AWS Well-Architected Framework

Operational Readiness Reviews (ORR)

Operational Readiness Reviews (ORR): AWS Well-Architected Framework

Table of Contents

Operational Readiness Reviews (ORR)

Publication date: **June 30, 2022** (*Document history*)

Amazon Web Services (AWS) created the Operational Readiness Review (ORR) to distill the learnings from AWS operational incidents into curated questions with best practice guidance. This document is intended to help you understand how the AWS ORR program was built and guide you in creating your own ORR program as part of the AWS Well-Architected Framework. Creating an ORR program can help supplement Well-Architected reviews by including lessons learned that are specific to your business, culture, tools, and governance rules.

Introduction

At AWS, we strive to build and operate highly resilient services, keeping in mind that everything fails all the time. When failures happen, we use a closed-loop mechanism called Correction of Errors (COE) to perform a post-incident analysis of any event of significance, even if customers don't see an impact. The focus of a COE is preventing the reoccurrence of that event in the workload where it happened by generating action items specific to that workload and event.

However, we also want to stop preventable, known risks that we've identified in the COE process from occurring in other workloads. We, like so many of our customers, can't afford to slow the pace of innovation; developer speed and agility is critical to our business. And given AWS's decentralized operational culture, we needed to create a scalable, self-service mechanism to share and enforce the best practices learned from our COE analysis without slowing builders down.

To do that, we created the Operational Readiness Review (ORR). The ORR program distills the learnings from AWS operational incidents into curated questions with best practices guidance. This enables builders to create highly available, resilient systems without sacrificing agility and speed. ORR questions uncover risks and educate service teams on the implementation of best practices to prevent the reoccurrence of incidents through removing common causes of impact. We generate different checklist templates from these questions based on the workload being reviewed and the outcome we want to achieve. Teams perform self-assessments on operational risks to achieve operational excellence by reviewing the appropriate ORR checklist throughout the complete lifecycle of their service, from inception to post-release operations. ORRs helps us achieve *shorter*, *fewer*, and *smaller* incidents. It uses a data-driven approach for reducing risk and improving the availability and resilience of our systems.

The focus of this paper is to help you understand how to build an ORR program and develop your own checklist questions to support both the Operational Excellence and Reliability pillars of the AWS Well-Architected Framework. The core value proposition of the ORR is using the data from your own post-incident analysis to generate best practices. Creating an ORR program can help supplement Well-Architected reviews by including lessons learned that are specific to your business, culture, tools, and governance rules. ORR is a complementary process to Well-Architected, using a data-driven approach to ensure a consistent review of operational readiness, with a specific focus on eliminating known, common causes of impact in your workloads.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

Building mechanisms

The [introduction to the Well-Architected Framework](#) introduces the concept of mechanisms.

> **"Good intentions never work, you need good mechanisms to make anything happen" — Jeff Bezos**

What this means is that you have to replace human best efforts with repeatable, scalable processes and tools, which are often automated, to achieve the desired outcome. A mechanism is a complete process where you create a **tool**, drive **adoption** of the tool, and **inspect** the results in order to make course corrections. It is a "virtuous cycle" that reinforces and improves itself as it operates. It takes controllable **inputs** and transforms them into ongoing **outputs** to address a recurring business challenge.
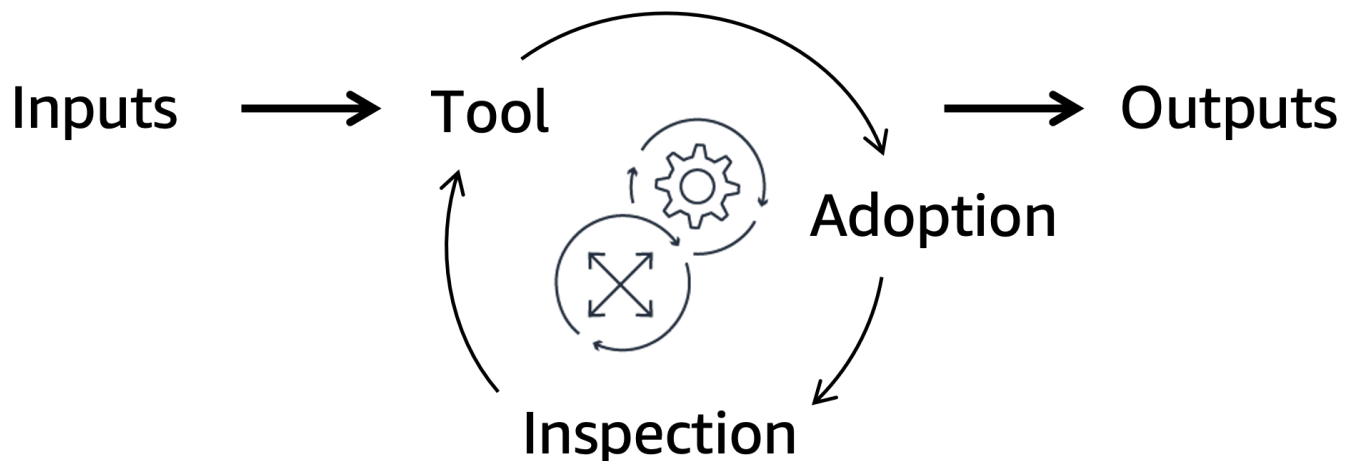

