aws marketplace

EBOOK

Architecting Your Cloud-Native Journey with AWS

Concepts, strategies, and tools
for building modern cloud applications

aws marketplace

ARCHITECTING YOUR CLOUD-NATIVE JOURNEY WITH AWS

Table of contents

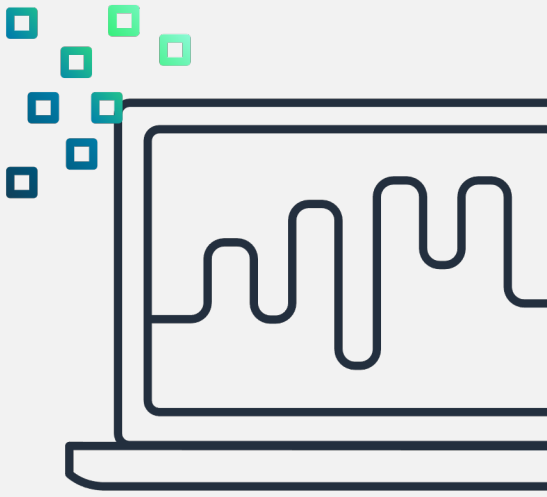
Why enterprises need to modernize their application development strategy

Today's engineering teams are under pressure to quickly and affordably deliver innovative products and solutions that can easily scale and evolve with customer demand. To support these intensifying business goals, software systems architectures are changing rapidly, enabling exciting new capabilities but also introducing complexities that drive a need for advanced development strategies.

Organizations that leverage modern cloud technology and operational models are able to innovate their applications faster and compete more effectively than those that still rely on legacy systems and processes. How? Let's first take a look at some of the shortcomings of the "old way."

It's also important to note that since everything is interdependent, the impact for each change can be far-reaching, making it difficult to adopt new technologies as your business seeks to grow.

Legacy systems rely on a monolithic approach to application development, meaning they try to do everything and are tightly coupled. When there are many interdependencies, releasing even a small change can make it necessary to rebuild or recompile the entire application. This often leads to fragile software that scales very rigidly and requires over-provisioning to handle increased loads.



aws marketplace

Pathways to cloud-native—the advantage of modern applications

To delight customers and win new business, organizations need to build reliable, scalable, and secure applications that create big customer value. That means breaking down and decoupling monolithic systems into smaller functional services—or microservices—that focus on one thing and do that one thing very well.

This drives a new way in which teams work by giving them the autonomy to architect, develop, deploy, and maintain each microservice. A major advantage of working this way is that teams can iterate and learn much faster by making lots of little changes to drive incremental innovation rather than pushing out a number of large changes on a longer schedule.

The business value of cloud-native at a glance

Amazon Web Services (AWS) benchmarked 22 unique KPIs to understand the value of using modern cloud services and found that organizations reported:

28%

revenue growth after adopting containers

39%

reduction in IT spend after adopting serverless

13%

faster time to resolve security incidents after adopting managed data

35%

faster time to insights after adopting managed analytics

43%

revenue growth after adopting all four of these pathways

According to internal AWS research

Starting on your cloud-native journey

Organizations are moving workloads to the cloud, using a cloud-native microservices approach to take advantage of the cloud's elasticity and platform and software services. They are modernizing their cloud infrastructure and leveraging services beyond basic compute and storage. That means adopting new technologies, practices, and consuming services as APIs.

An application programming interface (API) provides a set of defined rules that enables two or more computer programs to communicate with each other. An API acts as an intermediary layer that processes data transfers between systems, enabling organizations to make their application data and functionality available to third-party developers and other internal teams.

The takeaway? As software is evolving, you don't want to reinvent the wheel with every change. Cloud-native empowers developers to solve problems with existing environments and tools.

But what is cloud-native and why does it matter?

The Cloud Native Computing Foundation (CNCF) offers this definition:

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The AWS approach to cloud-native takes the concept further:

1. **First**, AWS is already reinventing what application development looks like today—all the software that is being built needs a place to run and a platform that enables all the pieces to work cohesively as a system. **AWS provides the infrastructure, storage, and compute services you need to achieve your goals.**

2. **Second**, at AWS, we have a broad set of capabilities in a family called serverless. **Serverless technologies remove the heavy lifting associated with running, managing, and maintaining servers so you can focus on core business logic and quickly adding value.**

3. The **third** point relates to scale. As an organization, Amazon has gone through and addressed the different problems that other organizations have or will have with scale. **The services that you see in the AWS and AWS Partner product catalogs are a direct reflection of that.**

AWS managed services for cloud-native development

Amazon engineers use AWS SDKs that have features such as retries, exponential backoff, circuit breakers, and jitter. This removes the obligation to deal with lower-level issues such as transient failures or backpressure because the SDK handles this logic for you in a manner that scales and doesn't cause additional problems that cascade downstream. There are many AWS tools that help remove the heavy lifting of managing and maintaining services, including but by no means limited to:

Amazon Elastic Kubernetes Service (EKS)
Kubernetes infrastructure management

Amazon EventBridge
Event bus management

Amazon Simple Notification Service (SNS)
Pub/sub system management

Amazon Simple Queue Service (SQS)
Fully managed message queues

Amazon Kinesis
Event and data streaming pipelines

aws marketplace

Where is your organization in its cloud-native journey?

