Building hexagonal architectures on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Building hexagonal architectures on AWS

Table of Contents

Building hexagonal architectures on AWS

Furkan Oruc, Dominik Goby, Darius Kunce, and Michal Ploski, Amazon Web Services (AWS)

June 2022 (document history)

This guide describes a mental model and a collection of patterns for developing software architectures. These architectures are easy to maintain, extend, and scale across the organization as product adoption grows. Cloud hyperscalers such as Amazon Web Services (AWS) provide building blocks for small and large enterprises to innovate and create new software products. The rapid pace of these new service and feature introductions leads business stakeholders to expect their development teams to prototype new minimum viable products (MVPs) faster, so that new ideas can be tested and verified as soon as possible. Often, those MVPs are adopted and become part of the enterprise software ecosystem. In the process of producing these MVPs, teams sometimes abandon software development rules and best practices, such as SOLID principles and unit testing. They assume that this approach will speed up development and reduce the time to market. However, if they fail to create a foundational model and a framework for software architecture at all levels, it will be difficult or even impossible to develop new features for the product. Lack of certainty and changing requirements can also slow down the team during the development process.

This guide walks through a proposed software architecture, from a low-level hexagonal architecture to a high-level architectural and organizational decomposition, that uses domain-driven design (DDD) to address these challenges. DDD helps manage business complexity and scale the engineering team as new features are developed. It aligns business and technical stakeholders to the business problems, called *domains*, by using ubiquitous language. Hexagonal architecture is a technical enabler of this approach in a very specific domain, called a *bounded context*. A bounded context is a highly cohesive and loosely coupled sub-area of the business problem. We recommend that you adopt hexagonal architecture for all your enterprise software projects regardless of their complexity.

Hexagonal architecture encourages the engineering team to solve the business problem first, whereas classical layered architecture shifts engineering focus away from the domain to solving technical problems first. Furthermore, if software follows a hexagonal architecture, it's easier to adopt a test-driven development approach, which reduces the feedback loop that developers need to test business requirements. Lastly, using commands and command handlers is a way to apply the single responsibility and open-closed principles from SOLID. Adhering to these principles

produces a code base that developers and architects working on the project can easily navigate and understand, and reduces the risk of introducing breaking changes to existing functionality.

This guide is for software architects and developers who are interested in understanding the benefits of adopting hexagonal architecture and DDD for their software development projects. It includes an example of designing an infrastructure for your application on AWS that supports hexagonal architecture. For an example implementation, see Structure a Python project in hexagonal architecture using AWS Lambda on the AWS Prescriptive Guidance website.

Overview

Domain-driven design (DDD)

In domain-driven design (DDD), a domain is the core of the software system. The domain model is defined first, before you develop any other module, and it doesn't depend on other low-level modules. Instead, modules such as databases, the presentation layer, and external APIs all depend on the domain.

In DDD, architects decompose the solution into bounded contexts by using business logic-based decomposition instead of technical decomposition. The benefits of this approach are discussed in the Targeted business outcomes section.

DDD is easier to implement when teams use hexagonal architecture. In hexagonal architecture, the application core is the center of the application. It is decoupled from other modules through ports and adapters, and has no dependencies on other modules. This aligns perfectly with DDD, where a domain is the core of the application that solves a business problem. This guide proposes an approach where you model the core of the hexagonal architecture as the domain model of a bounded context. The next section describes hexagonal architecture in more detail.

This guide doesn't cover all aspects of DDD, which is a very broad topic. To gain a better understanding, you can review the DDD resources listed on the Domain Language website.