Figure 1: The complete process of a mechanism

The cyclic nature of a mechanism makes it best suited for solving recurring problems or opportunities, as opposed to one-off challenges. The ORR is a mechanism with a tool, an adoption process, and an inspection process that operates in a complete cycle. In the next section we'll discuss the ORR mechanism and how to build one.

The ORR mechanism

To build a mechanism, work backwards from the challenge you want to solve. The ORR was designed by AWS to help prevent the reoccurrence of known, common causes of impact in services without slowing builders down. The design and operations of those services are the inputs. The outputs are what you want to achieve by resolving the business challenge. In our case at AWS, the desired business result was higher levels of availability and resilience in our systems by decreasing the frequency of incidents (*fewer*), decreasing the duration of incidents (*shorter*), and decreasing the scope of impact of an incident (*smaller*). You can start with the same business challenge and results when you create your own ORR mechanism.

The following sections examine each component of the mechanism. Each section describes the AWS approach and provides recommendations for that part of the mechanism. After you have defined your business challenge, use these sections as a guide for building the tool, driving adoption, inspecting the process, and iterating.

Topics

- [The ORR tool](#)
- [Gaining adoption](#)
- [Inspect the process](#)
- [Iteration](#)

The ORR tool

The main tool for ORR is the checklist of questions itself. AWS has built a web service around this checklist to create templates, provide a consistent user interface (UI), links to cautionary tales, and set up integrations with the AWS ticketing system. This allows teams to perform a self-service review of their workload, record the results, understand their residual risk, and track action items that result from the review, which can be directly added to their backlog.

Let's take a look at an example of a question AWS might ask in one of those checklists. Some systems choose to implement [certificate pinning](#). While there are some potential security benefits, this practice poses a significant availability risk if the pinned certificate is replaced, which can occur for any number of reasons. A question and guidance in your ORR checklist for certificate pinning might look like the following.

> **Question: Do any of your hosts pin certificates?**
>
> **Guidance**
>
> We recommend against using certificate pinning because it introduces a potential availability risk. If the certificate to which you pin is replaced, your application will fail to connect. If your use case requires pinning, we recommend that you pin to a certificate authority (CA) rather than to an individual certificate.
>
> ☐ Yes | High Risk
>
> ☐ No | No Risk

If you haven't had an incident related to certificate pinning, or it's not a high-priority item to address across your enterprise, then don't include this question. The ORR checklist is most effective when it's focused on incidents that present critical risks. These are the types of risks that would prevent a General Availability (GA) launch of a service. Medium or low risks aren't included in the ORR to keep it a lightweight process that doesn't overburden teams and reduce their agility and ability to innovate.

Customer recommendations

To get started with an ORR program, you don't need the same level of tooling that AWS has built. The most important component is generating the questions themselves. It is recommended to review three different categories:

- Real incidents that you've had in the past

- Near-misses that you've had in the past

- The failure modes that haven't occurred, but that you're concerned about

Out of this set of categories, you can begin to develop questions and associated best practices that can either prevent, or reduce in scope of impact or duration of, those incidents in the future. You can take lessons you've learned in both AWS as well as on-premises environments, they aren't exclusive to operating in the cloud. See Appendix A: Creating ORR guidance from an incident for a complete example of how you can generate ORR guidance from an incident. See Appendix B: Example ORR questions for example questions that you can use to start building your own ORR

checklist, keeping in mind that these are only examples and you should tailor the checklist for your specific use cases, environment, and workloads.