Understanding where you are in your cloud-native journey and the capabilities you need to build to take full advantage of cloud-native is essential. Capabilities built across teams, roles, and the organization work together to get you there.

AWS has developed a Cloud-Native Maturity Matrix to help you learn about the specific capabilities you need to gain and where they fit into a cloud-native development strategy. Read on to understand the concept behind the matrix. Use the matrixes in the following pages to identify your progress in these three important categories:

1. People

2. Processes

3. Technology

AWS Marketplace Cloud-Native Maturity Matrix

Download full version here >

The capabilities on the left-hand side of each matrix are borrowed from a modern DevOps-centric approach, as DevOps and cloud-native go hand-in-hand. DevOps is about speed and agility, with some great side effects such as cost savings if done well. Marrying DevOps and cloud-native is a natural progression as DevOps practices only get better with cloud-native tooling.

As an example, consider the row labeled "Everything as Code." It's in the **People** matrix because Everything as Code actually starts with people. You'll notice that going from L1 - L5 the tooling is flexible, but from a people maturity standpoint you start to graduate from defining infrastructure to configuration and eventually to inner sourcing. Once you get to L5 **inner sourcing**, you realize that this culture is all about maturing the organization and shifting to a sharing and contributing mindset, which is why Everything as Code is included here.

Capability	L1: Initial	L2: Piloting	L3: Adoption	L4: Maturity	L5: Evolution
Leadership **Does leadership understand the need to adopt cloud-native DevOps tools and practices?**	Leadership agrees to support discovery and training	Department-level leaders initiate trial use of new tools and processes	Initiatives are implemented and metrics are shared with other teams	Standardized DevOps process from beginning to end of the dev process	Cross-organization leaders collaborate to more efficiently achieve common goals
Upskilling **Are employees equipped with the knowledge, skills, and tools required to take advantage of new cloud-native technologies?**	Attract employees who want to learn about cloud-native strategies	Develop cloud-native experts who are hungry to hone their skills	Cloud-native champions on teams train employees	Needs assessment and training—mentoring and workshops	Continuous improvement of capabilities
Communication and Collaboration **How can employees and leadership uplevel the efficiency with which they communicate and collaborate?**	Set up tools to facilitate communication (Slack, Jira, Asana)	Intra-team communication and collaboration	Active cross-team integration	Collaboration across org with metrics to identify areas of improvement	Well-established communication mechanisms across the org
Everything as Code **How do your teams build on the success of other teams and initiatives and avoid unnecessary redundancies?**	Define infrastructure using Terraform, CloudFormation, CDK, or Pulumi	Infrastructure as code deployed through a pipeline and fully tested	Configuration through code and deployed via pipeline	Everything defined as code and checked into a repository deployed through a pipeline or GitOps	Inner source code shared throughout org and anyone can update and improve through pull requests

The term inner sourcing references the practice of sharing what is already available in your organization. Think of it like open source but within the boundaries of your organization. Large enterprises often have access controls and systems that are built for specific groups and projects. And sometimes they don't open access to other groups, who then won't be able to benefit from them. From an inner sourcing point of view, you make all systems and data available so other teams can use it on their own projects, eliminating redundancies and hastening product delivery.

The **Process** aspect of the maturity model is where organizations look to improve process-related capabilities such as change management, lean delivery teams, and chaos engineering. Just like the last aspect (People), maturity goes from left to right and, as the organization matures, they are strengthening their capabilities and moving closer to cloud-native DevOps practices.

As an example, let's look at the capability to experiment. An individual or team would start testing features in a non-production environment. This initial step is simply to start the iterative process and learn from it. The next stages go from basic feature flags to dynamically turning on and off features to gathering metrics and to create the feedback loop. These stages are an important maturity step in not only maturing the process but also building confidence and promoting a culture of experimentation.

Capability	L1: Initial	L2: Piloting	L3: Adoption	L4: Maturity	L5: Evolution
Experimentation **Where is your organization in its ability to try new strategies and technologies?**	Test features—non-production	Ability to turn experiments on or off	Dynamic feature flags without system restart	Data & metrics prove worth	Run controlled experiments in production
Continuous Deployment **How close are you to being able to automate the process of pushing code into production?**	Deploy to lower-level environments	Schedule delivery process	Continuous integration and delivery	Gated deployment to production	Deployments from dev to production are fully automated
Change Management **How easily can you adapt processes and strategies to evolving needs?**	Change management system in place	Integrated bug tracking	System integrated with CI/CD pipelines	Automated creation of change control record	Automated change management process
Continuous Feedback **How can you evaluate the effect of each release on user experience to improve future releases?**	Telemetry from application	Telemetry from systems	End-to-end tracing	Signals and metrics across entire dev lifecycle	SaaS-based observability & AIOps

At the final level of maturity, organizations are running experiments in production, providing feedback to all stakeholders to help everyone make data-driven business decisions.

The last aspect of the maturity model is **Technology**. Here you focus on gaining capabilities that allow the organization to improve the tooling and overall technology that it uses.

Here, we'll use DevSecOps as our example. This capability is all about shifting left, shifting right, and making security a priority for the entire organization. Levels 1, 2, and 3 focus on getting security checks at the developer level and into the Continuous Integration/Continuous Delivery (CI/CD) pipeline. At levels 4 and 5, shift to other aspects of security such as policy and having security ingrained throughout the organization and providing feedback not only to security teams but to other stakeholders such as developer, product management, project management, and IT Operation teams.

