Finding Solutions for Cloud Engineering Problems

Prepared by: Ben Leyder

Executive Summary

The purpose of this report is to identify the key challenges faced by a small cloud engineering team and to propose practical solutions to each. While these challenges can affect engineering teams of any size, their impact is felt most acutely by smaller teams operating with limited personnel and budgetary resources — where every inefficiency carries an outsized cost.

Five core challenges have been identified for examination. Data source variability refers to the inconsistency of file formats used across the team's systems, which creates complexity and risk in data processing pipelines. Proactive monitoring and observability addresses how the team tracks the performance and health of its cloud infrastructure to ensure reliability and uninterrupted service for customers. Resource constraints encompasses both the limitations of a small team size and the budgetary realities of a smaller organization, where the burden of work can quickly outpace the capacity of the people responsible for it. Documentation and guidance speaks to the critical practice of recording how infrastructure and code works and why decisions were made, ensuring that knowledge is preserved and accessible when systems are revisited in the future. Finally, the integration of CI/CD — Continuous Integration and Continuous Deployment — examines how adopting automated deployment practices can streamline the delivery of software updates and reduce the manual burden on the team.

Each of these areas represents both a challenge and an opportunity. Addressing them thoughtfully and systematically will position the team to work more efficiently, deliver more reliably, and build a stronger foundation for growth.

Problem 1: Data Source Variability

The team is using different data sources that have various formats. This can cause problems with data quality and make it hard to process the data correctly.

Analysis

Establishing a standard data format across all sources eliminates a wide range of downstream problems. Different formats require different dependencies to be read and interpreted correctly, and each additional dependency introduces another potential point of failure in the pipeline. Beyond reliability, data quality itself suffers — formats represent values like dates, times, and phone numbers differently, making consistent validation difficult. Much like diplomats at the United Nations relying on interpreters who may not have direct translations for every word, pipelines forced to interpret multiple formats are prone to misunderstanding and error. A single agreed-upon format removes the need for that interpretation layer entirely.

Proposed Standard: JSON

For a cloud engineering environment without a highly specific use case, JSON is the most logical choice for a standardized data format. It is readable by both humans and machines, making it accessible for developers at all levels. Most developers are already familiar with JSON, lowering the barrier to adoption across teams. Perhaps most importantly, nearly all AWS services support JSON natively, which significantly reduces dependency complexity — only a small number of services would require any transformation layer in the pipeline.

Convincing other teams to adopt this standard would rely on these same arguments. Demonstrating that JSON reduces their own maintenance burden, is already familiar to their developers, and aligns with how AWS services communicate natively makes a compelling case. The goal is to frame standardization not as an inconvenience imposed on them, but as something that makes their own work easier.

Communication and Stakeholder Strategy

Standardizing data formats across all teams is critical to ensuring a consistent and efficient workflow. A straightforward example is date formatting — the United States uses MM-DD-YYYY while many other countries use DD-MM-YYYY. For the first twelve days of any month, these two standards are indistinguishable from one another, creating genuine ambiguity in data quality and verification. When teams operate on different formats, this kind of confusion compounds across every data interaction.

Beyond data quality, format inconsistency creates friction in communication and collaboration between teams. A single data standard removes the need for interpretation layers between formats, reducing dependency complexity and eliminating unnecessary points of failure. Fewer points of failure means improved reliability, reduced risk of service downtime, and less labor spent troubleshooting avoidable problems.

Building strong relationships with data source teams is essential to making this standard stick. The most effective approach is to frame standardization not as a mandate, but as a mutual benefit — one that saves every team time, reduces their own troubleshooting burden, and ultimately protects the company's most valuable resources. When stakeholders understand that this decision translates directly to saved time and money, the conversation shifts from resistance to collaboration.

Problem 2: Proactive Monitoring and Observability

The team currently waits for problems to happen before fixing them. This can lead to longer outages and inefficiencies.

Key Metrics and the Case for Early Detection

Identifying the right metrics is the foundation of any proactive monitoring strategy. For AWS cloud systems, key health indicators include CPU utilization and memory usage on compute resources like EC2 instances, storage capacity on services like RDS and S3, network traffic patterns, error rates, latency and response times, and overall service availability. Each of these signals can indicate a system under stress before it reaches a breaking point — an overloaded EC2 instance, a database running out of storage, or rising error rates are all warnings that something is wrong before something actually fails.

