is documented (which is most of the time). The step change comes when the assistant queries

## Insight 2:

### The toolchain must be unified, not just connected

Connecting five tools to an AI interface is not the same as unifying them. Unification means the assistant can reason across all of them simultaneously: it can see that the CloudWatch alarm fired three minutes after a GitLab pipeline deployed a new container image, and that a Jira ticket from six weeks ago describes an identical symptom pattern. This cross-source reasoning is what compresses investigation time from hours to minutes.

## Insight 3:

### Read-only by default is not a limitation — it is the right default

Platform teams evaluating AI assistants often ask about automation: can it fix things, not just diagnose them? The right answer is: it can, but you should implement read-only first. Read-only deployment gives teams immediate value with zero risk. It builds trust. It lets engineers verify that the assistant's diagnoses are accurate before they trust it to act. Mutating actions — restarting services, applying Terraform changes, updating Jira tickets — should be gated behind explicit human approval, not automated away.

## Insight 4:

### The CLI and chat interfaces serve different workflows

The standard AI assistant for DevOps works on static documents — runbooks, architecture diagrams, past tickets. This is useful, but it breaks down the moment the live state diverges from what is documented (which is most of the time). The step change comes when the assistant queries

# Introducing the Self-Service DevOps Assistant (SDA)

SDA is a cloud-agnostic AI assistant built for DevOps, SRE, and platform engineering teams. It connects your cloud infrastructure, CI/CD pipelines, issue trackers, code repositories, and knowledge base into a single interface — and answers engineering questions in natural language, based on what is actually happening in your environment right now.

Engineers interact with SDA from the CLI or Microsoft Teams. They ask questions the same way they would ask a senior colleague: “What changed in the payment service in the last two hours?” or “Why did this pipeline fail and has it happened before?” SDA queries live sources, cross-references them with your internal documentation and ticket history, and returns a diagnosis with evidence — not a generic answer.

## Key Capabilities

Infrastructure troubleshooting across AWS and Azure — queries live resource state, logs, and metrics in real time

Incident investigation and root cause analysis — correlates cloud events, recent deployments, and ticket history

CI/CD pipeline creation, analysis, and debugging — reads pipeline logs and repository context to identify and explain failures

Jira / ITSM ticket analysis — surfaces related tickets, patterns, and recommended resolutions based on historical data

Infrastructure-as-Code support including drift detection — compares desired state against live infrastructure

Knowledge-base search across Confluence, runbooks, and repositories — retrieves relevant documentation in the context of the current issue

Extensible agent architecture via MCP integrations — connect additional tools as your stack evolves

CLI and Microsoft Teams interfaces — works where your engineers already work

# What Makes SDA Different

| Differentiator | What it means in practice |
|---|---|
| Live infrastructure context | SDA queries your cloud APIs, logs, metrics, and resource state in real time — not from a snapshot or a cached index. The answer reflects what is true now. |
| Answers from your knowledge base | SDA uses your Confluence docs, Git repositories, runbooks, and architecture materials. It does not give generic answers — it gives answers grounded in your environment. |
| Full DevOps toolchain in one assistant | Cloud, Jira, GitLab, GitHub, and Confluence connected in a single interface. No context switching. No manually assembling information from five tabs. |
| Cloud-agnostic architecture | Supports AWS and Azure today, with an extensible architecture that accommodates additional platforms as your infrastructure evolves. |
| Safe by default | SDA runs in read-only mode out of the box. Any mutating action — restarting a service, applying a Terraform change, updating a ticket — requires explicit human approval. Trust is built before automation is extended. |

# How SDA Works

SDA does not require replacing your existing toolchain. It connects to what you already run — cloud accounts, CI/CD pipelines, Jira, Git repositories, Confluence — and creates a unified context layer on top of them. Engineers interact through CLI or Microsoft Teams using natural language. SDA handles the querying, cross-referencing, and synthesis.

## Step 1

### Connect your sources

SDA integrates with your existing stack via standard APIs. A typical initial connection covers your primary cloud provider (AWS or Azure), your Git repositories (GitHub or GitLab), Jira or your ITSM tool, and Confluence. No agents to install on production infrastructure. No changes to existing pipelines. Connection is read-only by default.

## Step 2

### Ask in natural language

Once connected, engineers interact with SDA from the CLI or Microsoft Teams — wherever they already work. Questions are plain English: “What changed in the last 4 hours on the payments service?” or “Why did this pipeline fail and has it happened before?” SDA queries live sources in real time, not a cached index.

## Step 3

### Get evidence-based answers

SDA returns a diagnosis with sources — not a generic suggestion. It tells you which CloudWatch alarm fired, which Terraform change preceded it, which Jira ticket described the same pattern six weeks ago, and what the resolution was. Engineers validate and act. If a change is required, SDA can propose it — but executes only with explicit human approval.