To get started building your own checklist, it is suggested to group your questions and develop content in the following areas.

- **Architecture** — The focus is on how you've built your architecture, the dependencies your workload has taken, how you scale and manage capacity, and how you protect your workload from its customers (for example, preventing overload).
- **Release quality** — This section focuses on how you test and deploy changes to your workload including detecting problems, automated and manual rollback procedures, and how you phase changes incrementally to your systems.
- **Event management** — These questions focus on the processes and procedures required to deal with an event when it does occur, including topics like paging an on-call operator, the location and coverage of runbooks, the instrumentation and alarms associated with your workload, and metrics and dashboards you use to understand your workload's state.

Your questions may address people, process, and technology in each area. You may also choose to organize the checklist content in more granular categories, such as the following:

- Deployment safety
- Defense against customers
- Defense against dependencies
- Data recovery
- Operator safety
- Blast radius containment
- Event detection
- Service restart
- Forensics
- Escalation

Using custom lenses, you can build your checklists into the AWS Well-Architected Tool. You might decide to track action items from your ORR in a tool such as AWS System Manager OpsCenter. You also might choose to use the results of post-incident analysis in AWS System Manager Incident Manager as inputs to developing your questions. AWS offers several different engagement models

to help you build your own ORR checklist to complement what you're doing with Well-Architected. Contact your account team for additional details.

Gaining adoption

While the ORR name may imply on the surface that it is a "pre-launch" checklist, the process is actually built into the entire Software Development Lifecycle (SDLC). To be the most effective, ORRs should be integrated with, and adopted across that lifecycle. The following diagram demonstrates how AWS views the adoption spectrum for ORRs.
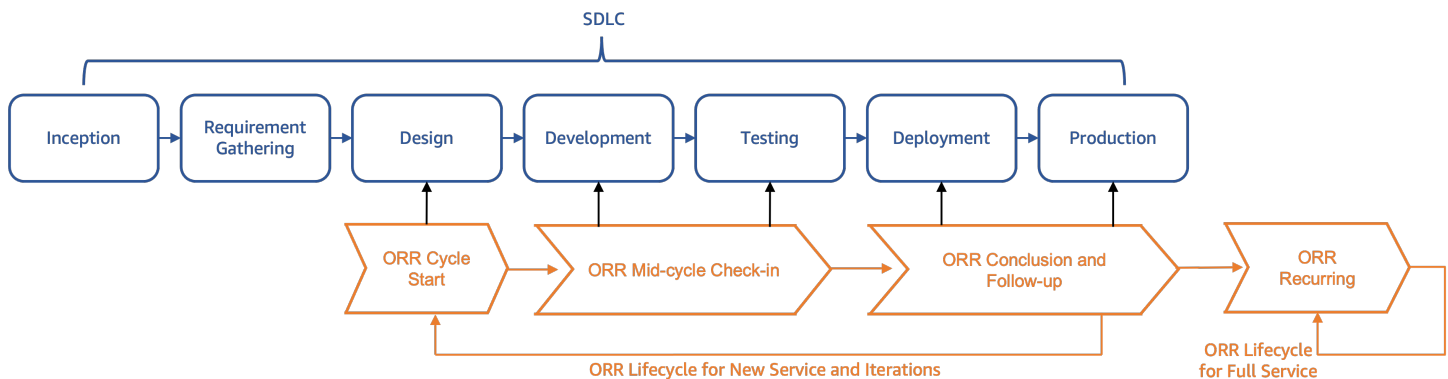

Figure 2: The ORR adoption and implementation lifecycle tied to the SDLC

Figure 2 shows that the **ORR Lifecycle for New Service and Iterations** is initiated when a new service, new feature, or architecture change is proposed. The **ORR Cycle Start** phase begins during the *Design* phases of the SDLC process. Teams start to answer the design and architecture questions. At the same time, the team has a holistic view of all ORR questions that are associated with the upcoming stages of the SDLC. During the **Mid-Cycle Check-in** phase teams start to answer *Development* and *Testing* related questions. Finally, in the **ORR Conclusion and Follow-up** phase, the team wraps up the ORR checklist and develops their risk mitigation and follow-up plan.

