



Synchronizing infrastructure as code (IaC) with event-driven programming (EDP)

AWS Prescriptive Guidance



AWS Prescriptive Guidance: Synchronizing infrastructure as code (IaC) with event-driven programming (EDP)

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Introduction	1
Targeted business outcomes	2
Benefits	3
IaC benefits	3
EDP benefits	3
Blind spot scenario	4
Unintentional consequences	4
Problematic approach	5
Recommended approach	5
Best practices	7
FAQ	8
Can updating the deployment of a CloudFormation stack (involving IaC code changes) fix drift?	8
What's the business impact of an outage?	8
What other issues should I consider?	8
Is the scenario with the EBS volume an edge case?	9
Next steps	10
Resources	11
Document history	12

Synchronizing infrastructure as code (IaC) with event-driven programming (EDP)

Ram Kandaswamy, Amazon Web Services

The [infrastructure as code \(IaC\)](#) practice and [event-driven programming \(EDP\)](#) practice (based on an event-driven architecture) are two different approaches to DevOps. Organizations often use some combination of each approach, but this can lead to unintentional consequences that are counterproductive and even harmful to the organization. This guide provides recommendations, best practices, and guidelines for combining IaC practices with EDP practices in the AWS Cloud in a way that's compatible, secure, and mutually beneficial for both IaC and EDP implementations.

You can use an IaC approach to provision and manage IT infrastructure resources by using code and creating configuration files (typically JSON or YAML files) that define your infrastructure specifications. An IaC approach is usually combined with a version control system such as Git to manage the configuration files. EDP is an approach where you develop code to run in response to events that you define. Typically, the code modifies a resource or its properties.

This guide is intended for technical roles that have a basic understanding of general DevOps principles, EDP architectures, and AWS Cloud services.

Targeted business outcomes

This guide can help you and your organization achieve the following objectives:

- Set up an enterprise-level design pattern that can help you align your IaC and EDP implementations as part of your AWS Cloud journey.
- Keep code repository resources and physical resources in sync with each other in the AWS Cloud.
- Seamlessly integrate your DevOps and production monitoring teams so that you can avoid blind spots related to resource creation and updates in AWS CloudFormation.
- Save time by quickly restoring applications to a working state and avoid the financial costs caused by a service outage.
- Enforce strict security for production systems involving automation.
- Achieve higher system availability by using automated responses that meet real-time requirements.

Benefits

This section covers some of the main benefits of using IaC and EDP to solve common problems with managing or provisioning IT infrastructure resources.

IaC benefits

The key benefit of an IaC approach is that it can help you represent and preserve the state of IT infrastructure resources by using version control. This approach can solve multiple problems, but the two most important problems include the following:

- **Ability to revert back to a previous configuration or state if a resource fails without worrying about the resource's values** – Typically, resources and their configurations are backed up and restored without IaC, but IaC can use a version control system to help you revert back (or roll back) to one or more versions of a backup in the backup timeline. You can also tag deployments so that versions that are unsuccessful due to configuration values could be marked and eventually discarded.
- **An accurate representation of a resource's state** – If a resource is successfully created or updated or a rollback is completed, then the template file will have the resource properties matching the actual resource. This means that you can scan and query the repository so that you can understand the state of the resource. Additionally, an IaC approach doesn't require developers to have environment permissions (for example, in a production environment). Finally, you can better ensure a tight security posture for your organization if you have an accurate representation of a resource's state.

EDP benefits

The key benefit of EDP is that it can help you plan for and solve issues that arise from specified future events without the need for an engineer (that is, human effort). Instead, you can use event-driven code to respond to potential issues. EDP also offers the following additional benefits:

- **Reduced deployment time** – An EDP approach reduces the time it takes to make deployment changes because code is invoked immediately after the event.
- **Scalable resources** – You can use an EDP approach to scale resources that must be modified, which could take much longer to modify manually.