A reactive approach to system monitoring — waiting for problems to occur before addressing them — carries significant costs that extend well beyond the initial incident. Much like neglecting preventive maintenance on a vehicle, the consequences of waiting for something to break are almost always more severe and expensive than catching the issue early. A timing belt that goes uninspected may eventually snap, destroying the engine entirely and leaving the driver stranded — what could have been a minor scheduled repair becomes a major unplanned expense.

In a cloud engineering context, the stakes are equally real. The window of time between when a problem begins and when someone notices it can produce a range of damaging outcomes. Customers lose access to services they are paying for, which erodes trust and risks losing their business entirely. Minor issues that go undetected can quietly compound, creating frustration that builds over time before becoming visible. Perhaps most critically, failures in distributed systems frequently trigger cascading failures downstream — meaning the original problem may be compounded by secondary failures that must each be identified and resolved independently. The team then spends time not only fixing the root cause, but untangling everything that broke as a result. A proactive monitoring approach interrupts this chain before it begins, protecting the company's reliability, reputation, and revenue.

Recommended Monitoring Strategy

Active monitoring of system health is critically important to ensuring reliability and uninterrupted customer access to services. AWS provides a suite of tools designed specifically for this purpose, and implementing them together creates a comprehensive proactive monitoring strategy.

CloudWatch serves as the foundation, collecting metrics, logs, and events across the entire infrastructure. This data provides both real-time visibility and a historical record that is invaluable when diagnosing issues. Building on this foundation, CloudWatch Alarms can be configured to trigger whenever a monitored asset operates beyond defined healthy parameters — CPU utilization thresholds, storage capacity limits, error rate spikes, or latency increases. When an alarm triggers, AWS SNS (Simple Notification Service) immediately notifies the team via email or SMS, giving engineers the opportunity to diagnose and address the problem before it escalates into a larger failure.

CloudTrail adds an additional layer of visibility by logging all API activity across the environment, creating an audit trail of every change made to the system and by whom. This is particularly valuable for identifying whether a configuration change contributed to a developing issue. Finally, AWS Trusted Advisor rounds out the strategy by proactively scanning the environment for

performance, security, and cost concerns — functioning like a scheduled inspection rather than a reactive alarm. Together these tools shift the team from a reactive posture to a proactive one, catching smaller problems before they become larger, more expensive ones.

Continuous Improvement and Feedback

Having the right monitoring tools in place is only the beginning — maintaining their effectiveness over time requires a deliberate feedback process. A regular review cadence should be established where designated team members periodically examine the data collected by CloudWatch, review the activity logs within CloudTrail, and consult the audit findings from AWS Trusted Advisor. The frequency and staffing of these reviews would scale with the size and complexity of the system.

Automated monitoring is a powerful asset, but it is not a complete substitute for human oversight. Automated systems excel at catching patterns and threshold breaches that a human might miss, but experienced engineers bring contextual judgment and intuition that no automated tool can fully replicate. The most resilient monitoring strategy treats human review and automated alerting as complementary layers rather than alternatives to one another.

Over time this review cycle creates a continuous improvement loop. Each review is an opportunity to assess whether existing alarms are calibrated correctly, whether new metrics should be added, and whether any gaps in coverage have emerged as the system evolves. This iterative process ensures the monitoring strategy remains effective as the infrastructure grows, ultimately protecting the team from lost time and revenue.

Problem 3: Resource Constraints

The team is small, so members often have to do many tasks. This can lead to stress and gaps in coverage.

Assessment of Team Roles and Risks

Operating with a small cloud engineering team inevitably means team members carry multiple responsibilities simultaneously. This distribution of workload is rarely perfectly equal, and the reality of keeping operations running often means long hours, late nights, and weekend work. Beyond the immediate inconvenience, this kind of sustained pressure has serious consequences for both the individuals involved and the organization as a whole.

For team members, an imbalanced workload erodes work-life balance and decreases job satisfaction over time. This is particularly acute in smaller organizations or startups where budgets are tight, compensation may not reflect the additional burden being carried, and hiring additional staff to provide relief is not immediately feasible. When people feel overworked and underappreciated, retention becomes a genuine problem — and losing an experienced engineer on an already small team compounds the pressure on everyone who remains.

The impact on work quality is equally significant. An overloaded team operating under chronic stress is more likely to make mistakes, cut corners, or miss warning signs. More errors mean more troubleshooting, more diagnostics, and ultimately more work to correct problems that might have been avoided under better conditions. This creates a negative feedback loop — stress reduces quality, reduced quality creates more problems, more problems increase stress — that left unaddressed can escalate into a catastrophic failure of both the system and the team. Recognizing and addressing these pressures early is essential to breaking that cycle.

