Intent-Based Feedback Loops in Cloud Environments

by Dinis Cruz and ChatGPT Deep Research, 2025/03/24

Introduction

Cloud computing has revolutionized how we deploy and manage infrastructure, but it still faces a critical gap: after making changes to cloud resources (for example, deploying an AWS CloudFormation stack or updating an Azure Resource Manager template), users receive limited feedback about the *consequences* of those changes.

Today, a successful deployment typically yields only a confirmation that resources were created or updated – it does **not** automatically verify that the user's higher-level goal was achieved, nor does it warn of any side effects such as new security vulnerabilities, performance regressions, or unexpected costs. As a result, cloud engineers and DevOps teams often must manually hunt through logs and metrics, or rely on separate best-practice scanners, to discover if a change truly had the intended effect or if it introduced new issues.

This white paper argues for an **intent-based feedback loop** model in cloud operations, where users declare *what* they intend (their goal or outcome) and the cloud platform provides continuous feedback including:

1) Confirmation that the intended goal has been achieved 2) Insights into side effects (security, performance, cost, consistency issues, etc.) 3) A framework for acknowledging or mitigating risks introduced by the change.

We also explore how recent advances in generative AI – such as Amazon Q and ChatGPT – could facilitate this model by interpreting user intent, predicting side effects, and suggesting remediations.

The discussion covers current shortcomings of these AI tools, the exacerbating role of Identity and Access Management (IAM) complexity, the challenge of eventual consistency in cloud environments, and compares the state of feedback mechanisms across AWS, Azure, and GCP. We conclude with actionable proposals aimed at improving developer experience and operational security through intent-based operations.

The Need for Better Feedback in Cloud Operations

In current cloud operations, the feedback cycle after a configuration change is often slow and manual. When an engineer updates an infrastructure-as-code template (e.g. AWS CloudFormation or Terraform) to modify cloud resources, success is usually measured in terms of resource state convergence (the stack shows status **CREATE_COMPLETE** or **UPDATE_COMPLETE** in AWS, for instance). However, this low-level success doesn't guarantee that the *higher-level goal* was met. For example, an engineer's intent might be "enable logging for resource X so that all activities are recorded." They might deploy a template to enable logging on an S3 bucket or turn on VPC Flow Logs. The deployment could succeed from the API's perspective, yet logs might **not actually be delivered** due to some hidden issue (wrong bucket policy, missing IAM role, no traffic to log yet, etc.). The cloud system typically won't proactively tell the user that "logging is now functioning and logs are flowing to the destination" – the user would have to discover this themselves by checking the log destination or waiting for an incident. In other words, there is a missing feedback loop between the *user's intent* and the *real-world outcome*.

Equally important, cloud changes can produce side effects that are not immediately obvious. Enabling a feature might incur additional costs or affect performance, but the user isn't warned at deployment time. For instance, turning on detailed logging or deploying a trail could generate large volumes of data and charges. If misconfigured, the service might keep retrying an action in the background, silently accruing costs or hitting throttling limits. A concrete example is AWS CloudTrail: if you create an organization-wide CloudTrail with an S3 bucket that is not properly configured to accept log files, CloudTrail will continuously attempt to deliver logs for up to 30 days. During that time, the failed delivery attempts still incur standard CloudTrail charges (Troubleshooting issues with an organization trail - AWS CloudTrail). The user's intent ("audit all accounts via CloudTrail") isn't fulfilled, and a side effect (unnecessary cost) is introduced – yet *the system only provides minimal feedback*, such as a warning on the trail's detail page or an entry in the CLI `get-trail-status` output showing an S3 delivery error (Troubleshooting issues with an organization trail - AWS CloudTrail). It's easy for a busy operator to miss this warning until a significant bill or security gap has developed.

These gaps highlight why a stronger feedback loop is needed. An **intent-based feedback loop** would automatically confirm whether the higher-level intent was achieved (e.g. "Are logs now being delivered for all activities on resource X?") and immediately flag any side effects or gaps (e.g. "Logs are

not being delivered due to a bucket policy error – CloudTrail shows `LatestDeliveryError` indicating access denied (Troubleshooting issues with an organization trail - AWS CloudTrail). Also, retries will incur costs (Troubleshooting issues with an organization trail - AWS CloudTrail).”). With such feedback, the user can quickly adjust the configuration or acknowledge the risk (perhaps choosing to proceed anyway in some cases). In traditional software development, fast feedback loops are known to improve quality and speed – the same principle applied to cloud configuration (often referred to as GitOps or Infrastructure-as-Code practices) would greatly enhance **developer experience**, reduce time-to-detect for issues, and bolster security by catching misconfigurations early.

User Intent vs. Implementation Gaps: An Example

To illustrate the implementation gap between user intent and current feedback, consider a scenario of enabling network flow logs for an AWS VPC – a common operational task intended to improve visibility. The user’s *intent* is: “Begin capturing all network traffic metadata in my VPC for security analysis.” In AWS, this is implemented by creating a VPC Flow Log resource, which can publish logs to Amazon CloudWatch Logs or S3. Suppose the user deploys a CloudFormation template that creates a Flow Log streaming to CloudWatch Logs. The template might include an IAM role for the flow log to use. After deployment, CloudFormation reports success – resources were created. From the cloud’s perspective, the Flow Log is “active.” However, minutes later the engineer notices that no log data is appearing in CloudWatch. Was the intent achieved? Not yet – no logs are actually being delivered.

What went wrong? On investigation, they discover an error message in the Flow Logs console: **Access error**. According to AWS docs, an “Access error” status in VPC Flow Logs indicates that the IAM role is not allowing logs to be delivered, or the role lacks a trust relationship with the flow logs service (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud). In our scenario, perhaps the IAM role’s trust policy was misconfigured, so the Flow Log service couldn’t assume the role. This is a subtle configuration mistake – the stack deployment succeeded (the Flow Log resource exists), but the *intent (capturing traffic logs)* failed due to an IAM issue. The only feedback was an error string in a status field that the user had to know to check (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud). Additionally, the AWS documentation notes that even if a flow log is correctly configured, there can be a delay of 10 minutes or more for the log group and streams to be created, especially if no network traffic has occurred yet (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud). This **eventual consistency** delay means that even without an error, a user might not see immediate results and could be uncertain if their configuration works or not.

From this example we can extract the broader issues: today's cloud platforms treat the deployment step (e.g., creating the Flow Log resource) as the end of their responsibility, whereas the user really cares about the operational outcome (logs being recorded). The gap between **declaring intent** and **verifying outcome** is left to the user to bridge, often by manually querying the system or waiting. An intent-based feedback loop would close this gap. In our Flow Log scenario, such a system might automatically perform post-deployment checks: Is the CloudWatch Logs group receiving data? If not, retrieve the Flow Log's error status via API and surface a clear message like, "Flow Log was created but is not delivering data: IAM role missing trust relationship." It could even suggest the fix ("Update the IAM role's trust policy to trust `delivery.logs.amazonaws.com` (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud)"). Moreover, it would inform the user about the eventual consistency behavior: "It may take up to 10+ minutes for logs to appear after creation (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud). If no logs appear after that, there may be an issue." In essence, the platform would actively work to ensure the user's intent is met, or promptly explain why not.