Capability	L1: Initial	L2: Piloting	L3: Adoption	L4: Maturity	L5: Evolution
Continuous Delivery **Adopt a practice where code changes are automatically built, tested, and prepared for a release to production**	Define manual process for building software	Regular automated build and testing. Any builds can be recreated from source	Automated build and test cycle every time a change is committed	Build metrics gathered, made visible and taken into account	Continuous work on process improvement, better visibility, faster feedback
Cloud-native Architecture **Implement immutable infrastructure, microservices, declarative APIs, and containers**	Decoupled monolith	Distributed design	Strangler pattern	Leverages service abstractions such as, Amazon SQS, Amazon SES, Amazon SNS, and Amazon EventBridge	Event-driven architecture; microservices
DevSecOps **Integrate security initiatives at every stage of development**	Code is scanned at the repo level	Code checks are done in the developer's environment within the IDE	Code is checked at all stages of the CI/CD pipeline	Policy checks are integrated with repos and CI/CD	Integrate security with every phase of app delivery–from developer workstation to production
Observability **Use tools that help collect and analyze performance data to glean real-time insights**	Application and infrastructure monitoring	Shared dashboard and alerting capabilities	End-to-end visibility for additional use cases such as end-user experience monitoring	Analytics & intelligence provide data to business stakeholders	AIOps, surface anomalies, reduced signal to noise, automated remediation

Shift-left simply means to perform tests and validations earlier in the CI/CD pipeline—effectively, to shift them left. Shift-left helps engineering teams identify vulnerabilities earlier, reducing both their time to market and cost of deployment. In DevSecOps, shifting both left and right means implementing security at every stage of the application development lifecycle.

Key strategies—the journey to modernization

As an adjunct to the maturity matrixes, it's helpful to get a holistic view in the form of a cloud-native roadmap. The diagram below illustrates the step-by-step progression of overarching functionalities in which you need proficiency to reach cloud-native maturity.

Let's take a deeper look at each phase of this journey and identify the associated capabilities for each one.

1. Build and Architect:

Setting the stage for modernization

At this point in the journey, it's important to develop capabilities that allow development teams to build and iterate quickly. This is when you begin building the foundation for processes such as Continuous Integration/Continuous Delivery (CI/CD), test automation, Infrastructure as Code (IaC), Observability, and even feature flags. You may feel a bit hesitant to start to develop these capabilities, but now is the best time to build a development culture around these. If you wait too long, you'll end up accumulating a lot of **technical debt**.

In software development, technical debt is the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer. Analogous with monetary debt, if technical debt isn't repaid, it accumulates "interest," making it harder to implement changes down the line.

AWS Serverless Application Model: Bringing the cloud to the developer's workstation

An important tool with which to familiarize yourself at this stage of the cloud-native roadmap is AWS Serverless Application Model (SAM), which is an open-source framework that you can use to build serverless applications on AWS.

Associated capabilities for the Build and Architect phase include:

- Experimentation
- Continuous Feedback
- Lean Delivery Teams
- Upskilling Employees
- Continuous Integration
- Deployment Automation
- Continuous Delivery
- Observability
- Cloud-native Architecture

There are two components of SAM: The template and the SAM command line interface (CLI). The template allows you to describe the functions, APIs, permissions, and events that make up a serverless application. In the example below, we have 20 lines of code that—when paired with the SAM CLI—enables you to deploy the components on the right-hand side of the diagram and stitch all the interdependencies together with a simple deploy.

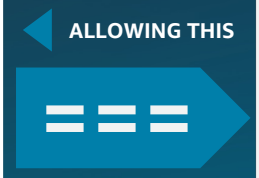
SAM enables developers to quickly build, test, experiment, and iterate without having to do all the heavy lifting and makes it easier to understand how all the pieces fit together on AWS. SAM also has a mode where you can develop and test locally on your workstation. The SAM CLI spins up a local API gateway and runs your function for you so you can do some testing locally using one of your favorite tools such as Postman, Insomnia, or Curl. When you are done testing, you run a deploy command that deploys everything into an AWS environment. The SAM template is highly configurable—as a team you can create custom templates that capture your team's best practices and patterns.

A typical SAM workflow:

1. Run SAM INIT
2. Answer the question prompts
3. Edit or replace the handler function and template
4. Run SAM locally
5. Test locally
6. Run SAM deploy to deploy to AWS

Code snippet from SAM

```
1  AWSTemplateFormatVersion:  '2010-09-09'
2  Transform: AWS: :Serverless-2016-10-31
3  Resources
4    GetProductFunction
5      Type: AWS: :Serverless: :Function
6      Properties
7         Handler: index.getProducts
8         Runtime: nodejs12.x
9         CodeUrl: src/
10        Policies:
11             DynamoDBReadPolicy
12                TableName: !Ref ProductTable
13        Events:
14           GetResource:
15              Type:Api
16              Propertites:
17                 Path:/products/{productid}
18                 Method:get
19   ProductTable:
20     Type: AWS: :Serverless: : SimpleTable
```



aws marketplace

2. Everything as Code:

Optimizing communication and collaboration

Step two brings us to an interesting capability that is directly related to how teams communicate and collaborate and how those aspects are improved by implementing an Everything-as-Code (EaC) approach. EaC refers to the idea of managing all aspects of software development as code. When you talk about automation, the first thing that gets automated is infrastructure. But there are a lot of other changes that happen on your path to production, and all of those changes need to be treated with the same rigor as application code or infrastructure. DNS entries, SSL cert automation/renewals, policies, schema changes, pipelines, and configurations are a few of the changes that need to be maintained as code in a version control system as well.