Resource Management Recommendations

When a small team faces more work than it has capacity to handle, creative resource management becomes essential. The first step is establishing clear task prioritization — not all work carries equal weight, and identifying which responsibilities are most critical ensures the team's limited bandwidth is directed where it matters most.

Cross-training team members across all major responsibilities is equally important. In a small team, single points of knowledge are as dangerous as single points of failure in infrastructure. If only one person knows how to perform a critical task and they are unavailable — whether due to time constraints, illness, or a family emergency — the entire team is left without coverage. Just as cloud infrastructure is designed with fault tolerance and redundancy to ensure continuity when individual components fail, a well-managed team should be built with the same principles in mind. Every significant responsibility should have at least one other team member capable of covering it.

Cross-training also creates an unexpected additional benefit — the ability to perform meaningful quality assurance. When multiple team members understand a given task, they can review each other's work before deployment, catching errors that the original engineer might have missed. This peer review layer adds resilience to the team's output and reduces the likelihood of costly mistakes reaching production. Distributing knowledge fairly across the team not only lightens individual burdens, it makes the entire team more capable, more adaptable, and more reliable.

Automation Opportunities

Automation presents one of the most impactful opportunities for a small team to reclaim time and reduce workload without adding headcount. Several key areas of cloud engineering are well suited to automation using industry standard tools.

Infrastructure deployment is perhaps the most significant opportunity. Rather than manually configuring and deploying each component through the AWS console — a time consuming and error prone process — Terraform allows the team to define the entire infrastructure as code and deploy it in minutes. This not only saves time but ensures consistency across deployments, eliminating the variability that comes with manual configuration.

For complex architectures that experience fluctuating demand, Kubernetes provides automated scaling, adjusting resources dynamically in response to real time usage without requiring manual intervention. Ansible complements this by automating the configuration of those resources and the installation of necessary software — the team simply writes inventory and playbook files and runs them to execute an entire sequence of tasks automatically.

Finally, Jenkins can be used to establish a complete CI/CD pipeline, automating the process of pushing updates to cloud assets rather than requiring manual deployment every time a change is made. Together these tools dramatically reduce the volume of repetitive, predictable work the team must perform manually.

That said, automation is not a replacement for human judgment. Incident diagnosis, architecture design, security evaluation, and stakeholder communication all require contextual thinking and decision making that automated tools cannot replicate. The goal is to automate everything that follows predictable rules so that the team's time and energy can be reserved for the work that genuinely requires a human touch.

Problem 4: Lack of Documentation and Guidance

Not having enough documentation can make it hard for new members to onboard and slow down the work process.

Impact of Poor Documentation

The impact of poor documentation on team performance becomes immediately apparent when working through real world deployments. In one firsthand experience working on a small four person cloud engineering team tasked with developing and deploying a new system architecture for a cloud migration, the absence of adequate documentation created significant obstacles. While the team had clearly defined individual roles, the available documentation was either outdated or failed to translate meaningfully to the new environment. The result was several hours of troubleshooting and diagnostics that could have been avoided entirely with accurate, current reference material.

This experience illustrates a broader truth about documentation in IT environments — its absence does not simply create inconvenience, it actively costs the company and its customers time and money. Every hour spent diagnosing a problem that documented guidance could have prevented is an hour not spent on productive work. For new team members onboarding into an environment without sufficient documentation, the challenge is even greater — they lack the institutional knowledge that experienced team members have accumulated over time, leaving them without a foundation to work from. Good documentation bridges that gap, accelerating onboarding, reducing errors, and ensuring that hard won knowledge is preserved and accessible to the entire team rather than siloed in any one person's memory.

Documentation Improvement Plan

A successful documentation strategy rests on three pillars: establishing a clear standard, keeping documentation current, and enforcing quality through peer review.

The need for standardized documentation becomes obvious when considering the reality of how institutional knowledge erodes over time. An engineer who performs an initial configuration may remember exactly what they did and why the following week, but six months later those details will have faded. If that engineer has moved on to another role or another company entirely, their successor inherits a system with no record of the decisions that shaped it. Without documentation, every handoff becomes a troubleshooting exercise.