Blind spot scenario

This section provides an example of a scenario where an engineering team can make an IT decision that results in a blind spot that has unintentional consequences that can harm their organization. The hypothetical scenario assumes that an [Amazon Elastic Block Store \(Amazon EBS\)](#) storage volume is created in the AWS Cloud by using [AWS CloudFormation](#) as part of an IaC approach.

For this example, assume that an engineering team writes and then deploys a fully-tested program that generates an automated response if the capacity of a utilized volume crosses 80 percent (an arbitrary number for this example). The program responds to the threshold event by calling APIs to increase the volume size. The engineering team expects that the volume will continue to grow over time but not lead to a production outage. The engineering team has solved the storage space problem, but another subtle and often neglected problem emerges unintentionally—drift.

Note

This is just an example and other scenarios are possible. For instance, you could replace the EBS volume with a security group where the engineering team creates an ingress rule that allows traffic from a dynamic IP address based on an event.

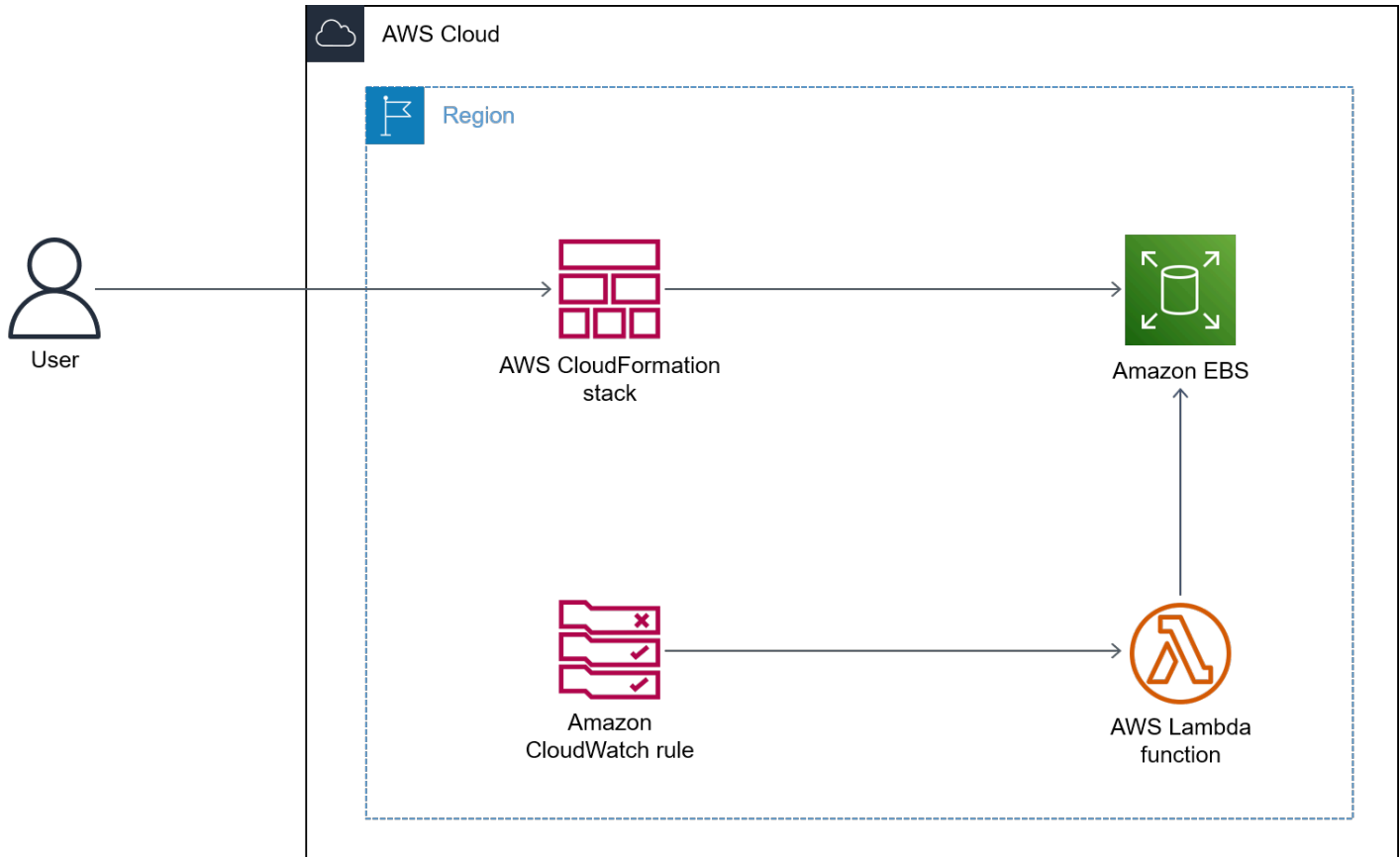
Unintentional consequences