As an example, picture a team that needs to create new systems and applications as it modernizes and breaks down its monolithic development approach into microservices. Traditionally, application architects would collaborate with developers and IT Ops teams to build the requirements in an Excel sheet, and then create a ticket in Jira. The resulting base image then needs approval from security and compliance teams before it is made available. This takes weeks if not months of iterations to get a "golden image" so the team can proceed to the next step of development. With this approach, any new app that needs to be built has to go through months of planning and processing before all teams have approved. This, of course, negatively impacts the ability to stay competitive and keep customers engaged.

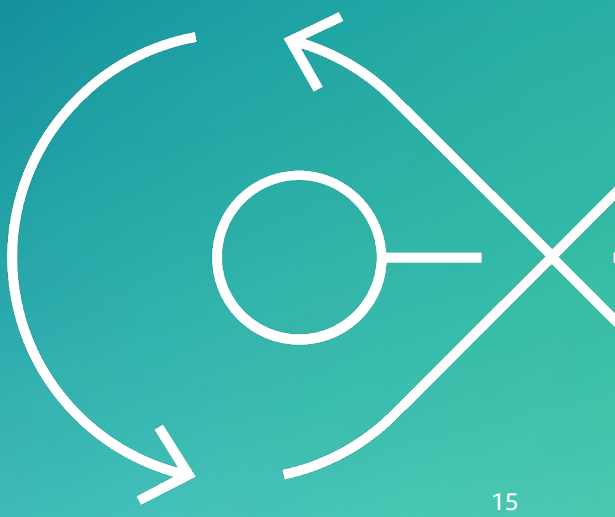
Associated capabilities for the Everything as Code phase include:

- Change Management
- Lean Delivery Teams
- Upskilling Employees
- Communication and Collaboration
- Deployment Automation
- Continuous Testing
- Continuous Delivery
- DevSecOps

A better approach is to implement Infrastructure as Code and GitOps to more effectively and expeditiously communicate requirements and collaborate across teams comprised of developers, architects, ITOps engineers, and security and compliance personnel. With standard templates for most common apps, cross-functional teams can collaborate and communicate on the code repositories, not only improving the speed of delivery but also helping avoid any environment drift. And with the elasticity of the cloud, capacity and budget planning cycles are also significantly reduced. What used to take months if not years is now done in just a few days thanks to the everything-as-code approach.

GitOps is a fast and secure method for developers to manage and update complex applications and infrastructure running in Kubernetes. GitOps increases agility for developers deploying and managing their cloud-native stack.

With GitOps, the entire stack is described declaratively, the desired state of the whole system or its building blocks is versioned in Git, and approved changes are automatically and reliably applied to the runtime environment.



aws marketplace

3. Continuous Delivery:

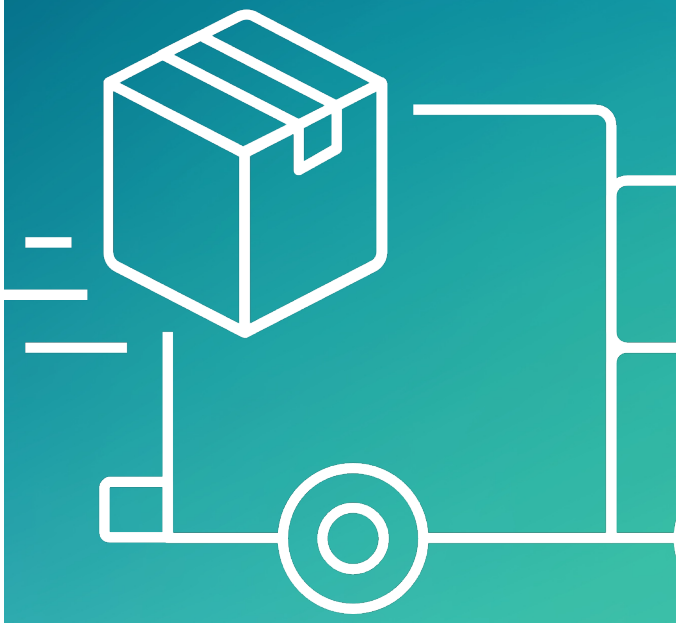
Unlocking speed by making smaller changes

In this phase, it's time to begin thinking about how to take your existing automation capabilities to the next level, improving processes from development to production. This is what Continuous Delivery is all about.

One of the key capabilities related to Continuous Delivery is the Continuous Feedback mechanism. When it comes to delivering your application into production, there are a lot of personas involved—product management, developers, security teams, operations teams, QA teams, performance teams—and everyone needs feedback from your Continuous Delivery pipeline to make improvements. Feedback and transparency also help build trust in your processes and tooling. So, feedback becomes a critical feature of Continuous Delivery and there is a responsibility to make sure that data is available for all of these personas.