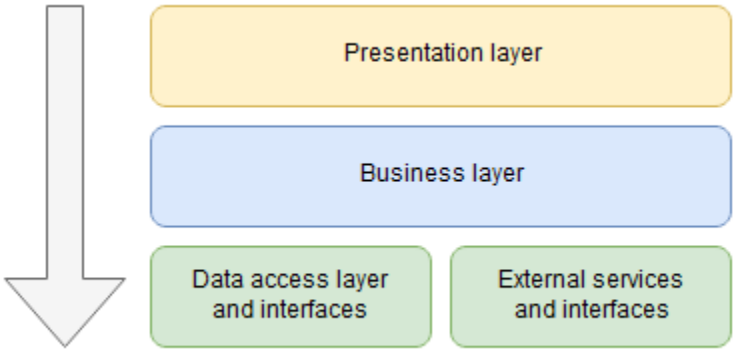
Hexagonal architecture

Hexagonal architecture, also known as *ports and adapters* or *onion architecture*, is a principle of managing dependency inversion in software projects. Hexagonal architecture promotes a strong focus on the core domain business logic when developing software, and treats external integration points as secondary. Hexagonal architecture helps software engineers adopt good practices such as test-driven development (TDD), which, in turn, promotes architecture evolution, and helps you manage complex domains in the long term.

Let's compare hexagonal architecture to classical layered architecture, which is the most popular choice for modeling structured software projects. There are subtle but powerful differences between the two approaches.

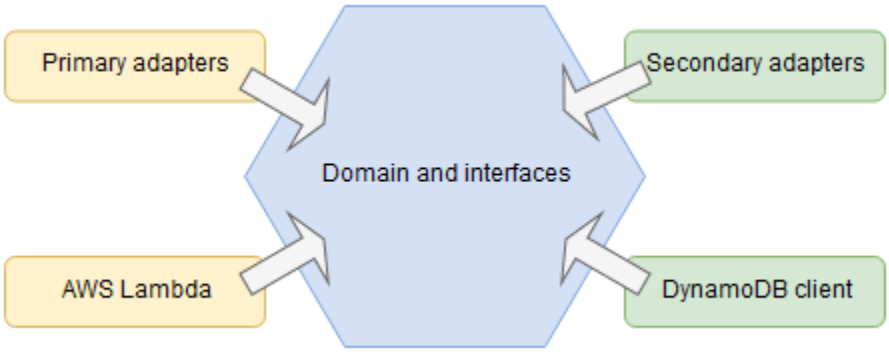
In layered architecture, software projects are structured in tiers, which represent broad concerns such as business logic or presentation logic. This architecture uses a dependency hierarchy, where

the top layers have dependencies on the layers below them, but not the other way around. In the following diagram, the presentation layer is responsible for user interactions, so it includes the user interface, APIs, command line interfaces, and similar components. The presentation layer has a dependency on the business layer, which implements domain logic. The business layer, in turn, has dependencies on the data access layer and on multiple external services.



The main disadvantage of this configuration is the dependency structure. For example, if the model for storing data in the database changes, this affects the data access interface. Any change to the data model also affects the business layer, which has a dependency on the data access interface. As a result, software engineers cannot make any infrastructure changes without affecting the domain logic. This, in turn, increases the likelihood of regression bugs.

Hexagonal architecture defines dependency relationships in a different way, as illustrated in the following diagram. It concentrates decision making around domain business logic, which defines all the interfaces. External components interact with the business logic through interfaces called *ports*. Ports are abstractions that define the interactions of the domain with the external world. Each infrastructure component must implement those ports, so changes in those components no longer affect the core domain logic.



The surrounding components are called *adapters*. An adapter is a proxy between the external world and the internal world, and implements a port defined in the domain. Adapters can be categorized in two groups: primary and secondary. Primary adapters are the entry points to the software component. They allow external actors, users, and services to interact with the core logic. AWS Lambda is a good example of a primary adapter. It integrates with multiple AWS services that can invoke the Lambda functions as entry points. Secondary adapters are external service library wrappers that handle communications with the external world. A good example of a secondary adapter is an Amazon DynamoDB client for data access.

Targeted business outcomes

The hexagonal architecture discussed in this guide helps you achieve the following objectives:

- Reduce time to market by improving the development cycle
- Improve software quality
- Adapt more easily to change

These processes are discussed in detail in the following sections.