The Concept of Intent-Based Feedback Loops

Intent-based feedback loops in cloud environments refer to a model where users specify high-level objectives (intents) and the system not only carries out the necessary changes but continuously validates and reports on the alignment between the *declared intent* and the *system's state*. This goes beyond today's declarative Infrastructure-as-Code (which defines *desired resource states*) by explicitly acknowledging the user's *goals and expectations* from those resources. Three key components define the model:

1. **Confirmation of the Intended Goal:** After a change is applied, the system actively checks whether the user's stated goal has been achieved. This is similar to a post-deployment unit test for infrastructure. For example, if the intent was "ensure all API Gateway access logs are enabled and being sent to CloudWatch," the platform would verify that log entries are indeed streaming from every API Gateway in scope. If the goal was "increase the CPU allocation for my database to improve performance," the system might monitor the database's CPU metrics to confirm they reflect the change (and possibly that performance improved). This confirmation should be communicated back to the user in clear terms, e.g., "Logging is now active for 12/12 API stages – verified (Troubleshooting issues with an organization trail - AWS CloudTrail)," or "DB CPU capacity increased from 2 vCPUs to 4 vCPUs and current utilization has dropped below 50% as expected."
2. **Insight into Side Effects:** Every change in a cloud environment can have ripple effects. Opening a port in a security group might fulfill a connectivity intent but introduce a security risk; enabling detailed monitoring may achieve better observability but add cost and slight latency; updating one resource could unintentionally disable another (e.g., updating an IAM role's policy could break a different service's access). An

intent-based approach means the cloud platform proactively analyzes and reports **side effects** of a change. This could include security implications ("Port 22 open to 0.0.0.0/0 – this is a potential security issue"), performance or cost impact ("Enabled X-Ray tracing on your Lambda – this can incur additional cost and slight overhead"), or propagation issues ("Not all sub-resources picked up the new setting due to permission constraints"). In our earlier example of CloudTrail, the side effect insight would be: "CloudTrail not delivering to S3 – likely a bucket policy issue. **Side effect:** repeated delivery attempts for 30 days will incur extra charges (Troubleshooting issues with an organization trail - AWS CloudTrail)." By surfacing such information, the system educates the user on consequences that typically only surface much later or in hindsight.

3. **Risk Acknowledgment and Mitigation Framework:** When side effects or potential issues are identified, the platform should provide a framework for the user to acknowledge and address them. This might involve interactive prompts or governance rules. For instance, if a change opens a security risk, the system could require the user to explicitly acknowledge the risk or even obtain approval from a higher authority (in a regulated enterprise scenario) before finalizing the change. Alternatively, the system might offer automated remediation or guardrails. An example would be AWS Config rules or Azure Policy that automatically revert or flag non-compliant configurations. In an intent-based loop, upon detecting a side effect, it could say: "Your intent was applied, but it caused a high-risk configuration. Do you want to *accept* this risk, or shall we apply a safe default fix?" For a cost-related side effect: "This change may increase your monthly bill by ~15%. Acknowledge or consider alternatives." By building risk acknowledgment into the workflow, organizations can enforce compliance (the risky change is documented or disallowed unless explicitly approved) and drive immediate follow-up actions (like scheduling a rightsizing after an upscale operation).

In summary, an intent-based feedback loop turns cloud operations from a one-way push (user sends config, cloud executes) into a **closed-loop interaction**. It aligns with principles of **autonomous systems and control theory** – the user's intent is the target state, and the system continuously senses the environment to ensure that state is reached and maintained, informing the user of deviations. This concept echoes the emerging paradigm of *intent-based networking* in the networking domain, where administrators specify high-level outcomes and the network controllers automatically implement and maintain those outcomes. As Ericsson's research on autonomous networks puts it, "Intent is a declarative expression of the requirements that an autonomous system needs to meet... the system is free to choose a solution strategy autonomously" (Creating autonomous networks with intent-based closed loops - Ericsson). In the cloud context, the "solution strategy" is the set of resource changes and validations needed to satisfy the user's requirements. The advanced feedback loop we propose is a step beyond static policy automation, aspiring to **self-driving cloud operations** where many decisions and checks currently made manually by engineers are handled automatically by the platform, guided by the user's declared intent.

Generative AI: A Catalyst for Intent Understanding and Analysis

Implementing intent-based feedback loops requires understanding human intents (often expressed in natural language or high-level terms) and mapping them to low-level cloud actions and checks. This is an area where **generative AI and large language models (LLMs)** can be transformative. Modern LLMs have demonstrated an ability to interpret natural language requests and even generate code or configurations, which suggests they could act as the "brains" translating a user's intent into cloud operations and subsequent analysis of outcomes.

Understanding and Translating Intent: Generative AI assistants like **Amazon Q** (AWS's AI helper) and **ChatGPT** (OpenAI) can parse user requests such as "Enable logging for resource X" or "Harden the security of my database" and break them down into specific cloud tasks. Amazon Q, for example, has been trained on 17+ years of AWS documentation and is marketed as capable of aiding developers and cloud operators (Amazon Q: Putting Amazon's new AI assistant to the test). If a user expresses an intent in a chat interface, the AI could identify which AWS service or configuration is relevant. (e.g., user says *"I want to make sure my S3 bucket is not public"*, the AI knows this relates to S3 Block Public Access settings or bucket policies). Generative models can also **generate the code or CLI commands** needed to implement the intent. Google Cloud's **Duet AI** showcases this capability – it can produce example `gcloud` CLI commands or Terraform snippets based on a description (How Duet AI speeds up development and operations | Google Cloud Blog) (How Duet AI speeds up development and operations | Google Cloud Blog). This lowers the barrier for users to declare intents without knowing exact syntax, and it provides a starting point for enforcement of those intents.

Assessing Side Effects with AI: Beyond implementation, AI can assist in *predicting and detecting side effects*. Because LLMs like ChatGPT have been trained on vast amounts of text including cloud knowledge, they "know" common pitfalls and side effects described in documentation and forums. For instance, an AI assistant aware of AWS might warn, "Enabling Classic Load Balancer access logs requires an S3 bucket with a specific policy; if not set, logs won't be delivered," drawing on known issues. In fact, Amazon's own generative assistant has a feature for **CloudFormation troubleshooting**. When a CloudFormation stack fails, you can click "Diagnose with Q" and the AI analyzes the error and provides a human-readable explanation of what went wrong, plus suggestions to fix it (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS) (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS). This is effectively providing feedback on the intent ("deploy my stack") by highlighting side effects ("the deployment failed due to missing EC2 parameters or inadequate permissions" (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS)). AI can do similarly for non-fatal issues: after a deployment, an AI agent could examine cloud logs, events, and configuration diffs to detect anomalies. For example, it might read CloudTrail events or Config changes and notice that right after the user's change, a certain error event occurred repeatedly – something a human might not notice immediately. We are

already seeing steps in this direction: Google Cloud's Duet AI can **summarize logs and explain errors** in Cloud Operations suite (Duet AI for Developers and in Security Operations now GA | Google Cloud Blog). Imagine after a change, Duet automatically summarizing: "Error rates on Service X increased after the deployment – likely related to the new configuration. Here is the error message trend and an explanation." This kind of insight generation is where generative AI shines, by digesting raw data into meaningful narratives.

Guidance for Remediation and Follow-up: Generative AI tools can also suggest next steps in an interactive way. If a side effect is detected, the AI could recommend remediation steps or even generate the code to mitigate it. Amazon Q's CloudFormation integration not only explains failures but, when asked "Help me resolve," it provides **actionable resolution steps tailored to the specific failure scenario** (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS). This concept can be extended: suppose the intent feedback system finds that enabling a feature increased monthly cost projections by 20%. The AI advisor could suggest: "Consider downsizing XYZ or enabling an idle resource off policy to mitigate cost." Or if a security group is now too open: "Suggested fix: restrict the security group to known IP ranges; here's a CLI command to do that." The AI, having context of the change and its consequences, can act like a smart co-pilot for cloud ops, guiding engineers through complex trade-offs.