Associated capabilities for the Continuous Delivery phase include:

- Change Management
- Continuous Feedback
- Communication and Collaboration
- Continuous Delivery
- DevSecOps
- Observability



Continuous Delivery versus Continuous Deployment

With Continuous Delivery, deployments are scheduled. So, if you have a weekly, biweekly, or monthly cadence, that's where you pause and wait for that time or change window to come and only then do you release the change. But when it comes to Continuous Deployment, the last time any manual intervention happens is at the point where the code is merged to the mainline, or the main production branch in code. Everything after this is fully automated all the way into production. There's no waiting for any kind of cadence or schedule like there is in Continuous Delivery.

Continuous Delivery

Continuous Deployment

The key takeaway for Continuous Delivery is that to unlock speed you have to make smaller changes. It may sound counter-intuitive, but smaller changes are easier to release to production.

There is the additional benefit that if something goes wrong you know exactly what caused the issue instead of trying to figure out what change among potentially hundreds caused it. Take a single commit and push it all the way into production. When you are doing multiple pushes into production each week, you more quickly hone your processes as opposed to doing larger pushes only once a quarter.

aws marketplace

4. Observability:

Metrics, events, logs, and traces (MELT)

Here we look at the concept of Continuous Feedback from a different perspective. Migrating and modernizing on the cloud gives you flexibility but can also introduce a lot of complexity if not managed properly. This is because your applications are decoupled and distributed.

For example, your infrastructure that used to run on a server now may be broken into microservices that are either just AWS Lambda functions on serverless or are running on containers. Or it's running on a third-party API that you're leveraging as a managed service. So, your application is now decoupled and distributed, and if you need to trace a transaction from end to end, you're going to need to have a complete picture of your systems, applications, and infrastructure—you need observability.

Observability is a key capability that you want to start practicing early on, when you start writing your code. The four key pillars of Observability are: Metrics, events, logs and traces (commonly known as MELT). OpenTelemetry and AWS Distro for OpenTelemetry (ADOT) provide standardized ways for organizations to instrument their applications to gain full visibility across their environments.

The four key pillars of Observability are:

Metrics

Events

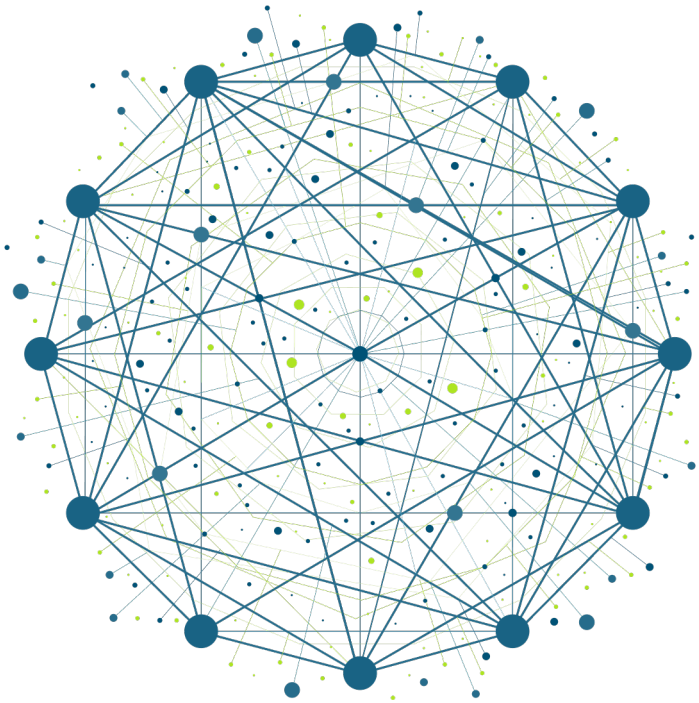
Logs

Traces

aws marketplace

Many technology companies have begun adopting OpenTelemetry and have included this capability into their software, essentially building observability into their product.

As a way to understand the importance of Observability, imagine the sphere on this page is a collection of microservices. How would you trace transactions through these systems? By collecting MELT data with cloud-native observability tools, you gain a complete picture and visibility into your development efforts. The key takeaway is that by instrumenting ADOT into your application, you'll get visibility into all the components that are supported by AWS and AWS Partners. This not only allows you to standardize, but also to pick up telemetry data from AWS and partners that support this functionality.



AWS Distro for OpenTelemetry (ADOT): Secure, production-ready, open-source distribution with reliable performance.

With ADOT you can:

- Instrument applications just once, then send metrics and traces to multiple AWS monitoring solutions
- Speed up performance troubleshooting and reduce mean time to resolution with increased visibility into your AWS resources
- Optimize performance, lower costs, and innovate faster with AWS support
- Validate code and components against security requirements with rigorous testing

aws marketplace

5. Modern Data Management:

The importance of purpose-built databases

As a progression from the previous stages of the journey, the next immediate capabilities that you want to build relate to how you manage your data. When you're modernizing your application stack, it's initially easy to break the monolith apart and create a few smaller microservices. But then you quickly realize that everything is tied together again at the database, defeating the purpose of decoupling your apps. You need to break down your monolithic database into business domain-specific databases that run on a purpose-built database.

This all leads to the necessity of having purpose-built databases, which are optimized around the type of data, relationships, and access patterns that solve a specific challenge. As a developer, the natural go-to database tends to be relational database management systems (RDBMS), and this makes sense as most developers are good with SQL and know how to build schemas, tables, entities, relationships, and normalize.