Improving the development cycle

Developing software for the cloud introduces new challenges for software engineers, because it is very hard to replicate the runtime environment locally on the development machine. A straightforward way to validate the software is to deploy it to the cloud and test it there. However, this approach involves a lengthy feedback cycle, especially when the software architecture contains multiple serverless deployments. Improving this feedback cycle shortens the time to develop features, which reduces the time to market significantly.

Testing in the cloud

Testing directly in the cloud is the only way to make sure that your architectural components, such as gateways in Amazon API Gateway, AWS Lambda functions, Amazon DynamoDB tables, and AWS Identity and Access Management (IAM) permissions, are configured correctly. It might also be the only reliable way to test component integrations. Although some AWS services (such as DynamoDB) can be deployed locally, most of them cannot be replicated in a local setup. At the same time, third-party tools such as Moto and LocalStack that mock AWS services for testing purposes might not reflect real service API contracts accurately, or the number of features might be limited.

However, the most complex part of the enterprise software is in the business logic, not in the cloud architecture. The architecture changes less often than the domain, which must accommodate new business requirements. Thus, testing business logic in the cloud becomes an intense process of making a change in the code, initiating a deployment, waiting for the environment to be ready, and validating the change. If a deployment takes as little as 5 minutes, making and testing 10 changes in the business logic will take an hour or more. If business logic is more complex, testing might require days of just waiting for deployments to complete. If you have multiple features and engineers on the team, the extended period quickly becomes noticeable to the business.

Testing locally

A hexagonal architecture helps developers focus on the domain instead of infrastructure technicalities. This approach uses local tests (the unit testing tools in your chosen development framework) to cover domain logic requirements. You won't have to spend time solving technical integration problems or deploy your software to the cloud to test the business logic. You can run unit tests locally and reduce the feedback loop from minutes to seconds. If a deployment takes 5

minutes but unit tests complete in 5 seconds, that's a significant reduction in the time it takes to detect mistakes. The Testing business logic first section later in this guide covers this approach in more detail.

Parallelization of development

The hexagonal architecture approach enables development teams to parallelize development efforts. Developers can design and implement different components of the service individually. This parallelization is possible through the isolation of each component and the defined interfaces between each component.

Product time to market

Local unit testing improves the development feedback cycle and reduces the time to market for new products or features, especially when these contain complex business logic, as explained previously. Furthermore, increased code coverage by unit tests significantly reduces the risk of introducing regression bugs when you update or refactor the code base. Unit test coverage also enables you to continuously refactor the code base to keep it well organized, which speeds the onboarding process for new engineers. This is discussed further in the Quality by design section. Lastly, if the business logic is well isolated and tested, it enables developers to adapt quickly to changing functional and non-functional requirements. This is explained further in the Adapting to change section.

Quality by design

Adopting a hexagonal architecture helps promote the quality of your code base from the start of your project. It is important to build a process that helps you meet expected quality requirements from the beginning, without slowing down the development process.

Localized changes and increased readability

Using the hexagonal architecture approach enables developers to change code in one class or component without affecting other classes or components. This design promotes the cohesion of developed components. By decoupling the domain from adapters and using well-known interfaces, you can increase the readability of the code. It becomes easier to identify issues and corner cases.

This approach also facilitates code review during development and limits the introduction of undetected changes or technical debt.

Testing business logic first

Local testing can be accomplished by introducing end-to-end, integration, and unit tests to the project. End-to-end tests cover the whole incoming request lifecycle. They typically invoke an application entry point and test to see if it has accomplished the business requirement. Each software project should have at least one test scenario that uses known inputs and produces expected outputs. However, adding more corner-case scenarios can get complex, because each test must be configured to send a request through an entry point (for example, through a REST API or queues), go through all the integration points that the business action requires, and then assert the result. Setting up the environment for the test scenario and asserting results can take a lot of developers' time.

In hexagonal architecture, you test business logic in isolation, and use integration tests to test secondary adapters. You can use mock or fake adapters in your business logic tests. You can also combine the tests for business use cases with unit tests for your domain model to maintain high coverage with low coupling. As a good practice, integration tests should not validate business logic. Instead, they should verify that the secondary adapter calls the external services correctly.