Shortcomings of Current Generative AI Tools: Despite their promise, today's generative AI assistants have notable limitations in cloud operations, and understanding these is crucial to designing a robust intent-feedback system:

- *Lack of Direct Environment Access:* Most AI models (like ChatGPT) do not have real-time access to a user's cloud environment for security and design reasons. They operate on training data and user prompts, which means they **cannot dynamically check the actual state** of resources unless explicitly provided. The Cloudsoft analysis of Amazon Q highlights this – when asked a direct question like whether an AWS root account had MFA enabled, Amazon Q could not answer because it had **no access to the account's live configuration** (Amazon Q: Putting Amazon's new AI assistant to the test). It had to admit it couldn't verify that detail. This is a fundamental gap: an intent feedback loop is most effective when it can query real state. Solutions here might involve securely granting the AI limited read access or coupling the AI with traditional monitoring tools. Some providers are moving in this direction (e.g., Duet AI can be enabled in a GCP project with specific permissions (Solved: DUET AI - Google Cloud Community)), but careful governance is needed.
- *Possible Hallucinations and Inaccuracy:* LLMs sometimes generate outputs that sound plausible but are incorrect – a phenomenon known as hallucination (How Duet AI speeds up development and operations | Google Cloud Blog). In a cloud ops context, a confidently wrong suggestion can be dangerous (it might lead an engineer to apply a faulty fix). For example, both Amazon Q and ChatGPT have been observed to produce

subtly incorrect cloud advice. In a comparative test, when asked how to block traffic from a specific IP, **Amazon Q incorrectly suggested using a Security Group rule (with a description "Deny IP")** which is not actually how AWS Security Groups work (they cannot explicitly deny traffic) (Amazon Q: Putting Amazon's new AI assistant to the test). ChatGPT in that case did slightly better by also suggesting Network ACLs, but it too wasn't perfect (Amazon Q: Putting Amazon's new AI assistant to the test) (Amazon Q: Putting Amazon's new AI assistant to the test). This shows that generative models may fall prey to common misconceptions or outdated information. Google's Duet AI addresses this by providing links to documentation in its answers so users can verify facts (How Duet AI speeds up development and operations | Google Cloud Blog), which is a good practice. Still, in an automated feedback loop, we must ensure the AI's analysis is reviewed or confirmed (perhaps by cross-checking with actual data from the environment or by having a human in the loop for critical decisions).

- *Knowledge Cutoff and Cloud Service Updates:* Cloud services evolve rapidly, with new features and changes introduced frequently. An AI model's training data may be months or years old, so it might not know about the latest service behaviors or best practices. This can limit its usefulness or cause it to recommend deprecated approaches. Ensuring the AI remains up-to-date (via fine-tuning on new docs or a retrieval mechanism to fetch current documentation) is an ongoing challenge. Amazon Q, being built by AWS, aims to be up-to-date with AWS docs, but even it might lag or not cover third-party tools. ChatGPT plugins or tools that fetch current docs at query time are one way to mitigate this.
- *Context Limitations:* Cloud configurations are complex and often the context of *why* something is failing or what side effects exist may span multiple services. Current AI assistants typically respond to one question at a time, based on the prompt given. If the prompt doesn't contain the right context (error messages, config snippets, etc.), the answer may miss the mark. In practice, to use ChatGPT for troubleshooting, engineers might paste error logs or portions of CloudFormation templates into the prompt. This works, but it's manual. For an integrated intent-based system, the AI ideally would automatically have context – e.g., it knows which stack was just deployed and can retrieve its events or relevant CloudWatch alarms. Achieving this integration is work in progress (some tools like AWS's re:Post or MS Azure's support bot attempt to fetch relevant knowledge base articles when given an error).

In summary, **generative AI is a key enabler** for interpreting user intent and providing rich feedback, but it must be coupled with real-time data access and robust safeguards. We are seeing early versions of this in industry: AWS's Amazon Q helps troubleshoot stack errors with natural language explanations (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS), and Google Cloud's Duet AI can highlight likely root causes and even recommend fixes for issues in operations (How Duet AI speeds up development and operations - Google Cloud) (Duet AI for Developers and in Security Operations now GA | Google Cloud Blog). However, these tools are not yet a silver bullet. They excel at

assisting humans, but an autonomous intent feedback loop would require them to be more deeply integrated and reliable. The next section delves into why certain cloud complexities – especially around IAM – still necessitate human insight and how AI might address them.

How IAM Complexity Worsens the Feedback Problem

One of the biggest pain points in cloud configuration – and a frequent source of hidden side effects – is the complexity of **Identity and Access Management (IAM)**. Each major cloud provider has its own IAM system (AWS IAM, Azure Active Directory and Role Assignments, Google Cloud IAM), but a common trait is that they are intricate, fine-grained, and can be **inconsistent across services**. This complexity directly impacts feedback loops for two reasons: misconfigured permissions often cause an intent to *partially fail* (in ways not always obvious), and diagnosing these permission issues can be extremely challenging due to inconsistent or opaque error reporting.

In AWS, IAM policies are notoriously powerful but complex. A recent analysis noted that *"IAM policies in AWS can get extremely complex, especially in large environments... While fine-grained control is an advantage, it can be a double-edged sword due to its complexity."* (Weaknesses in AWS IAM: An In-depth Exploration | by Shukhrat Ismailov | Medium). Multiple layers of policies (identity-based, resource-based, service control policies, ACLs, etc.) interact, and determining why a request was denied or why a service behaved a certain way can be non-trivial. For example, enabling a cross-service feature (like having an Lambda function write to an S3 bucket) might require changes in both the function's role and the bucket policy. If any one piece is missing, the outcome is a silent failure or a cryptic error. The AWS IAM team provides tools like the **IAM Policy Simulator** and **Access Analyzer**, but these are separate tools that the user must proactively use; they are not automatically part of the deployment feedback. Moreover, as one expert pointed out, *debugging real-world access issues is difficult, particularly when policies are embedded in infrastructure-as-code templates like CloudFormation or Terraform* (Weaknesses in AWS IAM: An In-depth Exploration | by Shukhrat Ismailov | Medium). In other words, the very practice of codifying IAM in large templates (which is necessary for automation) makes it harder to pinpoint which policy line caused a failure – the user might not even know an IAM issue is the cause without diving into logs or simulators.

IAM inconsistency across services further complicates things. AWS has hundreds of services, each with its own set of actions and sometimes unique behavior. The syntax and semantics of IAM can differ: some services support resource-level permissions, others don't; some require special trust policies (e.g., AWS Organizations service has a unique trust model). An insightful critique from AWS community observed that AWS APIs and features often lack consistency, putting *"the mental burden of inconsistent APIs on developers"* (How AWS dumps the mental burden of inconsistent APIs on developers). While that quote refers broadly to APIs, it applies to IAM as well – a developer's knowledge of how one service's permissions work may

not transfer cleanly to another. This inconsistency can result in unexpected side effects. For instance, an engineer might assume that granting an EC2 role permission to write CloudWatch Logs is enough to enable all logging, but certain logs (like VPC Flow Logs as we saw) need a separate service-specific role or trust policy (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud). If the engineer isn't intimately familiar with that service's quirks, they won't configure it correctly on first try, and the intent (logging) fails. The feedback from the system might just be "Access Denied" or an undocumented error code, leaving them to puzzle out which of the dozen policies in play is at fault.