But this is a one-size-fits-all type of approach and SQL databases might not be the most effective way to solve a problem. Take for example data that has many relationships and interconnections. You might want to query this database to understand the number of relationships to a specific entity or even traverse a hierarchical relationship to understand all the interdependencies. An RDBMS can handle this to a certain extent, but as the number of one-to-many and many-to-one relationships grow, it becomes cumbersome to manage foreign keys and you may end up in a situation in which a specific entity could have hundreds of foreign keys.

Associated capabilities for the Modern Data Management phase include:

- Experimentation
- Continuous Deployment
- Change Management
- Continuous Feedback
- Lean Delivery Teams
- Communication and Collaboration
- Everything as Code
- Continuous Integration
- Deployment Automation
- Continuous Testing
- Continuous Delivery
- Cloud-Native Architecture

For this type of data, it might be better to use a purpose-built graph database. This is especially true when you want to do things like traversals or even query to understand the distance or number of edges between different vertices. A real-world example would be a social graph designed to help us understand the relationships between people and their interconnections.

It is important to think about data management and its related capabilities at this point in the journey so you can modernize your data along with your applications. You also want to think about democratizing your data and making it available to all consumers that need it. A key capability related to this is continuous testing. You need good quality data for your tests—if your data is bad, then your results are not reliable and the confidence in your pipeline will go down, making it nearly impossible to automate deployments.

The key takeaway here is to have a good understanding of the data and how applications are going to consume this information, then choose a database model that is purpose-built for the queries you plan to run against that data.

aws marketplace

6. DevSecOps:

Removing security as a bottleneck to development

Once you solve your infrastructure and database challenges, security testing is often still a manual sign-off before deploying to production. In this way, security becomes a bottleneck in the pipeline.

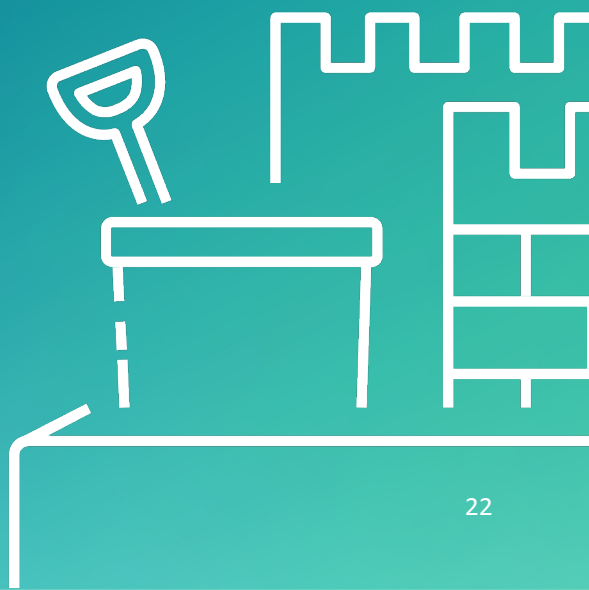
In modern development practices, security is everyone's responsibility. As a developer, you need to think about how to write secure code at the outset of the development process and as you develop features. You don't want that going into the pre-production environment where the security team is running a scan and reporting back to you weeks later—you may no longer have context to address the issue.

At this stage of the cloud-native journey, security teams should be integrating with Dev and Ops teams and defining OPA policies as code that forms guardrails.

Security scanning, checks, and policy guardrails need to be one of the key capabilities that the organization is building as part of this stage in the journey. And those processes need to be completely automated. At AWS, one of the ways security is scaled is to not only have guardrails and automated checks, but also have security champions on every development team. Security champions are developers, first and foremost, but these individuals work directly with the security team to learn about best practices and emerging trends in security.

Associated capabilities for the DevSecOps phase include:

- Continuous Deployment
- Change Management
- Continuous Feedback
- Lean Delivery Teams
- Upskilling Employees
- Communication and Collaboration
- Everything as Code
- Continuous Integration
- Deployment Automation
- Continuous Testing
- Continuous Delivery
- DevSecOps
- Observability



Open Policy Agent (OPA)

OPA is an open-source, general-purpose policy engine. OPA separates the policy decision-making from your software. What this means is that you can define a general policy for, say, no containers unless they are from your repo private.example.com. And any application that is integrated with OPA can ask OPA if this action can be allowed.

This example illustrates an OPA policy deployed in a file called constraint.yaml. The policy says to only allow private registries if the repository is private.example.com. We would deploy this policy definition using kubectl, a command-line tool for Kubernetes that lets you control or communicate with Kubernetes clusters or resources you create by using the Kube API. On the right-hand side of the illustration, we've enabled OPA Gatekeeper, which is a Kubernetes operator for OPA. When we deploy the container image myapp to Amazon Elastic Kubernetes Service (EKS), OPA Gatekeeper checks this against the policy that we applied earlier and allows the container to be deployed through the Kubernetes admission controller. But the image on the bottom, otherapp, is denied as it violates the policy because it is coming from a repository (public.example.com) that doesn't match the policies constraint.

This is just one example, but you can set OPA to decide what protocols to allow, such as allow https, but don't allow http or ssh protocols. Additionally, OPA is not specific to Kubernetes. It can be used to define policy for application authorization or naming standards for Git repository branches—basically anything for which you can define a policy.