Ideally, you can use test-driven development (TDD) and start defining domain entities or business use cases with proper tests at the very beginning of development. Writing the tests first helps you create mock implementations of the interfaces required by the domain. When the tests

are successful and domain logic rules are satisfied, you can implement the actual adapters and deploy software to the test environment. At this point, your implementation of domain logic might not be ideal. You can then work on refactoring the existing architecture to evolve it by introducing design patterns or rearranging code in general. By using this approach, you can avoid introducing regression bugs and you can improve the architecture as the project grows. By combining this approach with the automatic tests you run in your continuous integration process, you can decrease the number of potential bugs before they get to production.

If you use serverless deployments, you can quickly provision an instance of the application in your AWS account for manual integration and end-to-end testing. After these implementation steps, we recommend that you automate testing with every new change pushed to the repository.

Maintainability

Maintainability refers to operating and monitoring an application to ensure that it meets all requirements and minimizing the probability of a system failure. To make the system operable, you must adapt it to future traffic or operational requirements. You must also ensure that it's available and easy to deploy with minimum or no impact on clients.

To understand the current and historical state of your system, you must make it observable. You can do this by providing specific metrics, logs, and traces that operators can use to ensure that the system works as expected and to track bugs. These mechanisms should also allow operators to conduct root cause analysis without having to log in to the machine and read the code.

A hexagonal architecture aims to increase the maintainability of your web applications so that your code requires less work overall. By separating modules, localizing changes, and decoupling application business logic from adapter implementation, you can produce metrics and logs that help operators gain a deep understanding of the system and understand the scope of specific changes made to the primary or secondary adapters.

Adapting to change

Software systems tend to get complicated. One reason for this could be frequent changes to the business requirements and little time to adapt the software architecture accordingly. Another reason could be insufficient investment for setting up the software architecture at the beginning of the project to adapt to frequent changes. Whatever the reason, a software system could get complicated to the point where it is almost impossible to make a change. Therefore, it is important to build maintainable software architecture from the beginning of the project. Good software architecture can adapt to changes easily.

This section explains how to design maintainable applications by using hexagonal architecture that adapts easily to non-functional or business requirements.

Adapting to new non-functional requirements by using ports and adapters

As the core of the application, the domain model defines the actions that are required from the outside world to fulfill business requirements. These actions are defined through abstractions, which are called *ports*. These ports are implemented by separate adapters. Each adapter is responsible for an interaction with another system. For example, you might have one adapter for the database repository and another adapter for interacting with a third-party API. The domain is not aware of the adapter implementation, so it is easy to replace one adapter with another. For example, the application might switch from a SQL database to a NoSQL database. In this case, a new adapter has to be developed to implement the ports that are defined by the domain model. The domain has no dependencies on the database repository and uses abstractions to interact, so there would be no need to change anything in the domain model. Therefore, hexagonal architecture adapts to non-functional requirements with ease.

Adapting to new business requirements by using commands and command handlers

In classical layered architecture, the domain depends on the persistence layer. If you want to change the domain, you would also have to change the persistence layer. In comparison, in hexagonal architecture, the domain doesn't depend on other modules in the software. The domain is the core of the application, and all other modules (ports and adapters) depend on the domain

model. The domain uses the dependency inversion principle to communicate with the outside world through ports. The benefit of dependency inversion is that you can change the domain model freely without being afraid to break other parts of the code. Because the domain model reflects the business problem that you are trying to solve, updating the domain model to adapt to changing business requirements isn't a problem.

When you develop software, separation of concerns is an important principle to follow. To achieve this separation, you can use a slightly modified command pattern. This is a behavioral design pattern in which all required information to complete an operation is encapsulated in a command object. These operations are then processed by command handlers. Command handlers are methods that receive a command, alter the state of the domain, and then return a response to the caller. You can use different clients, such as synchronous APIs or asynchronous queues, to run commands. We recommend that you use commands and command handlers for every operation on the domain. By following this approach, you can add new features by introducing new commands and command handlers, without changing your existing business logic. Thus, using a command pattern makes it easier to adapt to new business requirements.