Azure and Google Cloud have their own IAM complexities. Azure's model ties heavily into Active Directory; often users must ensure the proper Role Assignment exists for a managed identity to perform an action. If an Azure Resource Manager deployment succeeds in creating resources but, say, an App Service cannot write to a storage account, the problem might lie in a missing role assignment or a mismatch in expected identity – and the error might only appear in a log after deployment. Google Cloud's IAM is similarly fine-grained, and GCP offers a Policy Troubleshooter tool that can explain *why* a given account doesn't have access to a resource (showing which policy in the chain denies it). That is a form of feedback, but it's a manual tool – one has to suspect an IAM issue and go use the troubleshooter.

In an intent-based feedback loop system, tackling IAM issues is paramount. The system should automatically flag IAM permission problems that prevent an intent from being realized. This could work by analyzing error messages (many AWS errors include phrases like "is not authorized to perform X" or specific error codes) and by running simulations: for example, after a deployment, an intent-checker could attempt a representative action (like writing a test log entry) to see if permissions are correctly set, and if not, immediately alert the user. Generative AI can help here by reading those errors and instantly explaining them. The AI planning research community is also looking at IAM misconfigurations; one paper notes that given IAM's complexity, *"misconfigurations can result which can lead to unintended side effects that result in cloud attacks"* (Leveraging AI Planning For Detecting Cloud Security Vulnerabilities). This suggests that not only operational outcomes, but security outcomes (like privilege escalation paths) are side effects of IAM configurations. An advanced system could use reasoning engines or AI planning (as in that research) to simulate potential attacks or unauthorized accesses introduced by a new policy and feed that back as, for example, "By granting this role wildcard `s3:*` access, you have unintentionally allowed it to delete buckets (privilege escalation risk) (Weaknesses in AWS IAM: An In-depth Exploration | by Shukhrat Ismailov | Medium). Are you sure you want to proceed?" Such feedback would dramatically improve security posture by catching overly permissive intents.

In summary, IAM complexity means cloud changes often don't have the effect you expect until all the right permissions are in place, and figuring that out is hard. An intent-based approach, augmented by AI, could centralize and simplify that feedback: instead of the user needing to dig through multiple places, the system itself would detect the permission or inconsistency issue and tell the user in plain language what happened. By reducing

the cognitive load of IAM debugging (How AWS dumps the mental burden of inconsistent APIs on developers) (How AWS dumps the mental burden of inconsistent APIs on developers), we not only make engineers' lives easier but also reduce the risk of security holes left open due to oversight.

Eventual Consistency and Its Impact on Debugging

Cloud environments are highly distributed systems. As such, many cloud services operate under **eventual consistency** models, where changes propagate asynchronously through the system. This can significantly impact the feedback loop after a change – often introducing timing-related uncertainty. An operation might technically succeed, but its effects aren't immediately visible everywhere, leading an operator to wonder if something failed when it actually just hasn't taken effect yet. Without explicit feedback about this propagation delay, engineers can become confused or may take unnecessary (even harmful) follow-up actions (like re-trying operations or making additional changes that conflict).

AWS documents this behavior for EC2 resources: *"the result of an API command that affects your Amazon EC2 resources might not be immediately visible to all subsequent commands... if you run a command to modify or describe the resource that you just created, its ID might not have propagated... and you will get an error... that does not mean the resource does not exist."* (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud) (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud). In practical terms, if you create a new EC2 instance and then immediately query it, you might get an `InvalidInstanceID.NotFound` error, just because the query hit a part of the system that hasn't caught up (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud). AWS suggests workarounds such as **exponential backoff retries and wait time** up to about 5 minutes to let the system reach consistency (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud). This is essentially putting the onus on the user (or their automation scripts) to handle eventual consistency, which is a form of *blind spot* in feedback: the platform isn't explicitly saying "Resource is created but give me a moment to catch up," rather it just throws a not-found error or empty result.

Now consider the debugging scenario: if an engineer doesn't know about this eventual consistency window, they might interpret the error as a real failure and possibly try to create the resource again, leading to duplicate or conflict. Or they might open a support ticket unnecessarily. Eventual consistency also affects things like AWS Config and CloudWatch metrics – for instance, AWS Config (which tracks resource state) might not immediately record a change, or CloudWatch metrics might take several minutes to appear after a change in a resource's behavior. Similarly, Azure's resource management might show a resource as created, but other dependent services haven't updated (Azure has some strong consistency in ARM deployments, but certain data, like logs or diagnostics, still have delays). In Google Cloud, enabling a new API or service can take time to propagate across a project.

For our intent-based feedback loops, handling eventual consistency is crucial. The system itself should be aware of these delays and incorporate them into the feedback. CloudFormation actually introduced a notion of a **"configuration complete" event**, which is an event that signals when a resource has passed an internal stability check after creation (Understand CloudFormation stack creation events - AWS CloudFormation). CloudFormation spends significant stack deployment time performing *eventual consistency checks* to ensure the resource is fully operational before marking it complete (Understand CloudFormation stack creation events - AWS CloudFormation). This is a form of built-in feedback mechanism: CloudFormation knows certain resources (like AWS ECS clusters, or ECR repositories) might need to reach a stable state or else dependent resources might fail, so it waits and only proceeds when ready (Understand CloudFormation stack creation events - AWS CloudFormation). It even offers this as an event developers can use (the `CONFIGURATION_COMPLETE` detailed status) to trigger their own actions or skip waiting if they choose (Understand CloudFormation stack creation events - AWS CloudFormation). However, this is largely an internal and optional feature – an average user doesn't directly interact with eventual consistency events except to possibly speed up deployments if they accept the risk.

An ideal system would make eventual consistency *explicit*. For example, after a resource creation, it could say: "Resource X created. Waiting for it to become globally visible... (This may take a couple of minutes.)" and then follow up with "Resource X is now visible and active." This removes the mystery for the user. It could even provide a progress indication or at least an acknowledgment that "lack of immediate feedback does not imply failure." This ties back into confirming the intended goal: part of confirming is sometimes waiting until the system has actually realized the change everywhere.

Eventual consistency also can mask side effects. In the VPC Flow Log example, even if everything is configured correctly, the **logs might not appear for several minutes** (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud). Without knowing this, a user might prematurely assume the configuration failed or start changing things. Conversely, if a user deploys a change that inadvertently breaks something (say disables an IAM role needed for a service), there might be a delay until the effect is felt – a window where the system *looks* fine immediately post-change but then problems surface. If feedback loops only check immediately, they might miss an issue that appears after a short lag. Thus, the feedback mechanism should possibly include a short "monitoring window" after changes to catch delayed effects.

Another area is **distributed logs and events**: CloudTrail (AWS's audit log) itself is eventually consistent. It may take a few minutes for events to show up in CloudTrail after an API call. AWS even warns about this eventual consistency in CloudTrail's behavior (Managing data consistency in CloudTrail - AWS Documentation). If an intent-based system were using CloudTrail events to verify something (like "did a user action actually happen"), it would need to account for that delay or use other immediate signals.