[Drift](#) occurs when you change the properties of the resources outside of the IaC system. In this scenario, that means the resources that are provisioned by CloudFormation. Engineers in this scenario use an API to increase the size of an EBS volume. The drift introduced by using the API instead of CloudFormation has a ripple effect that leads to additional problems. The issue is also magnified with the update deployment of the same stack in a DevOps process.

Because the volume size changed on the resource outside the code repository, the CloudFormation template isn't aware of the change. Therefore, any deployment will fail until the volume size increase is incorporated in the CloudFormation template. More importantly, if the engineers aren't aware of the volume resource changes, then their templates won't have the necessary updates. Volumes are particularly vulnerable to drift. After you increase volume storage size, you can't decrease volume storage size. Therefore, the CloudFormation template not only fails to update but also fails to roll back because the value of the size parameter can't be applied.

Problematic approach

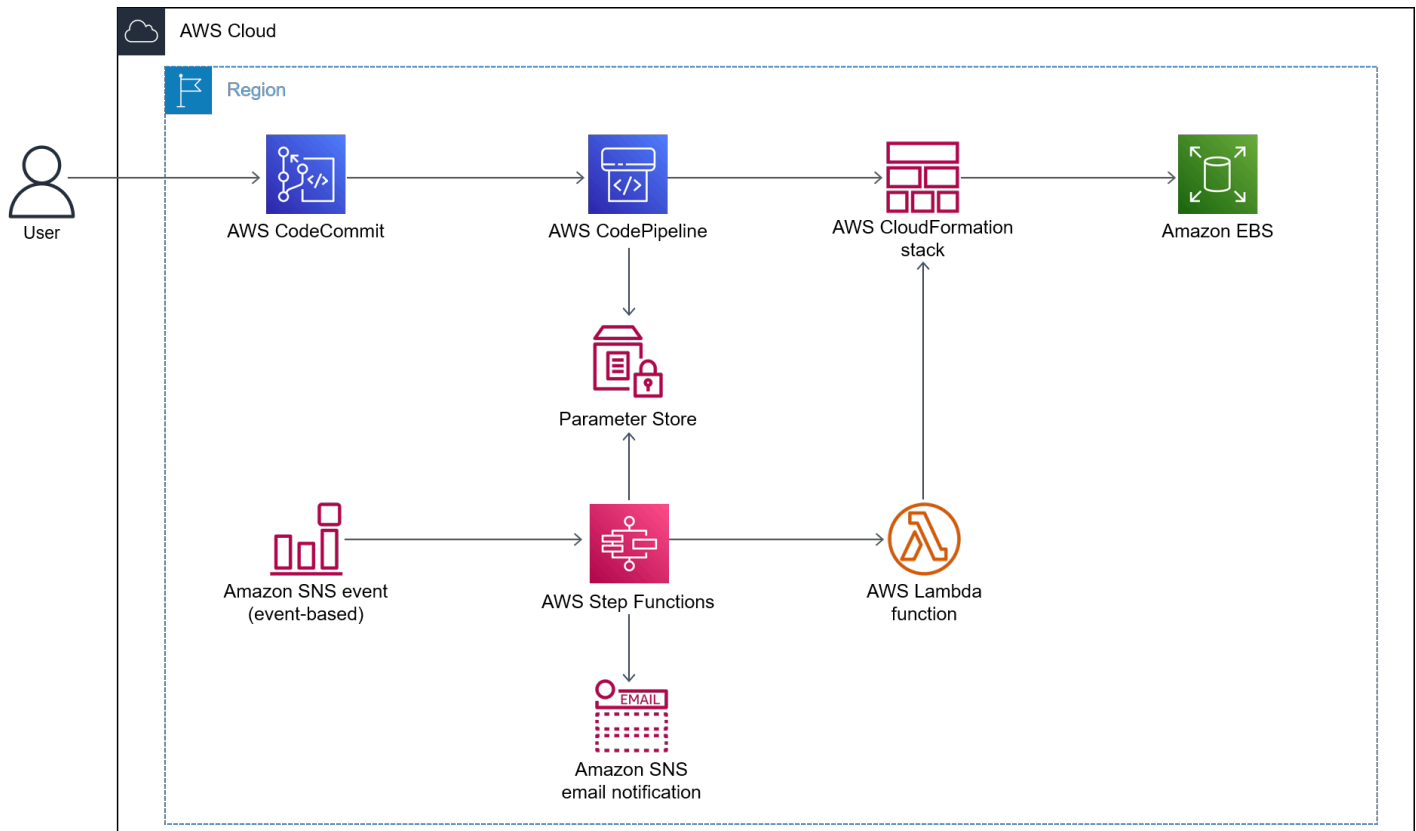
The following diagram shows a problematic solution to the example scenario. The user, as part of the workflow, follows an IaC approach and provisions an EBS volume by using a CloudFormation template. The production monitoring team uses an EDP automation approach and creates an [Amazon CloudWatch Events](#) rule. This rule is configured to invoke an [AWS Lambda](#) function when the EBS volume reaches a specific storage threshold. Then, the Lambda function calls the API to increase the EBS volume size.