Decoupling components by using the service façade or CQRS pattern

In hexagonal architecture, primary adapters are responsible for loosely coupling incoming read and write requests from clients to the domain. There are two ways to achieve this loose coupling: by using a service façade pattern or by using the command query responsibility segregation (CQRS) pattern.

The service façade pattern provides a front-facing interface to serve clients such as the presentation layer or a microservice. A service façade provides clients with several read and write operations. It's responsible for transferring incoming requests to the domain and mapping the response received from the domain to clients. Using a service façade is easy for microservices that have a single responsibility with several operations. However, when using the service façade, it is harder to follow single responsibility and open-closed principles. The single responsibility principle states that each module should have responsibility over only a single functionality of the software. The open-closed principle states that code should be open for extension and closed for modification. As the service façade extends, all operations are collected in one interface, more dependencies are encapsulated into it, and more developers start modifying the same façade. Therefore, we recommend using a service façade only if it's clear that the service would not extend a lot during development.

Another way to implement primary adapters in hexagonal architecture is to use the CQRS pattern, which separates read and write operations using queries and commands. As explained previously, commands are objects that contain all the information required to change the state of the domain. Commands are performed by command handler methods. Queries, on the other hand, do not alter the state of the system. Their only purpose is to return data to clients. In the CQRS pattern, commands and queries are implemented in separate modules. This is especially advantageous for projects that follow an event-driven architecture, because a command could be implemented as an event that is processed asynchronously, whereas a query can be run synchronously by using an API. A query can also use a different database that is optimized for it. The disadvantage of the CQRS pattern is that it takes more time to implement than a service façade. We recommend using the CQRS pattern for projects that you plan to scale and maintain in the long term. Commands and queries provide an effective mechanism for applying the single responsibility principle and developing loosely coupled software, especially in large-scale projects.

CQRS has great benefits in the long term, but requires an initial investment. For this reason, we recommend that you evaluate your project carefully before you decide to use the CQRS pattern. However, you can structure your application by using commands and command handlers right from the start without separating read/write operations. This will help you easily refactor your project for CQRS if you decide to adopt that approach later.

Organizational scaling

A combination of hexagonal architecture, domain-driven design, and (optionally) CQRS enables your organization to quickly scale your product. According to Conway's Law, software architectures tend to evolve to reflect a company's communication structures. This observation has historically had negative connotations, because big organizations often structure their teams based on technical expertise such as database, enterprise service bus, and so on. The problem with this approach is that product and feature development always involve crosscutting concerns, such as security and scalability, which require constant communication among teams. Structuring teams based on technical features creates unnecessary silos in the organization, which result in poor communications, lack of ownership, and losing sight of the big picture. Eventually, these organizational problems are reflected in the software architecture.

The Inverse Conway Maneuver, on the other hand, defines the organizational structure based on domains that promote the software architecture. For example, cross-functional teams are given responsibility for a specific set of bounded contexts, which are identified by using DDD and event storming. Those bounded contexts might reflect very specific features of the product. For example,

the account team might be responsible for the payment context. Each new feature is assigned to a new team that has highly cohesive and loosely coupled responsibilities, so they can focus only on the delivery of that feature and reduce the time to market. Teams can be scaled according to the complexity of features, so complex features can be assigned to more engineers.

Best practices

Model the business domain

Work back from the business domain to the software design to ensure that that the software you're writing fits the business need.

Use domain-driven design (DDD) methodologies such as event storming to model the business domain. Event storming has a flexible workshop format. During the workshop, domain and software experts explore the complexity of the business domain collaboratively. Software experts use the deliverables of the workshop to start the design and development process for software components.

Write and run tests from the beginning

Use test-driven development (TDD) to verify the correctness of the software that you are developing. TDD works best at a unit test level. The developer designs a software component by writing a test first, which invokes that component. That component has no implementation at the beginning, therefore the test fails. As a next step, the developer implements the component's functionality, using test fixtures with mock objects to simulate the behavior of external dependencies, or ports. When the test succeeds, the developer can continue by implementing real adapters. This approach improves software quality and results in more readable code, because developers understand how users would use the components. Hexagonal architecture supports the TDD methodology by separating the application core. Developers write unit tests that focus on the domain core behavior. They don't have to write complex adapters to run their tests; instead, they can use simple mock objects and fixtures.