In summary, eventual consistency is a fundamental cloud characteristic that any feedback loop must factor in. The current state is that users need to be mindful of it (with AWS explicitly advising to build in delays and checks (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud)). A more user-friendly approach is the platform handling it: informing the user of delays, and making sure not to give false negatives in verification checks. By doing so, we reduce the confusion and false alarms in debugging, allowing engineers to focus only on genuine issues. Moreover, acknowledging eventual consistency in the feedback means the system might temporarily report an intent as "in progress – pending propagation" rather than success or failure, which is more truthful in distributed systems.

CloudFormation Logging Case Study: From Frustration to Feedback

Let's revisit a specific scenario that encapsulates many of these issues: **debugging CloudFormation log delivery issues**, such as enabling AWS CloudTrail or AWS VPC Flow Logs through an AWS CloudFormation stack. This case study will show how current tools handle it and how an intent-based feedback loop could dramatically improve the experience.

Scenario: A DevOps engineer writes a CloudFormation template to enable organization-wide CloudTrail logging across multiple AWS accounts, with the logs delivered to a central S3 bucket and also forwarded to CloudWatch Logs for real-time monitoring. The intent is: *"I want a comprehensive audit trail of all accounts, stored durably in S3 and visible in CloudWatch Logs."* They deploy the stack, and CloudFormation returns **CREATE_COMPLETE** – seemingly a success. However, a day later, when the engineer checks the S3 bucket, they find it empty, and the CloudWatch Logs log group for CloudTrail is not receiving any events. This is alarming: their intent (having audit logs) is not fulfilled.

Current Debugging (Without Intent Feedback): The engineer now has to troubleshoot. They go to the CloudTrail console and see a warning icon on their trail. On the trail details page it states **"Delivery to S3 bucket failed"** (Troubleshooting issues with an organization trail - AWS CloudTrail), and perhaps another warning that **"Delivery to CloudWatch Logs failed."** CloudTrail's status information (accessible via CLI `get-trail-status`) shows fields `LatestDeliveryError` and `LatestCloudWatchLogsDeliveryError` indicating specific errors (Troubleshooting issues with an organization trail - AWS CloudTrail) (Troubleshooting issues with an organization trail - AWS CloudTrail). For S3, a common error is that the bucket policy doesn't allow CloudTrail to write logs. For CloudWatch, the error often notes the role policy is insufficient. Indeed, the AWS docs say: to resolve S3 delivery issues, *"fix the bucket policy so that CloudTrail can write to the bucket"* (Troubleshooting issues with an organization trail - AWS CloudTrail); for CloudWatch delivery, *"fix the CloudWatch Logs role policy"* (Troubleshooting issues with an organization trail - AWS CloudTrail). Perhaps in our template, the

engineer created the S3 bucket but forgot to include a policy that grants CloudTrail (which acts under a specific service principal) access to it. And maybe the IAM role for CloudWatch Logs (which CloudTrail assumes) was missing a permission or trust.

Armed with this knowledge, the engineer edits the template or the bucket policy out-of-band, adds the missing permissions, and eventually gets the logs flowing. But in the meantime, **what was the cost of this gap?** They had a period with no logs (a security risk), they spent time troubleshooting (reducing productivity), and they incurred charges for repeated delivery attempts (Troubleshooting issues with an organization trail - AWS CloudTrail). Furthermore, if they were not diligent in checking, this misconfiguration could have gone unnoticed for weeks – a dangerous situation for compliance.

Intent-Based Feedback Applied: If an intent-based feedback loop had been in place, the story would be different. The moment the CloudFormation stack finished, the system would not simply stop at "CREATE_COMPLETE." It would initiate post-deployment checks aligned with the user's intent "audit all accounts to S3 and CloudWatch." The system would, for example, simulate a CloudTrail event (or simply wait a few minutes and check for expected log file delivery). Failing to detect logs, it would automatically retrieve CloudTrail's status. Upon seeing a delivery error, it would correlate that with known causes. It might use an AI agent that recognizes the error string (e.g., "Access denied to S3" or "CloudWatch Logs delivery failed due to IAM"). The feedback to the user could be a **detailed report** soon after deployment, delivered via the CloudFormation console, email, or chatops message:

- *Confirmation:* "CloudTrail stack deployed, but **audit logging is NOT yet fully operational**."
- *Analysis of Side Effects/Issues:* "Logs are not being delivered to S3. Reason: Bucket policy is missing required permissions for the CloudTrail service (Troubleshooting issues with an organization trail - AWS CloudTrail). Logs are not being delivered to CloudWatch Logs. Reason: The IAM role used by CloudTrail lacks permissions or trust for CloudWatch Logs (Troubleshooting issues with an organization trail - AWS CloudTrail). Additionally, CloudTrail will continue to retry for 30 days, accruing costs (Troubleshooting issues with an organization trail - AWS CloudTrail) while logs are failing to deliver."
- *Remediation Guidance:* "To fix S3 delivery, update the S3 bucket policy with the following statement allowing the CloudTrail service principal to put objects (see AWS docs on organization trail bucket policy). To fix CloudWatch Logs delivery, ensure the role policy includes `logs:CreateLogStream` and `logs:PutLogEvents` on the target log group and that the role's trust policy allows CloudTrail to assume it (Troubleshooting issues with an organization trail - AWS CloudTrail). Would you like to apply these fixes automatically?"
- *Risk Acknowledgment:* If automated fix is possible and approved, it could then apply the changes (maybe even using CloudFormation Stack Updates or a subordinate StackChange), or at least prepare a change set. If the user chooses not to fix immediately (perhaps it was intentional to

not send logs to CloudWatch due to cost considerations), the system would log that the risk was acknowledged and maybe set a reminder.

This scenario demonstrates how much more **streamlined and safe** cloud operations could be with intent-based feedback. The engineer would have known right away that their intent wasn't met and why, and they could address it before it became a lasting problem. Importantly, it also closes the compliance loop: if this were an audited environment, the system's report itself becomes evidence that, yes, logging was intended and any lapse was caught and resolved promptly.

From an academic perspective, this is essentially building a control loop: user declares desired state (all logs on), system acts (deploy trail), system senses actual state (no logs coming), system adjusts or alerts. In control theory terms, it reduces the "steady-state error" of the system with respect to the desired outcome. Practically speaking, it's just making cloud operations more **robust**.

The CloudFormation logging case is one of many – we could do similar analyses for enabling AWS Config rules, setting up an Azure Diagnostic Setting, or configuring GCP Cloud Audit Logs sinks. In each case, the difference between a frustrating day of debugging and a smooth, confident deployment lies in feedback and knowledge of consequences. Our intent-based approach essentially bakes in that knowledge at deployment time, aided by AI and automation.

Existing Tooling Across Cloud Providers: Gaps and Glimmers

Major cloud providers have begun to recognize these issues and offer tools or features that partially address feedback and side-effect awareness, though none yet provide a complete intent-based loop as described. Here we compare AWS, Azure, and GCP on relevant capabilities:

- **AWS (Amazon Web Services):** AWS provides a variety of tools that, when combined, offer pieces of the feedback puzzle. For basic deployment feedback, AWS CloudFormation gives stack events and error messages if something fails. In recent updates, AWS introduced **CloudFormation AI assistance** via Amazon Q, which can highlight the root cause of stack failures and suggest fixes in plain language (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS) (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS) – a big usability boost for error feedback. AWS also has **Change Sets** and **Drift Detection** in CloudFormation. Change Sets let you preview changes (resources to be created/modified/deleted) before executing an update, but they don't inform you of side effects, only the direct changes. Drift Detection can tell you if the actual state of resources has "drifted" from the last deployed template – useful

to catch out-of-band changes or deployment gaps – but it must be run manually and supports only certain resource types. For ongoing side-effect analysis, AWS offers **Config** and **CloudWatch Alarms**. AWS Config Rules can continuously evaluate resource configurations for compliance or problematic patterns (like open security groups or unencrypted volumes) and alert if a change introduces something against best practices. This is a form of feedback on security/compliance side effects, though it's a separate service and requires one to set up appropriate rules. **AWS Trusted Advisor** is another service that analyzes your environment and provides recommendations on cost optimization, performance, security, and fault tolerance. For example, Trusted Advisor might flag underutilized EC2 instances (cost issue) or S3 buckets with public access (security issue). However, Trusted Advisor's feedback is periodic and global – it's not specifically tied into the moment you make a change. You have to go look at its dashboard or notifications. **Amazon CloudWatch** and **CloudTrail** provide the raw monitoring and logging, but interpreting those is left to humans or third-party tools.