## Step 4

### Extend as trust grows

After the read-only phase, teams can enable SDA to take approved mutating actions: restarting a service, creating a Jira ticket, triggering a rollback pipeline. Each action type is explicitly enabled by the team and requires approval at runtime. The architecture is extensible via MCP integrations — additional tools and data sources can be added as your stack evolves.

## What a first day with SDA looks like:

### Morning

On-call engineer asks SDA to summarize overnight alerts and flag anything that correlates with recent deployments. Gets a 30-second briefing instead of opening five dashboards.

### Afternoon

Pipeline fails. Engineer asks SDA to diagnose. Gets root cause, relevant ticket history, and a suggested fix in under two minutes.

### End of day

Team lead asks SDA to surface Jira tickets linked to services with elevated error rates. Backlog grooming prep done in minutes, not hours.

# Common Mistakes — and What's Coming Next

The obvious response to the context problem is: buy more tooling. Observability platforms, AIOps tools, runbook automation — the market is full of solutions. The problem is that each of them solves one layer of the stack in isolation.

## The five mistakes teams make with AI-assisted ops

### 1. Connecting everything before measuring anything

Teams integrate all tools in week one and then cannot tell which connections are providing value. Start with two or three sources, establish a baseline, then expand.

### 2. Treating the knowledge base as a solved problem

The assistant will only be as good as the documentation it has access to. If your Confluence is full of outdated runbooks and missing architecture docs, the assistant will produce inaccurate answers. Use early deployment gaps to identify and fix documentation debt.

### 3. Skipping the read-only phase

Teams eager to automate remediation skip the diagnostic phase and immediately build automation. This backfires. Build trust in the assistant's diagnoses before trusting its actions.

### 4. Deploying for engineers but not for incidents

Some teams deploy the assistant as a background tool for exploration, but do not integrate it into the actual on-call workflow. Incident response is where the time savings are largest. Integrate it into your runbooks from day one.

### 5. Expecting out-of-the-box accuracy on domain-specific questions

Generic AI assistants will give generic answers. The value of a connected assistant comes from project-specific context: your architecture, your naming conventions, your incident history. Budget time to tune prompts and verify early outputs.

## What is coming next

The current generation of AI-assisted ops tools is primarily reactive — engineers ask questions, assistants respond. The next phase is predictive: assistants that monitor infrastructure state continuously and surface anomalies before they become incidents.

Early signals already point in this direction: drift detection that flags Terraform state divergence before deployment, CI/CD pattern analysis that identifies flaky tests before they block releases, ticket classification that routes new incidents to the correct team automatically based on historical patterns.

Teams that implement read-only AI-assisted operations today are building the foundation — trusted data connections, quality knowledge bases, team familiarity with AI-assisted workflows — that will make predictive operations practical in 12–18 months.

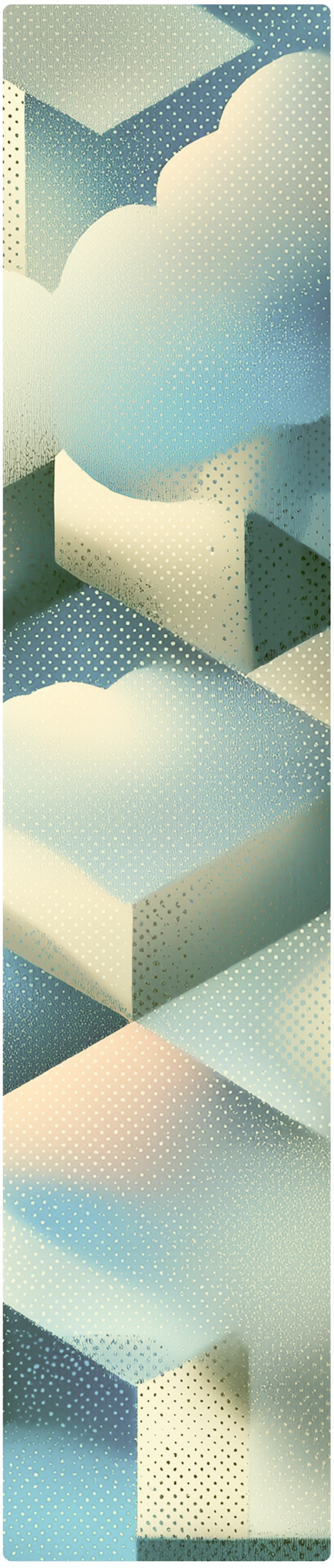
# Use Cases

The following three scenarios are composite illustrations based on common platform engineering workflows. They are representative of the types of problems AI-assisted operations solves, rather than specific client engagements.