Keeping documentation current is one of the most persistent challenges in the industry. Without an enforced standard, documentation quality varies widely between team members — some may document thoroughly while others omit details or neglect to update records after minor changes. A clearly defined documentation standard removes that ambiguity by establishing consistent expectations for what must be recorded, how it should be structured, and when it must be updated. Every change to a system, however minor, should be reflected in the documentation.

The third pillar is peer review. Just as developers submit code for colleague review before deployment, documentation should undergo the same scrutiny. A second set of eyes ensures the standard is being upheld, catches omissions the original author may have overlooked, and confirms that the documentation is clear enough for someone unfamiliar with the work to follow.

It is also worth noting that documentation can become bloated over time as systems evolve and changes accumulate. Outdated or irrelevant information left in place can cause confusion just as readily as missing information. While caution should be exercised before removing anything,

periodically pruning documentation of content that is clearly no longer relevant keeps it clean, navigable, and trustworthy. Together these three practices — standardization, currency, and peer review — form a documentation culture that saves the team time and the company money.

Fostering a Knowledge-Sharing Culture

Knowledge sharing within a team benefits not only individual members but the organization as a whole. Every engineer brings their own areas of expertise and real world experience, and some of the most valuable lessons come from failures that required diagnosis and correction. Establishing formal and informal practices to share that knowledge ensures the entire team can learn from experiences they were not directly part of.

Lunch-and-learn sessions offer a low pressure, informal environment where one team member presents a topic in which they have deep knowledge. Beyond the educational value, these sessions build camaraderie by giving the team an opportunity to interact outside of the pressures of daily work, which is particularly valuable in a small team environment.

Structured onboarding programs are equally important, particularly in technology environments where every company has its own internal structure, preferred tools, and established ways of doing things. A new hire without proper guidance will struggle to perform tasks in the manner the rest of the team expects, creating unnecessary stress and friction. A well designed onboarding program led by a senior team member gives new engineers the foundation they need to contribute effectively from the start.

An internal wiki functions as a private library for the team — a central repository where anyone can contribute knowledge, publish guides, or document processes that others can reference at any time. Rather than interrupting a supervisor every time a question arises, team members have a first point of consultation they can access independently, saving time for everyone.

Pair working is a productive informal practice where two engineers collaborate on a task together, transferring knowledge organically in real time while simultaneously accomplishing work for the company. Knowledge sharing and productivity reinforce each other naturally in this format.

Finally, post-incident reviews should be conducted after any service outage or failure. Bringing the team together to document what happened, why it happened, and what was learned transforms every incident into a learning opportunity and helps prevent the same problem from recurring.

Each of these practices serves the same fundamental goal of knowledge sharing, but through different methods and in different environments — which is important because different people learn in different ways. A team that invests in multiple channels of knowledge sharing builds collective resilience that no single document or training session could achieve alone.

Problem 5: Integration of CI/CD Practices

The team has not fully adopted CI/CD practices, which can lead to delays in updates and changes.

The Strategic Value of CI/CD

Continuous Integration and Continuous Deployment — CI/CD — is a powerful methodology for streamlining the process of delivering software updates. Much like the automated assembly line that transformed Ford into an industrial powerhouse, CI/CD replaces slow, labor intensive manual processes with a fast, consistent, and reliable automated workflow.

The benefits of CI/CD are substantial. It accelerates the delivery of updates and fixes, reduces the number of manual steps in the deployment process, and eliminates potential points of failure introduced by human error. Problems are caught earlier in the pipeline where they are cheaper and easier to resolve, and engineers are freed from repetitive deployment tasks to focus their time and energy on building.

A team that has not adopted CI/CD practices operates at a meaningful disadvantage by comparison. Manual deployment requires more time to accomplish the same outcome, exposes the team to more opportunities for human error, and produces slower response times when updates or fixes are needed. Each of these inefficiencies carries a cost — in engineer hours, in troubleshooting time, and in revenue. Perhaps most critically, a team without CI/CD risks falling behind competitors who have adopted it. In a market where speed to delivery matters, being consistently slower to push updates or improvements can cost the company business. The same principle that made the assembly line a competitive advantage in manufacturing applies equally to modern software development — automation at scale is not just more efficient, it is strategically essential.

Implementation Strategy

Introducing CI/CD practices to a team that has not yet adopted them requires careful consideration of both the technical and human dimensions of the transition. The first priority is assessing the team's existing knowledge base. If the expertise needed for implementation does not exist in house, training will be necessary. One effective approach is to introduce the concept in a low pressure setting — an extended lunch-and-learn session featuring a guest speaker with hands on CI/CD implementation experience can build initial familiarity and enthusiasm without overwhelming the team.