Recommended approach

The following diagram shows a recommended approach that aligns with the best practices covered in the *Best practices* section of this guide. This approach involves greater complexity and more AWS services, including [Amazon Simple Notification Service \(Amazon SNS\)](#) for sending notifications, [AWS CodeCommit](#) for source control, [AWS CodePipeline](#) for automated code deployment, and [AWS Step Functions](#) for serverless workflow orchestration. In this approach, CodePipeline adds the EBS volume size to [Parameter Store](#) (a capability of [AWS Systems Manager](#)) initially, and then Step Functions makes any subsequent updates.

You can use Parameter Store to see a history of the values and use it for values that change. Parameter Store also integrates with multiple services, such as CloudFormation and Lambda. The Lambda function in this scenario doesn't directly interact with the EBS volume to increase volume size. Instead, the Lambda function interacts with CloudFormation to update the API. As a result, CloudFormation stacks are safeguarded from drift. Finally, this approach relies on CloudFormation as the single source for performing updates.



Best practices

To use IaC and EDP together in a way that's compatible, secure, and mutually beneficial, we recommend that you follow these best practices:

- **Avoid making API calls directly on a resource** – Use the CloudFormation [update-stack](#) API call as part of an EDP approach, and make sure that only CloudFormation makes changes to the EBS volume. To ensure a successful update operation, the volume size must be valid and must not be lower than the value of the resources currently provisioned. The volume size could be a parameter in the CloudFormation stack.
- **Avoid using a single, monolithic template** – Break your template into modular templates (similar to microservices). Modular templates are easier to update, maintain, and troubleshoot.
- **Use a feedback mechanism for EDP** – Design feedback mechanisms that are tailored to your organization's needs. In the previous hypothetical scenario, a feedback mechanism could be an auto-generated pull request to the version control system coupled with a solution to tag system events and metadata.
- **Create notification systems** – Notification systems can help notify the team when a certain event takes place. For example, you can create an email-based notification by using Amazon SNS. Finally, it's a best practice to always ensure that stacks are in sync and that your DevOps system is designed with guardrails that can help you avoid resource drift. For more information, see [Set up AWS CloudFormation drift detection in a multi-Region, multi-account organization](#) in the AWS Prescriptive Guidance documentation.