In addition to the ORR performed through the SDLC process, at least annually, teams are expected to perform an ORR on their full service using a checklist tailored to that event. This helps verify that they stay up to date with new or updated best practices, and also that nothing has changed within their systems.

To support adoption of the tool across these different phases, AWS uses different checklists for different occasions and workload types. A few examples are:

- Launching a new public service
- Launching a new major feature

- Launching a new minor feature

- Recurring annual review

- Serverless workload

- User console

- Server agent

Customer recommendations

Once you have established a working mental model for the tool, identify stakeholders who will be impacted by it. You might ask questions like: "Who needs to contribute to it?" or "What do they need to do in order to adopt and implement the tool?"

One way to drive adoption is to start small and expand. Find a new workload that is being built or one that is targeted for migration to AWS. Pilot the ORR process with that workload and generate lessons learned. Another approach would be to conduct ORRs on your most critical workloads first. An important lesson in driving adoption of anything is to "make it easy to do the right thing". The simpler it is for teams to consume and use the tool as part of their day-to-day business, the easier it is to gain broad adoption.

One of the most important factors to gaining adoption is ensuring the checklists are aligned with the outcome you want to achieve. As mentioned previously, keep the checklist questions to the minimum required to address critical risks. Verify that the checklists align with the type of workload being evaluated. Minimize the possibility for "not applicable" answers to make sure the process doesn't introduce unnecessary overhead and take additional time from your builders.

Inspect the process

AWS uses several different inspection processes for the ORR. First, AWS holds a weekly operational metrics meeting attended by thousands of engineers and leadership up to the Senior Vice President (SVP) level. Each week, different service teams present their metrics and dashboards to the group. One of the metrics that is inspected is when their last ORR was performed and the number of outstanding action items they have.

Secondly, the ORR mechanism is inspected as part of the ORR Conclusion and Follow-up phase. During this phase, teams generate a narrative to describe successes, risks, mitigations, and the tracking mechanisms to close the action items. This narrative is reviewed by senior AWS leadership for new service or major feature launches. The results of the ORR are reviewed during a scheduled

meeting with an audience including the engineering team, principal engineers in their organization, leadership and management, and any stakeholders from dependencies or customers of the service. During the meeting, attendees review the completed checklist and provide feedback on the findings. Any high-criticality findings are escalated to leadership as input to a go or no-go launch decision.

Finally, the effectiveness of the ORR mechanism is inspected during the COE process. The COE template asks questions like "When was your last ORR performed?" and "Would any ORR recommendations have reduced or avoided the impact of this event?". This helps AWS gauge whether the ORR mechanism is effective at preventing or diminishing the impact of events or whether the mechanism should be iterated upon to make it more effective, for example, by adding or changing questions.

Customer recommendations

You'll need to consider how you will inspect your mechanism. This is key in knowing whether the mechanism is actually helping you achieve your desired outcomes. What we find is that top-down buy-in to mechanisms not only helps drive their adoption, but creates an inspection process that is effective at achieving the desired business results. For ORRs to be a successful program for your business, you will need to create an inspection process that has the gravity to drive both cultural change and adoption of the mechanism's tool.

Use multiple perspectives for inspection. You should seek diverse input from product management, IT leadership, developers, and engineers. These different perspectives will give you different insights to the effectiveness of the mechanism and how you may need to alter it.

Iteration

It's unlikely that a mechanism will operate as designed from day one. It takes time to experiment with various tools and find one that works. Adoption can also take significant effort to push out to a large population. Inspection starts when the tool starts to become broadly adopted, but at each stage you may find that alterations need to be made to the mechanism. Here are few examples of how AWS iterates on the ORR mechanism.

First, AWS constantly seeks feedback on the ORR mechanism from its users, the AWS service teams. This drives the creation of new checklists for different occasions or different types of workloads (like a serverless application, user console, or server agent) so that each one is as pertinent as possible for its consumers. It also helps curate the guidance and questions used in each checklist

template. Finally, it also drives enhancements in the user experience provided by the ORR web service.