Code snippet from constraint.vaml

```
apiVersion:
constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
metadata:
  name: allow-only-private-
registry
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
  parameters:
    repos:
      - "private.example.com"
```

aws marketplace

7. Continuous Deployment:

Deploy changes into production automatically

The goal for this stage of the journey is to gain those capabilities that allow all changes, whether those are application code, infrastructure code, OS patches, or configuration changes—basically anything that can make up the components of a microservice—to be deployed to production automatically. Having these capabilities allows engineers to focus on core business value versus running, maintaining, and managing deployments.

The last manual step in a Continuous Deployment pipeline is a code review by a peer, and once the code is merged into mainline, everything from there is fully automated, which means at this stage you have a very high level of confidence in the automation, systems, and tests that make up your development process. Any feature that gets merged into mainline will land in production automatically if it passes all the quality, security, and compliance checks.

Associated capabilities for the Continuous Deployment phase include:

- Continuous Deployment
- Change Management
- Continuous Feedback
- Lean Delivery Teams
- Deployment Automation
- Continuous Testing
- Cloud-Native Architecture
- Continuous Delivery
- Observability

One benefit of Continuous Deployment is that it forces all teams to collaborate on the pipeline. Functional tests, security guardrails, and compliance checks are all delivered as a service and the pipeline is now the consumer of these services. Each team is providing its expertise and requirements to the pipeline, and any change that makes it out of the pipeline is expected to be production-grade.

At Amazon, after a barrage of tests and series of promotions through lower-level environments, every production deployment starts with a strategy called “one-box.” This is a single virtual machine, a single container, or an AWS Lambda function in an AWS Region or Availability Zone. With one-box deployed, we collect metrics and other telemetry data to build confidence in the deployments. Automated alarms and thresholds are set to detect issues and if any adverse effect is detected, the system is designed to automatically roll back and alert an on-call engineer.

8. Everything as a Service:

Don’t rebuild what’s already available

At this point you should be thinking about how you create that domain-specific platform that the rest of the organization can leverage. Don’t just think about infrastructure and databases and other capabilities of the service—think about your own platform capabilities as a service. Build APIs with a strict contract mechanism that you need to follow, maintain, and respect. At this stage you should be exposing your business capabilities as APIs that others across the organization can use as building blocks.

Treat the microservice like a product and the consumers of the service like customers, even if the consumption is done by another microservice.

In cloud-native, a platform is anything that is not part of your business domain. Developers should already be somewhat familiar with this concept as it’s rare that anyone develops software from scratch. As a developer, you would use libraries and modules, or going further you might use a framework or a third-party API as part of the application you’re developing. So, in a cloud-native world you want to consume everything that’s not part of your business domain as a service.

Associated capabilities for the Everything as a Service phase include:

- Lean Delivery Teams
- Upskilling Employees
- Everything as Code
- Cloud-Native Architecture
- Observability

To use a “service” like this, a user would open a ticket to the email team requesting access to send email. This may sound like it fits into the realm of Everything as a Service but in reality, it’s an anti-pattern, something that works against your guiding principle.

A service shouldn’t require tickets and manual approvals to start using it —consumption should be self-service.

The other point, which is a big one, is that the organization shouldn’t be running, managing, and maintaining any platform services. In cloud-native, virtually all platform services are provided by AWS or partners. What should be provided as a service are business domain-specific services.

Between AWS and AWS Partners, we have you covered when it comes to platform services. We have databases, messaging buses, queuing systems, API gateways, caching systems, storage, testing, security, and much more. These platform services are robust and can scale, plus they’re built on a foundation of lessons AWS has learned over many years.

As an example, let’s imagine you’re developing an application and your application needs to send email. What you don’t want to do as an organization in this case is stand up an email server, build a team around it to run, manage, and maintain it, and then call it a service.

9. Cloud Operations:

Day two

We're at the final step of our long but worthwhile journey. We've developed a number of capabilities to secure our services, built the mechanisms needed to automatically deploy to production, and we've exposed our business domain as a service to the rest of the organization.

Chaos engineering and game days

Now we want to turn our attention to building the operational muscle. This is where you want to practice chaos engineering and game days. For those not familiar with these terms, chaos engineering is a practice used to test a service or services by causing failure scenarios. The tests are used to uncover issues with design or implementation before an unplanned event affects the service. Game days are scheduled events that the team uses to practice responding to incidents. Chaos engineering and game days are independent, but complementary.

At this stage in the journey, it's important to develop these capabilities because, as organizations mature, more services are automated and people start to forget the core manual processes they've learned to automate, which is one downside to automation. The answer is to practice, practice, practice—so when a real incident happens, everyone is fully prepared.