From there, a phased rollout is the most practical path forward. Rather than attempting to integrate CI/CD across all projects simultaneously, the team should begin with smaller, less critical projects where the stakes are lower and there is more room to learn and correct course. This approach minimizes disruption to the existing workflow while giving the team the opportunity to iron out the challenges that inevitably accompany any new technology adoption.

As the team gains experience and confidence with each successful implementation, they can progressively move on to mid-level projects and eventually to the larger, more critical systems. This layered structure serves two purposes — it reduces stress and strain on an already stretched team by keeping early implementations manageable, and it builds the collective knowledge and confidence that makes each successive adoption easier than the last. Over time the cumulative reduction in manual deployment work begins to ease the team's overall workload, justifying the initial investment in the transition.

Two strong tool candidates worth evaluating for this implementation are Jenkins and GitOps with Argo CD. Each has its own merits and trade-offs, and the final selection should be based on the team's specific infrastructure requirements and existing toolchain.

Anticipated Challenges and Change Management

Implementing CI/CD practices is as much a human challenge as it is a technical one. Change is inherently difficult — people are creatures of habit, and resistance to new ways of working is a natural and predictable response. Understanding the specific forms that resistance is likely to take is the first step toward addressing it effectively.

Senior team members may question the necessity of change, having built confidence in established processes over years of experience. Others may fear that automation will render their roles irrelevant, while some may simply feel uncomfortable with tools and technologies they have not yet encountered. There is also a legitimate conversation to be had around compensation — learning new skills represents real value, and acknowledging that openly with the team demonstrates respect for their effort. Each of these concerns is valid and deserves to be addressed directly rather than dismissed.

Beyond the human dimension, there are practical growing pains inherent to any technology adoption. New tools introduce new error types, new failure modes, and new troubleshooting challenges that the team will need to work through before reaching proficiency. These early friction points are inevitable and should be anticipated rather than treated as signs that the approach is wrong.

Securing genuine team buy-in is critical to a successful implementation. Attempting to force change risks morale failures, active resistance, loss of team members, or even deliberate sabotage. The most effective approach is honest, transparent communication — presenting both the benefits and the challenges of the transition openly. For a small team already stretched thin and working overtime, the most compelling arguments are concrete and personal: CI/CD reduces manual workload, improves work-life balance, and builds valuable experience with industry standard tools that enhances each team member's professional value. Rather than framing the change as a threat to existing roles, it should be presented as an opportunity — opening the door to new specializations and a more sustainable, efficient way of working that benefits both the team and the company.

Conclusion

Implementing the changes and adopting the tools outlined in this report will yield significant benefits for both the engineering team and the company as a whole. Each solution addresses a specific pain point, but together they form a cohesive strategy for building a more efficient, resilient, and sustainable operation.

Establishing JSON as a standardized data format reduces pipeline complexity, eliminates unnecessary dependencies, and creates a more reliable foundation for all data processing work. Proactive monitoring and observability through AWS CloudWatch, SNS, CloudTrail, and Trusted Advisor ensures the health of the infrastructure is continuously tracked, protecting service reliability and the customer experience. Addressing resource constraints creatively — through task prioritization, cross-training, and the application of fault tolerance principles to the team itself — helps prevent burnout, reduce human error, and build a more adaptable workforce without requiring additional headcount.

The automation tools available to the team — Terraform, Ansible, Jenkins, and GitOps with Argo CD — are powerful force multipliers that reduce manual workload, minimize human error, and free engineers to focus on higher value work. Investing in documentation standards and knowledge sharing practices ensures that institutional knowledge is preserved, onboarding is streamlined, and every team member has the resources they need to grow within their role. Finally, the adoption of CI/CD practices, while requiring the most significant investment in training and transition, represents the greatest opportunity for long term efficiency gains — transforming the software deployment process from a manual, error prone burden into a fast, reliable, and automated workflow.

Across all five areas, a common thread emerges: the most effective solutions are those that respect both the technical and human dimensions of the challenges being addressed. Tools and processes alone are not enough — the team must be supported, informed, and motivated to embrace change. A team that works efficiently, shares knowledge freely, and has the right tools at its disposal is not just more productive — it is more valuable, more resilient, and better positioned to grow alongside the company it serves.