Use behavior-driven development (BDD) to ensure end-to-end acceptance at a feature level. In BDD, developers define scenarios for features and verify them with business stakeholders. BDD tests use as much natural language as possible to achieve this. Hexagonal architecture supports the BDD methodology with its concept of primary and secondary adapters. Developers can create primary and secondary adapters that can run locally without calling external services. They configure the BDD test suite to use the local primary adapter to run the application.

Automatically run each test in the continuous integration pipeline to constantly evaluate the quality of the system.

Define the behavior of the domain

Decompose the domain into entities, value objects, and aggregates (read about implementing domain-driven design), and define their behavior. Implement the behavior of the domain so that tests that were written at the beginning of the project succeed. Define commands that invoke the behavior of domain objects. Define events that the domain objects emit after they complete a behavior.

Define interfaces that adapters can use to interact with the domain.

Automate testing and deployment

After an initial proof of concept, we recommend that you invest time implementing DevOps practices. For example, continuous integration and continuous delivery (CI/CD) pipelines and dynamic test environments help you maintain the quality of the code and avoid errors during deployment.

- Run your unit tests inside your CI process and test your code before it is merged.
- Build a CD process to deploy your application into a static dev/test environment or into dynamically created environments that support automatic integration and end-to-end testing.
- Automate the deployment process for dedicated environments.

Scale your product by using microservices and CQRS

If your product is successful, scale your product by decomposing your software project into microservices. Utilize the portability that hexagonal architecture provides to improve performance. Split query services and command handlers into separate synchronous and asynchronous APIs. Consider adopting the command query responsibility segregation (CQRS) pattern and event-driven architecture.

If you get many new feature requests, consider scaling your organization based on DDD patterns. Structure your teams in such a way that they own one or more features as bounded contexts, as discussed previously in the Organizational scaling section. These teams can then implement business logic by using hexagonal architecture.

Design a project structure that maps to hexagonal architecture concepts

Infrastructure as code (IaC) is a widely adopted practice in cloud development. It lets you define and maintain your infrastructure resources (such as networks, load balancers, virtual machines, and gateways) as source code. This way, you can track all changes to your architecture by using a version control system. In addition, you can create and move the infrastructure easily for testing purposes. We recommend that you keep your application code and infrastructure code in the same repository when you develop your cloud applications. This approach makes it easy to maintain infrastructure for your application.

We recommend that you divide your application into three folders or projects that map to the concepts of hexagonal architecture: `entrypoints` (primary adapters), `domain` (domain and interfaces), and `adapters` (secondary adapters).

The following project structure provides an example of this approach when designing an API on AWS. The project maintains application code (`app`) and infrastructure code (`infra`) in the same repository, as recommended earlier.

```
app/  # application code
|--- adapters/  # implementation of the ports defined in the domain
     |--- tests/  # adapter unit tests
|--- entrypoints/  # primary adapters, entry points
     |--- api/  # api entry point
          |--- model/  # api model
          |--- tests/  # end to end api tests
|--- domain/  # domain to implement business logic using hexagonal architecture
     |--- command_handlers/  # handlers used to run commands on the domain
     |--- commands/  # commands on the domain
     |--- events/  # events emitted by the domain
     |--- exceptions/  # exceptions defined on the domain
     |--- model/  # domain model
     |--- ports/  # abstractions used for external communication
     |--- tests/  # domain tests
infra/  # infrastructure code
```

As discussed earlier, the domain is the core of the application and doesn't depend on any other module. We recommend that you structure the `domain` folder to include the following subfolders:

- `command handlers` contains the methods or classes that run commands on the domain.