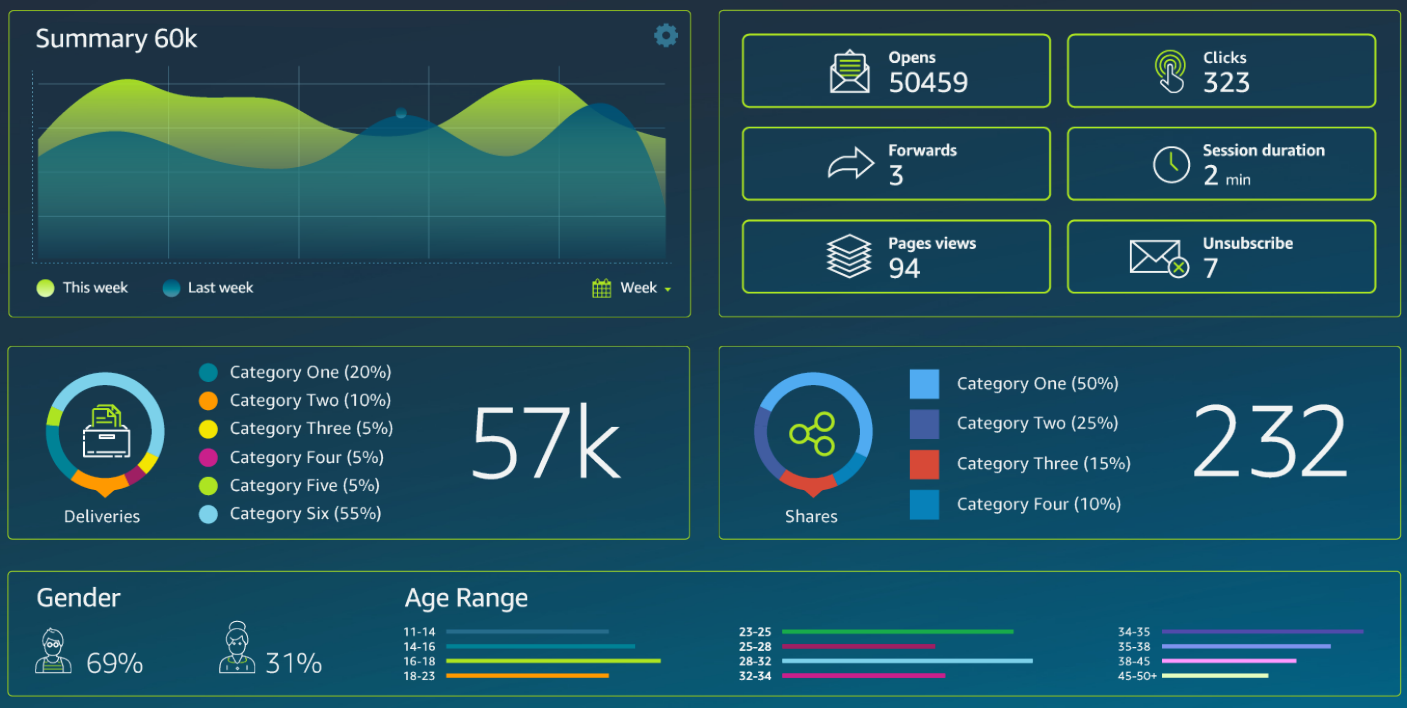
Associated capabilities for the Cloud Operations phase include:

- Experimentation
- Continuous Deployment
- Continuous Feedback
- Chaos Engineering
- Communication and Collaboration
- Continuous Testing
- Cloud-Native Architecture
- Observability

Dashboarding

Another important component of this phase is the capability of dashboarding. Dashboards are an often-overlooked tool that's an important part of Continuous Feedback. For most of us, dashboards are something we periodically use to check status, gather information, or if there is an incident, we use a dashboard to help us triage the problem.

Here at AWS, maintaining and updating dashboards is ingrained in our development process. Before completing changes, and during code reviews, our developers ask, "Do I need to update any dashboards?" Developers are empowered to make changes to dashboards before the underlying changes are deployed. This helps avoid a situation where an operator has to update dashboards during or after a system deployment to validate the changes being deployed.



Example of a dashboard displaying critical application metrics

aws marketplace

Tools to support your cloud-native journey

As you go through your journey to cloud-native there will be a number of key technologies to learn about to make sure you acquire the right capabilities. At AWS, we've long been believers in enabling builders to use the right tool for the job—and when you build on AWS you're provided with choice. Builders can use native services from AWS and also use AWS Marketplace to find, try, and acquire new tools from across the DevOps landscape.

To get started, visit: https://aws.amazon.com/marketplace/solutions/devops

AWS Marketplace

Third-party research has found that customers using AWS Marketplace are experiencing an average time savings of 49 percent when needing to find, buy, and deploy a third-party solution. And some of the highest-rated benefits of using AWS Marketplace are identified as:

Time to value

Cloud readiness of the solution

Return on Investment

Part of the reason for this is that AWS Marketplace is supported by a team of solution architects, security experts, product specialists, and other experts to help you connect with the software and resources you need to succeed with your applications running on AWS.

Over **13,000 products** from **3,000+ vendors**:

APPDYNAMICS
bridgecrew BY PRISMA® CLOUD
Quali
circleci
HashiCorp
JFrog

Buy through AWS Billing using flexible purchasing options:

- Free trial
- Pay-as-you-go
- Hourly | Monthly | Annual | Multi-Year
- Bring your own license (BYOL)
- Seller private offers
- Channel Partner private offers

Deploy with multiple deployment options:

- AWS Control Tower
- AWS Service Catalog
- AWS CloudFormation (Infrastructure as Code)
- Software as a Service (SaaS)
- Amazon Machine Image (AMI)
- Amazon Elastic Container Service (ECS)
- Amazon Elastic Kubernetes Service (EKS)

aws marketplace

Continue your journey with AWS Marketplace

Visit aws.amazon.com/marketplace to find, try and buy software with flexible pricing and multiple deployment options to build your ideal DevOps toolchain.

https://aws.amazon.com/marketplace/solutions/devops

Authors:

James Bland
Global Tech Lead for DevOps, AWS

Aditya Muppavarapu
Global Segment Leader for DevOps, AWS