## Use Case 1

### Incident Investigation — Payment Service Latency Spike

| Context | Before | After |
|---|---|---|
| Team | 6-person SRE team, AWSnative stack, Jira for incidents, Confluence for runbooks | Same team and stack |
| Trigger | P1 alert: payment service p99 latency > 3s | Same alert |
| Investigation | On-call engineer spends 35 min checking CloudWatch, recent deployments, Jira tickets, and Confluence manually. Escalates to two additional engineers. | Engineer asks assistant: 'What changed in payment service infrastructure in the last 4 hours and is there a related Jira history?' Assistant returns: RDS connection pool configuration change in last Terraform apply, linked to similar ticket from 3 months ago with resolution steps. |
| Time to diagnosis | ~50 minutes | ~8 minutes |
| Resolution | Manual config rollback | Engineer validates assistant's recommendation, applies rollback with approval gate |



## Use Case 2

# CI/CD Pipeline Failure — E-commerce Deployment Blocked

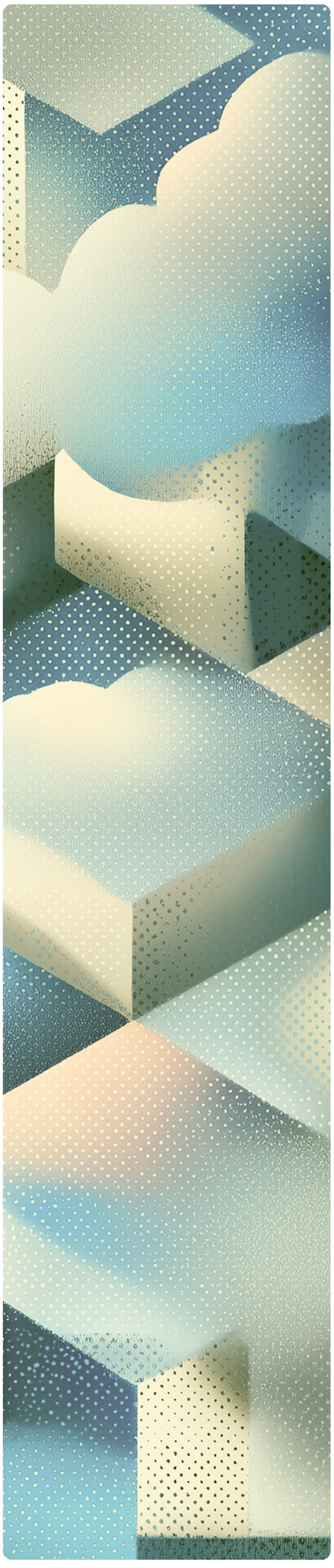
A 12-stage GitLab CI pipeline fails at the integration test stage, blocking a release. The error log contains a cryptic container networking error that does not appear in the team's runbooks.

Without AI assistance: engineer reads 400 lines of raw pipeline logs, searches Confluence for similar errors, finds nothing, opens a ticket, waits for a second opinion. Resolution takes 90 minutes.

With AI assistance: engineer asks the assistant to analyze the failed pipeline run. The assistant identifies that the networking error pattern matches a known Docker overlay network issue that appeared in a closed GitHub issue in the project repository three months prior — a fix that was never documented in Confluence. It returns the fix and a recommendation to add the resolution to the knowledge base.

### Key pattern

the assistant found institutional knowledge that existed in the repository but had never been promoted to documentation. This is one of the most consistent patterns across CI/CD use cases — the answer exists somewhere in the toolchain, but no one knows where to look.



## Use Case 3

# Jira Ticket Analysis — Platform Team Backlog Review

A platform team of 8 manages a backlog of 180+ open Jira tickets. The team lead spends 3–4 hours each sprint preparing for backlog grooming: reading tickets, identifying duplicates, grouping related
issues, and surfacing patterns.

With AI assistance: the team lead asks the assistant to analyze the current backlog and return: (1) clusters of related tickets, (2) tickets that appear to describe the same underlying issue, (3) tickets that reference infrastructure components currently showing elevated error rates in CloudWatch.

The assistant cross-references Jira with live CloudWatch data and returns a structured summary in 90 seconds. Backlog grooming preparation drops from 3 hours to 20 minutes. More importantly, the cross-reference with live infrastructure state surfaces two tickets that had been deprioritized but describe symptoms of an active, low-severity issue — allowing the team to address it before it escalates.

# See SDA in Action

The fastest way to understand what SDA can do for your team is to see it working against a real engineering scenario — live infrastructure, a failed pipeline, or an open incident. We run a focused 45-minute demo tailored to your stack and the problems you are actually dealing with.

In the demo we will connect SDA to a representative environment, walk through a realistic incident or pipeline failure, and show you exactly what the assistant surfaces — and how fast. No slides. No generic product walkthrough.

## What to expect from the demo

- 45 minutes, tailored to your stack (AWS / Azure, GitLab / GitHub, Jira / ServiceNow)
- Live walkthrough of incident investigation, CI/CD debugging, or ticket analysis — your choice
- Q&A with Andersen's platform engineering team
- A clear picture of what integration would look like for your environment

---

**Book a demo**

**Email: info@andersenlab.com**

ANDERSEN

# Roman Gorge

Head of Cloud | AI & Data | ex-AWS

Linkedin

# Make your project outcomes predictable!