Another way AWS iterates on the ORR mechanism is through a specialist engineering community called "Operational Champions" or "Ops Champion" for short. They provide two different functions for the ORR program. The first is as part of the ORR process itself. Teams engage an Ops Champion during their review. The Ops Champion challenges the team on their answers to the checklist, provides context on the adoption and prioritization of best practices, and ends up influencing everything from workload architecture to operational culture in the team. They are part of the complete process, including the review meeting and in retrospectives after the ORR is complete to review lessons learned.

The second function they provide is as a working group to continue to unify and document emerging best practices from around our decentralized service teams to avoid pockets of institutional knowledge. They focus on the ORR questions to ensure they are asking the right thing, providing the right guidance, verifying results can be measured, determining risk severity, or developing solutions to make the best practices easier to adopt and implement. They review the outcomes of COEs and create new lessons learned and new best practices. There is a tight coupling between the COE process and ORR, we use the lessons learned to continually generate new content to deal with evolving risks in distributed systems. We also use that information to ensure we prioritize the right risks for inclusion into the ORR checklists.

Customer recommendations

Quick iteration has proven to be a valuable approach for building modern distributed systems and is equally valuable in building mechanisms. Just as with the inspection process, seek diverse perspectives to create a more holistic understanding of how your mechanism might need to change. Provide opportunities for honest and, if possible, anonymous feedback on the tool and process. Find out which questions were the most useful or which problems can be solved with automation or centralized solutions. Use this feedback to improve the tool and make it easier to drive greater adoption.

Additionally, developing a community of operationally focused specialists helps improve the effectiveness of the tool, drives further adoption of the tool, and enhances the ability to iterate on the mechanism. You will likely build your own Ops Champion community as you iterate on your ORR program. AWS Solutions Architects (SAs) and Technical Account Managers (TAMs) can help you develop this community.

Conclusion

This document described how AWS built Operational Readiness Reviews as a mechanism to achieve smaller, fewer, and shorter incidents without slowing builders down. The ORR mechanism consists of a tool that builders are driven to adopt, and then the results of the process are inspected. Based on the inspection, the mechanism is iterated and improved upon so that its effectiveness in achieving the desired results can be measured.

ORRs can be an effective mechanism that you can use to prevent the reoccurence of known causes of impact as part of the Well-Architected Framework. Use ORRs as focused reviews supporting the Operational Excellence and Reliability pillars of the Well-Architected Framework to make your workloads more resilient to failures and create a culture of operational excellence. To get started building an ORR program, contact your AWS account team.

Appendix A: Creating ORR guidance from an incident

This section will provide an example of using a post-incident analysis to generate questions and guidance for an ORR.

Scenario

The workload in this incident is deployed using AWS CloudFormation to create infrastructure like a VPC, Amazon EC2 instances, auto scaling groups, Amazon DynamoDB tables, Amazon S3 buckets, and AWS Identity and Access Management (IAM) roles and policies. The EC2 instances are configured with an IAM role that allows them to GET and PUT items into the DynamoDB table. During an approved manual change event, the IAM role was updated with additional permissions to allow scan and query operations of the DynamoDB table to support new functionality in the service. The updated features to the service are deployed and it operates without issue for several months.

The team is now planning on releasing an additional feature that will require their EC2 instances to download data from an Amazon S3 bucket. As part of the feature release, they update their CloudFormation template with the required Amazon S3 permissions as well as a number of other changes, like creating the S3 bucket, and setting the bucket policy. The update is first run on a beta environment that is used for developer testing, but the environment doesn't exactly mirror production and changes aren't controlled through CI/CD. The update completes successfully in beta and they proceed to deploy the update in production. Then, the engineer applying the change immediately starts seeing errors in the logs that all scan and query operations against their DynamoDB table are failing.

Mitigation

After seeing the errors, it was obvious to the engineer that the CloudFormation deployment was the cause. To revert the new changes, the engineer ran the most recent known-to-work commit of their CloudFormation template. After running the update, the impact was still not mitigated and the errors continued. At this point, the engineer realizes that the necessary IAM permissions for those operations were not in the CloudFormation template and manually adds a new statement to the role's policy to allow them. This mitigates the impact and the service returns to normal operation.

Post-incident analysis

The root cause of this incident was that the manual change to update the IAM policy was not recorded in the CloudFormation template. When the last update was run, CloudFormation replaced the current IAM policy with the one in its template, which resulted in removing the permissions for scan and query operations. The impact wasn't detected in the beta environment because the role being used there had an additional IAM policy attached that allowed all actions on all resources to prevent testing issues related to permissions. There was no other pre-production environment that exactly mirrored production.