A noteworthy new entrant is **Amazon Q (Developer)**, which is a generative AI assistant. Amazon Q can assist with certain operational questions (like network reachability troubleshooting using VPC Reachability Analyzer (Amazon Q: Putting Amazon's new AI assistant to the test - Cloudsoft)) and as noted, CloudFormation issues. It essentially brings some intent understanding ("why did my instance not launch?") and side-effect explanation into the AWS ecosystem. AWS is likely to extend Q to cover more scenarios in the future. Still, as of now (late 2024/early 2025), AWS's approach to feedback is somewhat siloed: you have to use Config for compliance feedback, CloudWatch for performance, Cost Explorer for cost feedback, etc. There isn't a single unified "intent dashboard" where you state a goal and see all green/red lights about that goal.

- **Microsoft Azure:** Azure's deployment model via ARM (Azure Resource Manager) templates or Bicep is similar to AWS – you get success or failure on deployment, with error details if something failed. Azure has a nifty feature called **What-If** for ARM deployments, which is akin to CloudFormation Change Sets. It shows the resources that will change, helping catch mistakes before applying. But again, it doesn't simulate the run-time behavior or side effects, just the intended resource diff. For post-deployment insights, Azure leans on **Azure Advisor** and **Azure Monitor**. **Azure Advisor** is described as a "personalized cloud consultant" that analyzes your configurations and usage telemetry to recommend solutions that improve cost, performance, reliability, and security (Introduction to Azure Advisor - Azure Advisor | Microsoft Learn). It essentially runs analyses (similar to Trusted Advisor) and will tell you things like "You can save cost by resizing this database" or "Enable backup on these VMs for reliability." It's a great service for ongoing optimization and does serve to highlight side effects in a broad sense (like if you forgot to turn on a recommended security feature, it will remind you). However, Advisor's feedback isn't in real-time with deployments; it updates recommendations on a cadence (and as the documentation notes, it might take up to a day after a change to update its recommendations (Introduction to Azure Advisor - Azure Advisor | Microsoft Learn)). **Azure Policy** is another tool – it can enforce certain rules (like disallowing insecure configurations) and can also audit and flag resources that violate policies. If a user's deployment violates an active policy, the deployment might even be blocked

(which is immediate feedback of a sort – a guardrail). Azure Monitor and Application Insights then cover the runtime metrics and logging. Azure has been integrating more with AI as well – for example, GitHub Copilot (backed by OpenAI) is being integrated into Azure developer workflows, and one can imagine a future where Copilot helps with Azure CLI or troubleshooting. Microsoft's strong partnership with OpenAI means we might see something like "Azure Copilot for Cloud" which could mirror what AWS is doing with Q and GCP with Duet. As of now, such an offering is not generally available, but Azure users can leverage the Azure OpenAI Service to build their own chatops bots that answer questions about their environment. In summary, Azure has robust advisory and policy tools, but like AWS, the feedback loop is spread out: you have to consult Advisor for optimizations, Azure Security Center for security alerts, Monitor for performance, etc., and correlate that to your changes.

- **Google Cloud Platform (GCP):** Google Cloud has been very proactive in adding intelligent tooling under the umbrella of **Active Assist** (a suite of tools that includes Recommender, Diagnoser, etc.). **Google Cloud Recommender** automatically provides a variety of insights – for instance, it can recommend IAM policy tightenings (removing unused permissions), resizing compute instances, or deleting idle resources (The Benefits and Limitations of Google Cloud Recommender Service). These are analogous to AWS Trusted Advisor or Azure Advisor suggestions, and they run continuously in the background. GCP's unique strength is in things like **Policy Troubleshooter** which can explain access issues by analyzing IAM policies (a very targeted feedback for IAM intents). GCP also has **Error Reporting** and **Cloud Monitoring** that can alert on anomalies. Where Google has taken a leap is with **Duet AI for Google Cloud**. Announced in 2023, Duet AI is being integrated into the Google Cloud Console and IDEs. It can provide chat-based assistance contextual to your cloud project. For example, an engineer can ask in the console, "Why is my VM not reachable?" and Duet might use the VPC diagnostics to answer, or "How do I set up a Cloud Storage bucket with retention policy?" and it will provide steps or even offer to execute some. One notable feature is Duet's integration with operations: *"AI log summarization and error explanation integrated into Cloud Logging"* helps accelerate troubleshooting (Duet AI for Developers and in Security Operations now GA | Google Cloud Blog). This means when something goes wrong, Duet can look at logs and quickly summarize what the issue is – essentially giving direct feedback on side effects (e.g., if a new deployment is causing a flood of error logs, Duet would highlight that). GCP also has a concept of **Service Health Insights** that can detect if a recent change likely caused an incident (like a regression detector). All that said, GCP's solutions again tend to be piecewise – the engineer needs to engage with these tools. Duet AI is the closest to an interactive intent-based assistant, but it's still user-driven (you have to ask it questions). It's evolving toward being more proactive; for instance, it might proactively show relevant info if a resource is failing.

In comparing the three: **AWS** has strong building blocks and now a dedicated AI assistant in preview (Q) for specific tasks; **Azure** emphasizes built-in best practice checks and policy enforcement for governance; **GCP** is pushing the frontier with integrated AI (Duet) and a host of automatic

recommenders. None of them yet says “tell me your intent and I’ll handle it end-to-end,” but they’re inching there. Each has tooling for *some* side effects: cost and security recommendations (AWS Trusted Advisor, Azure Advisor, GCP Recommender), config compliance (AWS Config, Azure Policy, GCP Config Validator), and to some extent, troubleshooting help (AWS Q, GCP Duet).

However, an overarching observation is that these are not unified. A developer often has to navigate multiple dashboards and tools to piece together the full picture. There is an opportunity for each provider (or a third-party platform) to combine these signals and present them in a single workflow centered on user intents. For example, a future AWS might integrate Trusted Advisor and Config evaluations directly into CloudFormation or the CDK deployment process, guided by Q’s AI analysis. Or Azure could have an “Advisor/Copilot live” mode that kicks in after a deployment to walk through “here’s what changed and what to watch out for.” GCP’s Duet might evolve to automatically comment on a deployment as it’s happening (“you just opened a firewall rule to 0.0.0.0/0; that’s risky – consider restricting it”).

In conclusion, while cloud providers have *some* feedback mechanisms (and they are improving, especially with AI), the current tooling still requires a lot of human correlation and initiation. We have “glimmers” of intent-based operations (like an AI telling you why a deployment failed, or a policy engine preventing a bad config), but the **comprehensive loop** – from intent declaration to automatic confirmation and side-effect reporting – remains to be fully implemented.