FAQ

Can updating the deployment of a CloudFormation stack (involving IaC code changes) fix drift?

CloudFormation performs an update with a fail-safe approach. That is, CloudFormation doesn't apply the template changes with the `--force` option on its resources. Instead, CloudFormation identifies the changes to be performed on an existing resource by creating a [change set](#). To identify the change set, CloudFormation uses the resource's current properties instead of relying on the previous template version. The stack update can fail depending on the resource type and the contract of the resource's API. In the hypothetical scenario covered previously, an EBS volume contract doesn't allow you to downgrade the volume size and such updates result in a failed state. CloudFormation recommends that you manage a resource's state through CloudFormation only and not directly with APIs.

What's the business impact of an outage?

There is a direct financial impact that depends on the number of resources involved in a service outage. If the pipeline is unaware of the changes done to a stack through API calls on resources and the stack fails, then troubleshooting the root cause of the problem and restoring the infrastructure to a working state can become a time-consuming and difficult process due to the `UPDATE_ROLLBACK_FAILED` state.

What other issues should I consider?

If a CloudFormation stack involves custom resources and the EDP solution makes changes to a resource's state, it's possible for the resource to revert back to an old configuration (blind spot). This is because the CloudFormation template is unaware of the change to a resource or property. There is an unintended restore to an old configuration, and the stack could deploy successfully but with the incorrect configuration. It can be time-consuming to troubleshoot the issue, depending on the specific resource.

Is the scenario with the EBS volume an edge case?

The EBS volume scenario isn't an edge case. The EBS volume scenario illustrates the importance of ensuring synchronization between CloudFormation and actual AWS resources. Another example of a common scenario involves modifying an Amazon Simple Storage Service (Amazon S3) bucket policy. You can make a policy update to an S3 bucket by using the AWS Management Console or an API (when you [use an S3 bucket as an origin](#) for an Amazon CloudFront distribution). In this scenario, the Amazon S3 policy could be overwritten by pipeline.

Next steps

We recommend that you consider the following next steps:

- **Use CloudFormation to manage all stack resources** – After you launch a CloudFormation stack, use the [CloudFormation console](#), [API](#), or [AWS Command Line Interface \(AWS CLI\)](#) to update resources in your stack. Avoid making changes to stack resources outside of CloudFormation. Doing so can create a mismatch between your stack's template and the current state of your stack resources, which can cause errors if you update or delete the stack. For more information, see [Walkthrough: Updating a stack](#) in the AWS CloudFormation User Guide.
- **Use AWS Systems Manager Parameter Store to create parameters** – Consider using Parameter Store to create parameters and then use the parameters in your CloudFormation template. You can use Parameter Store to better maintain your templates and reduce operational overhead. For more information, see [Working with Parameter Store](#) in the AWS Systems Manager User Guide.

Resources

- [AWS CloudFormation](#) (AWS documentation)
- [Infrastructure as Code](#) (AWS Whitepaper)
- [What is an Event-Driven Architecture?](#) (AWS documentation)
- [AWS Lambda](#) (AWS documentation)
- [What Is Amazon CloudWatch Events?](#) (Amazon CloudWatch User Guide)
- [Introduction to DevOps on AWS](#) (AWS Whitepaper)
- [Optimizing Enterprise Economics with Serverless Architectures](#) (AWS Whitepaper)
- [Enable Amazon GuardDuty conditionally by using AWS CloudFormation templates](#) (AWS Prescriptive Guidance documentation)

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Rename strategy	New title: <i>Synchronizing infrastructure as code (IaC) and event-driven programming (EDP)</i> . Previous title: <i>Combining infrastructure as code (IaC) and event-driven programming (EDP) approaches on AWS</i> .	October 13, 2025
Initial publication	—	March 6, 2023