ORR guidance generation

Based on this event, you can create a question that asks:

> **How are you ensuring that you do not impact customers when using AWS CloudFormation to manage AWS resources?**

The guidance for this question would be the following:

- Apply the same level of scrutiny to infrastructure managed by CloudFormation as you do for software or configuration changes.
- Separate stateful resources into their own stack to reduce the scope of impact of changes.
- Use drift detection to detect when resources have been changed outside of CloudFormation.
- Use change sets to validate the intent of your stack update matches the actions that CloudFormation will take when the change set is executed.
- Test CloudFormation stack updates in a pre-production environment that mirrors production.
- Apply stack policies to protect critical resources from being unintentionally updated or deleted.

These are some additional questions to consider when preparing a response for this:

- How do you ensure that an operator cannot accidentally delete a stack or critical resources managed by CloudFormation?
- Are you using CloudFormation change sets to validate that the intent of a change matches the actions CloudFormation will apply?

- How are you ensuring that a CloudFormation stack update doesn't affect your largest fault container (usually a Region, zone, or cell)?

- How are you ensuring changes are not made to CloudFormation managed resources directly?

- How are you partitioning resources across CloudFormation stacks?

Finally, provide one or more links to the post-incident analysis that drove the creation of this question to provide cautionary tales of how this has gone wrong before. This helps provide context for the guidance and makes it more concrete for your engineering teams using the ORR.

Summary

The previous guidance could have helped prevent this event. Using change sets would have provided insights that the managed policy was going to be updated. This would have allowed the engineer to quickly do a comparison of the *as-is* and *to-be* configuration and identify that those permissions were being removed. Additionally, running the update in a pre-production environment that mirrored production would have identified that the update resulted in errors before ever being applied to production. Finally, enforcing that all changes to resources provisioned by CloudFormation are made through CloudFormation would ensure that the permissions updates were included in the template. For [resources that support it](), running drift detection before the update will identify that out of band changes were made to a resource.

Appendix B: Example ORR questions

These are a few example questions that you can use to get started building your own ORR program.

Architecture

What have you included in your architectural design to reduce the blast radius of failures?

Please provide a table enumerating all customer impacting APIs, an explanation for what each does, and the components and dependencies of your service that it touches. Include all APIs whether they are public or private from the customer's perspective.

Please construct a failure model listing soft and hard failure modes for each of your components and dependencies.

Guidance

Your failure model should include columns for **Component** or **Dependency**, **Failure Type**, **Service Impact**, and **Customer Impact**.

Please address an outage of your service in its largest blast radius unit, such as a cell, an Availability Zone (AZ), or a Region, plus a total infrastructure outage in its largest blast radius (an AZ).

What is the retry and back-off strategy for each of your dependencies?

Guidance

For dependency calls made within the context of a synchronous API call, you should generally retry once immediately, then give up.

What is the retry and back-off strategy for each of your dependencies?

For dependency calls made within the context of work requested through an asynchronous API call, you should generally exponentially back off and retry with jitter for retryable failures.

Have you intentionally set appropriate retry and socket timeout configuration for all SDK usage?

Guidance

Not setting the appropriate retry and timeout logic for your AWS SDK clients can lead to a thread pool with all threads engaged in dependency operations. It's better to fail fast and a return a response to the client for dependency calls made within the context of sync calls from customers to let the client decide how and when to retry then timeout customer requests.

What throttling techniques are you using to defensively protect your service from customers ?

Guidance

See Fairness in multi-tenant systems

Are you using distributed throttling on your front-end?

Do you have pre-authentication throttles?

Are limits on request size enforced before authentication?

If your service was temporarily deactivated or shut down, what is your recovery time objective (RTO) for restarting your service?

Guidance

Have you practiced it? Do you have a runbook for restarting your service? Are there any off-box dependencies that are mandatory for restart? Have you confirmed that there are no circular dependencies?

Release quality

Do your customer impacting deployments automatically rollback incorrect deployments before they breach your internal SLAs?

Guidance

AWS CodeDeploy: [Automatic Rollback on Amazon CloudWatch Alarm](#)

AWS Lambda: Possible with [Traffic Shifting using CodeDeploy](#), supported in [Serverless Application Model](#). Specify a list of CloudWatch Alarms in the `DeployPreference` property [Lambda deployments with Automatic Rollback](#).