Toward Intent-Based Cloud Operations: Proposals and Future Directions

Achieving robust intent-based feedback loops will likely require a combination of technical advancements and changes in how we approach cloud management. Below we outline some proposals and ideas, drawing from theoretical foundations and practical considerations, to move towards this goal. These proposals aim to improve developer experience by reducing guesswork and to enhance operational security by catching issues early.

1. Unified Intent Declaration Interface: First, cloud providers (or third-party frameworks) should introduce a way for users to explicitly declare *intents/goals* as part of their infrastructure changes. This might be an extension of infrastructure-as-code languages or a higher-level metadata layer. For example, a YAML schema could allow something like:

```
Intent: "Enable comprehensive logging on all resources in Stack XYZ"
Resources: [ ... definition of resources ... ]
ValidationCriteria:
```

```
  - type: log_delivery
    resource: XYZ-Resource
    target: CloudWatchLogs
  - type: log_delivery
    resource: XYZ-Resource
    target: S3
```

This is a rough sketch, but the idea is to encode what the user ultimately cares about (in this case, that logs from a resource reach certain destinations). The cloud deployment service could then use these hints to know what to check after provisioning. In the absence of an explicit interface, even interpreting natural language descriptions in commit messages or deployment commands could be useful (with AI help). The key is to capture intent in a machine-readable form to drive the feedback loop.

2. Instrumentation and Post-Change Checks: Cloud systems should perform automatic *post-change verification tests*. This is analogous to running unit tests after deploying code. For each type of intent, there can be predefined or auto-generated tests. If the intent is enabling a feature (like encryption, logging, backup), the system should query the resource or related services to ensure that feature is indeed active and working (e.g., create a dummy object and see if it gets encrypted or logged). If the intent is performance-related (scale up for better throughput), the system could compare key metrics before and after, or at least confirm the scaling occurred and no bottleneck flags are raised in monitoring. Some of this is already done in specific cases (CloudFormation's eventual consistency checks (Understand CloudFormation stack creation events - AWS CloudFormation), or health checks in rolling deployments), but we propose extending it broadly. These checks should be run soon after deployment and then possibly again after a short delay to catch eventual consistency effects. The results should be summarized to the user as a report.

3. AI-Driven Analysis of Outcomes: Integrate a generative AI agent that takes the user's intent and the telemetry from the deployment (logs, events, metrics, configuration diffs) and produces an **"Assessment Summary."** This summary would state whether the intent was met and list any side effects or concerns. By leveraging AI, this report can be in natural language, referencing documentation or known best practices. For instance, after a deployment the AI might say: *"All resources were created. The web server is reachable and serving traffic (intent achieved). However, I noticed the security group for the server allows traffic from anywhere on port 22 (SSH) – this is a security risk (Creating autonomous networks with intent-based closed loops - Ericsson). If that was intentional, consider restricting it or using an alternative secure access method. Also, CPU utilization on the database jumped to 85% after this change, which could impact performance."* The AI would get these insights by cross-referencing configuration (saw 0.0.0.0/0 in a security group), monitoring data (saw CPU CloudWatch metric spike), and security knowledge (knows SSH open to world is bad). This kind of holistic analysis is exactly what humans do in post-mortems or code reviews; an AI can do it faster and at scale for every change.

4. Closed-Loop Remediation and Suggestions: The feedback loop should not end at detection; it should assist in remediation. Building on the AI analysis, the system can offer one-click or automated fixes for certain issues. For example, if a bucket policy is missing for log delivery, the system could offer: "Apply missing bucket policy for CloudTrail" as a fix, perhaps implemented as a Stack update or a patch. If performance is degraded, it might suggest enabling autoscaling or a larger instance class and could even initiate a trial of that change in a sandbox. This turns the feedback loop into a continuous improvement loop, where the system doesn't just say "here's a problem" but also "shall I fix it or guide you through fixing it?" Users remain in control (especially for potentially destructive changes), but their workload is reduced. Over time, as confidence grows, some of these can be allowed to auto-remediate (for example, auto-revert a change that causes an outage, akin to how some CD systems roll back on health check failure).

5. Risk Dashboard and Acknowledgment Logs: For changes that introduce risks that cannot or should not be auto-remediated (e.g., opening an access that might be needed but is risky), the platform could maintain a "risk dashboard." This would list active risks resulting from recent changes, and require owners to acknowledge them (with maybe a justification). This is somewhat similar to how cloud providers list "Security Findings" (e.g., AWS Security Hub findings, or GCP Security Command Center alerts), but tied to specific change events. The acknowledgement process means there's a record that "Yes, we know port 22 is open to the world on server X as of change Y, and we accept this risk or will mitigate later." This is valuable for internal audits and encourages teams to not ignore warnings. It's essentially bringing the devops idea of "no change without monitoring" to a higher level: no risky change without documented acceptance.

6. Learning from Intent Outcomes (AI Training): Each time an intent is executed and feedback is gathered, that data can be fed back into improving the models and rules. For instance, if the AI missed a certain side effect, engineers can flag it, and the system can learn. If a new type of error occurs, that can be incorporated into the troubleshooting knowledge base. This creates a self-improving system. It is akin to how recommendation engines improve – here the recommendations are about system config and fixes. Cloud providers could gather anonymized telemetry of common issues (e.g., "30% of people enabling feature X forget permission Y") and then update the AI and documentation to catch that upfront. Over time, the aim would be that many issues are anticipated (shifted left) so that by the time you deploy, you already have fixed the things that would have shown up as feedback. In other words, the loop might eventually close earlier: the AI warns you *before* deployment that "If you proceed, likely outcome is logs won't deliver because of missing permission; let's fix that now." That's the ultimate developer experience win: preventing mistakes rather than just catching them.

7. Cross-Cloud and Stack-wide Intents: Many organizations are multi-cloud or have complex stacks spanning many services. Intent-based models should allow goals that are broad, like "Ensure my web application meets the company's security baseline." Achieving this might involve checking configs in AWS, Azure, and on CI/CD pipelines. While cloud providers will focus on their turf, an open approach could use something like Infrastructure

as Code pipelines combined with policy as code (e.g., Open Policy Agent) to express such intents. Then feedback might come as a consolidated report. Academic research and standards bodies (like the **TM Forum for telco networks** looking at intent-driven automation (Creating autonomous networks with intent-based closed loops - Ericsson)) could help define interoperable intent schemas and closed-loop control processes.

8. Human-in-the-Loop Controls: Despite automation, it's important to keep humans in control, especially for critical decisions. The system should be configurable as to what it auto-fixes vs. what it flags for human action. It should also provide clear visualization of what it's found. A developer or SRE should be able to drill down from a high-level intent status ("Logging: **Error**") into the details (which resource, what error, evidence from logs, etc.). This fosters trust in the system and helps with debugging things the system itself might not fully understand.

Implementing these proposals would likely improve the **developer experience** significantly. Engineers could spend more time designing systems and less time debugging them. The cognitive load from dealing with inconsistent APIs and hidden gotchas would decrease because the system would surface those inconsistencies (remember the frustration described by developers about inconsistent AWS APIs (How AWS dumps the mental burden of inconsistent APIs on developers) – a smart feedback loop can mask those by handling differences internally). For **operational security**, we'd have a safety net catching many misconfigurations that today turn into breaches or outages. For example, the risk of leaving an S3 bucket public unintentionally would be almost nil because either the system would auto-lock it down or at least immediately scream for attention, rather than relying on someone noticing a month later or after an incident.

There are challenges to implementing intent-based feedback loops: performance overhead (running all these checks and AI analyses costs time and resources), potential false positives (over-alerting), and the need to maintain a robust knowledge base of what to check for each intent. But cloud providers are uniquely positioned to tackle this because they have a wealth of data and control over the platform. They can build these loops natively where third-party tools might struggle with visibility.

In essence, this vision aligns with the trend toward **autonomous cloud operations**. Much as self-driving cars aim to handle driving with humans only supervising, self-managing cloud systems would handle routine verification and adjustment, with humans only handling exceptions and high-level guidance. Intent-based feedback loops are a big step in that direction, making cloud platforms not just passive executors of commands but active participants in achieving outcomes.

Conclusion

Cloud engineers and DevOps professionals operate in an increasingly complex environment where assuring that every change achieves its intended outcome is non-trivial. The current paradigm – where users specify *how* to configure resources and then manually verify *whether* their higher-level objectives were met – is showing its limits in terms of efficiency, reliability, and security. **Intent-Based Feedback Loops** offer a promising new model that elevates cloud operations to focus on *what the user actually wants to accomplish*. By having cloud platforms understand intents, execute the necessary changes, and then continuously validate and inform the user about the real-world effects and side effects, we close the gap between expectation and reality.

In this white paper, we highlighted how such an approach would address common pain points: misconfigurations (like missing an S3 bucket policy for log delivery) would no longer linger unnoticed (Troubleshooting issues with an organization trail - AWS CloudTrail), performance or cost impacts of a change would be immediately pointed out (instead of surprising you in a billing report), and security regressions could be caught at the moment they are introduced rather than by the next penetration test or incident. We also discussed how generative AI, despite its current limitations, is a key enabler in parsing intents and making sense of complex system data. Tools like Amazon Q and Google's Duet AI already show how AI can translate error messages into plain explanations and guide troubleshooting (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS) (Duet AI for Developers and in Security Operations now GA | Google Cloud Blog). Leveraging such AI in an automated feedback pipeline amplifies its benefit – it's like having a knowledgeable co-pilot review every change you make, at cloud scale.

The challenges of IAM complexity, inconsistent service behaviors, and eventual consistency were identified as major contributors to the current feedback gap. An intent-based system, augmented with AI, can tame IAM complexity by quickly diagnosing permission issues (using knowledge that is otherwise buried in docs and forums (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud)) and even preventing them by learning from past patterns (Leveraging AI Planning For Detecting Cloud Security Vulnerabilities). It can account for eventual consistency by building in appropriate checks and delays, so users aren't left perplexed by transient states (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud). In short, it can bring **order and clarity** to what is today a somewhat fragmented and opaque process.

From a cross-cloud perspective, all major providers have pieces of the puzzle but none have solved it end-to-end. This presents an opportunity for innovation. As cloud adoption matures, attention is shifting from "can we do this?" to "how well and safely can we do this at scale?". Intent-based feedback loops directly target that need by reducing errors and the mean-time-to-detect issues. They also open the door to more **declarative security and compliance**: imagine an organization stating its intent that "no data should be publicly accessible" and the cloud continually enforcing and reassuring that this holds true across all services (with any violation immediately flagged for remediation).

For the audience of cloud engineers, SREs, and security teams, the evolution towards intent-based operations means a future where the cloud feels more like a collaborative teammate than a collection of opaque black boxes. Instead of pouring over logs at 3 AM to figure out why a setting didn't stick, the platform will have already told you – or even fixed it. Instead of dreading the launch of a big change because of unknown unknowns, you can rely on a safety net that catches many of them in real-time. Academic researchers can also find rich areas here: from formalizing "intent" in machine-readable ways, to designing AI that can reason about complex system states and recommend actions, to control theory models for cloud resource management.

In conclusion, intent-based feedback loops represent a significant shift from reactive to proactive cloud operations. By confirming goals, illuminating side effects, and structuring risk handling, they promise a more **transparent, reliable, and secure** cloud experience. Embracing this model will require effort – integrating AI, updating tools, and perhaps rethinking some workflows – but the payoff is a cloud that not only scales with our needs but also smartly guards and guides us as we build on it. The cloud would truly become an "autonomous partner" in operations, embodying the vision of self-healing, self-optimizing infrastructure that has been on the horizon for years (Creating autonomous networks with intent-based closed loops - Ericsson) (Creating autonomous networks with intent-based closed loops - Ericsson). It's time to bring that vision to reality, one intent at a time.

References and Sources

- Amazon Web Services – *"Accelerate AWS CloudFormation troubleshooting with Amazon Q…."* – AWS News (Nov 21, 2024) (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS) (Accelerate AWS CloudFormation troubleshooting with Amazon Q Developer assistance - AWS)
- Cloudsoft – *"Amazon Q: Putting Amazon's new AI assistant to the test"* – *Cloudsoft Blog* (2024) (Amazon Q: Putting Amazon's new AI assistant to the test) (Amazon Q: Putting Amazon's new AI assistant to the test)
- AWS Documentation – *"Troubleshooting AWS CloudTrail"* – *AWS CloudTrail User Guide* (Troubleshooting issues with an organization trail - AWS CloudTrail) (Troubleshooting issues with an organization trail - AWS CloudTrail)
- AWS Documentation – *"Troubleshoot VPC Flow Logs"* – *Amazon VPC User Guide* (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud) (Troubleshoot VPC Flow Logs - Amazon Virtual Private Cloud)

- AWS Documentation – *"Eventual consistency in the Amazon EC2 API" – Amazon EC2 Developer Guide* (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud) (Eventual consistency in the Amazon EC2 API - Amazon Elastic Compute Cloud)
- AWS Documentation – *"Understand CloudFormation stack creation events" – AWS CloudFormation User Guide* (Understand CloudFormation stack creation events - AWS CloudFormation) (Understand CloudFormation stack creation events - AWS CloudFormation)
- Shukhrat Ismailov – *"Weaknesses in AWS IAM: An In-depth Exploration" – Medium* (Oct 19, 2024) (Weaknesses in AWS IAM: An In-depth Exploration | by Shukhrat Ismailov | Medium) (Weaknesses in AWS IAM: An In-depth Exploration | by Shukhrat Ismailov | Medium)
- Luc van Donkersgoed – *"How AWS dumps the mental burden of inconsistent APIs on developers" – Last Week in AWS* (Sept 24, 2021) (How AWS dumps the mental burden of inconsistent APIs on developers) (How AWS dumps the mental burden of inconsistent APIs on developers)
- Arxiv (Research) – *"Leveraging AI Planning for Detecting Cloud Security Vulnerabilities"* (Feb 2024) (Leveraging AI Planning For Detecting Cloud Security Vulnerabilities)
- Ericsson – *"Creating autonomous networks with intent-based closed loops" – Ericsson Technology Review* (Apr 19, 2022) (Creating autonomous networks with intent-based closed loops - Ericsson)
- Google Cloud – *"How Duet AI speeds up development and operations" – Google Cloud Blog* (Aug 2023) (How Duet AI speeds up development and operations | Google Cloud Blog)
- Google Cloud – *"Duet AI for Developers and in Security Operations now GA" – Google Cloud Blog* (Nov 2023) (Duet AI for Developers and in Security Operations now GA | Google Cloud Blog)
- Microsoft Azure – *"Introduction to Azure Advisor" – Microsoft Learn* (Jan 13, 2025) (Introduction to Azure Advisor - Azure Advisor | Microsoft Learn)