☐ Yes | No Risk

☐ No | High Risk

Do your customer impacting deployments run on-host validation tests to verify that the software has started successfully and is responding correctly to health checks on `localhost` before reregistering with the load balancer?

Guidance

CodeDeploy: Write validation scripts and execute them from the `ValidateService` [lifecycle event hook](#)

Lambda: Select Not Applicable

☐ Yes | No Risk

☐ No | High Risk

☐ Not Applicable

Do you have a mechanism in place to ensure all code changes (software, configuration, infrastructure, canary, and ops tools) to production systems are reviewed and approved by someone other than the code author?

Guidance

You may choose to add a manual approval action into your AWS CodePipeline pipelines and limit permissions to a set of approvers.

☐ Yes | No Risk

☐ No | High Risk

What is your load test plan?

Guidance

You should assume that you will find the breaking point of your service multiple times, iteratively addressing uncovered performance bottlenecks and repeating the load test. It should consider a small number of very large customers, a large number of very small customers, and sinusoidal load.

Conduct a load test that simulates a surge of traffic from a single customer to validate behavior under this kind of load. Conduct one large scale load test against your production environment (a) before you launch, and (b) subsequently during each quarter, to validate proper scaling as you grow (or for any potential peak usage).

Do you publish your canary synthetics errors to an independent metric? Subsequently, do you alarm on this metric?

Guidance

Ensure that your canary synthetics errors are published to their own metrics, as opposed to being combined with all errors. This allows your service to alarm on an individual canary error rate.

☐ Yes | No Risk

Do you publish your canary synthetics errors to an independent metric? Subsequently, do you alarm on this metric?

☐ No | High Risk

Event management

Have you performed a gameday to verify that your service's monitoring and alarming function as expected and your on-call engineers are engaged and able to rapidly diagnose and remediate failures?

Does your canary synthetic tests detect and alarm on shallow API test failures in under five minutes?

Do you monitor (and alarm on) your JVM statistics? Do you monitor (and alarm on) your hosts for file system, `inode`, and file descriptor utilization? Do you monitor (and alarm on) your hosts for CPU and memory utilization?

When do you look at your weekly and operator dashboards?

Guidance

Here's an example schedule for an ops meeting:

- Review outstanding action items from the previous week.
- Review last week's high severity tickets.
- Review pipelines for things like rollbacks or blocks.
- Review open customer support tickets.
- Review open high severity tickets.
- What new runbook entries were added this week?

When do you look at your weekly and operator dashboards?

- Review the detailed metrics dashboard for one of your components.

- Discuss on-call rotation.

Does your operational dashboard contain a view with metrics for critical dependencies?

Do your performance synthetics measure P50, P99, and P99.9s to track variability (including tail latency)?

Guidance

Performance variability should be measured along with median performance since there are edge cases which can affect both overall performance as well as the customer perception. Understanding this variability will allow your service to improve customer experience.

Do you have the ability to weight your workload out of an AZ within 15 minutes in the event of an issue, and do you have a runbook that clearly documents the process?

Guidance

Choose Not Applicable only if your workload has no AZ weight away requirements.

☐ Yes | No Risk

☐ No | High Risk

☐ Not Applicable

Would your workload be able to withstand the loss of an AZ without causing customer impact? Have you architected your workload to be statically stable during an AZ failure so that you do not have to make changes or deploy new capacity in response?

☐ Yes | No Risk

☐ No | High Risk

Contributors

Contributors to this document include:

- Michael Haken, Principal Solutions Architect, Amazon Web Services
- Arvind Raghunathan, Sr. Specialist Technical Account Manager, Amazon Web Services
- Vincent Brancato, Director of Safety Engineering, Amazon Web Services
- Kris Morton, General Manager of Incident Prevention Tools, Amazon Web Services

Further reading

For additional information, see:

- [AWS Well-Architected](#)
- [AWS Architecture Center](#)
- [Operational Readiness Review Template: Towards Operational Excellence](#) article

Document history

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Minor updates	Corrected risks associated with example questions.	July 14, 2022
Initial publication	Whitepaper first published.	June 30, 2022

> **ⓘ Note**
>
> To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser that you are using.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.