- `commands` contains the command objects that define the information required to perform an operation on the domain.
- `events` contains the events that are emitted through the domain and then routed to other microservices.
- `exceptions` contains the known errors defined within the domain.
- `model` contains the domain entities, value objects, and domain services.
- `ports` contains the abstractions through which the domain communicates with databases, APIs, or other external components.
- `tests` contains the test methods (such as business logic tests) that are run on the domain.

The primary adapters are the entry points to the application, as represented by the `entrypoints` folder. This example uses the `api` folder as the primary adapter. This folder contains an API `model`, which defines the interface the primary adapter requires to communicate with clients. The `tests` folder contains end-to-end tests for the API. These are shallow tests that validate that the components of the application are integrated and work in harmony.

The secondary adapters, as represented by the `adapters` folder, implement the external integrations required by the domain ports. A database repository is a great example of a secondary adapter. When the database system changes, you can write a new adapter by using the implementation that's defined by the domain. There is no need to change the domain or business logic. The `tests` subfolder contains external integration tests for each adapter.

Infrastructure examples on AWS

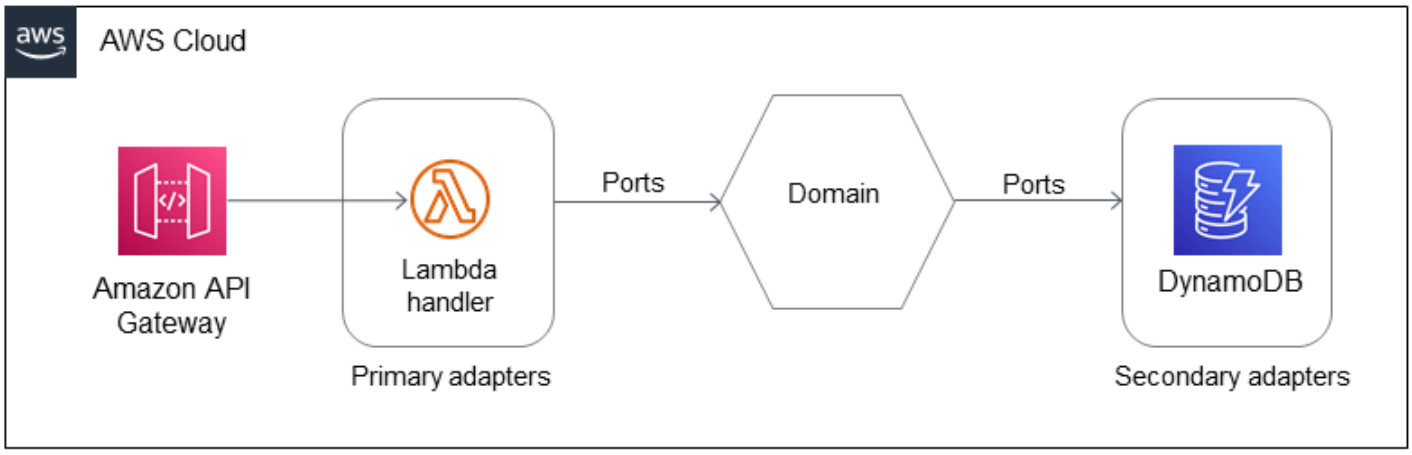
This section provides examples for designing an infrastructure for your application on AWS that you can use to implement a hexagonal architecture. We recommend that you start with a simple architecture for building a minimum viable product (MVP). Most microservices need a single entry point to handle client requests, a compute layer to run the code, and a persistence layer to store data. The following AWS services are great candidates for use as clients, primary adapters, and secondary adapters in hexagonal architecture:

- **Clients:** Amazon API Gateway, Amazon Simple Queue Service (Amazon SQS), Elastic Load Balancing, Amazon EventBridge
- **Primary adapters:** AWS Lambda, Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Service (Amazon EKS), Amazon Elastic Compute Cloud (Amazon EC2)
- **Secondary adapters:** Amazon DynamoDB, Amazon Relational Database Service (Amazon RDS), Amazon Aurora, API Gateway, Amazon SQS, ELB, EventBridge, Amazon Simple Notification Service (Amazon SNS)

The following sections discuss these services in the context of hexagonal architecture in more detail.

Start simple

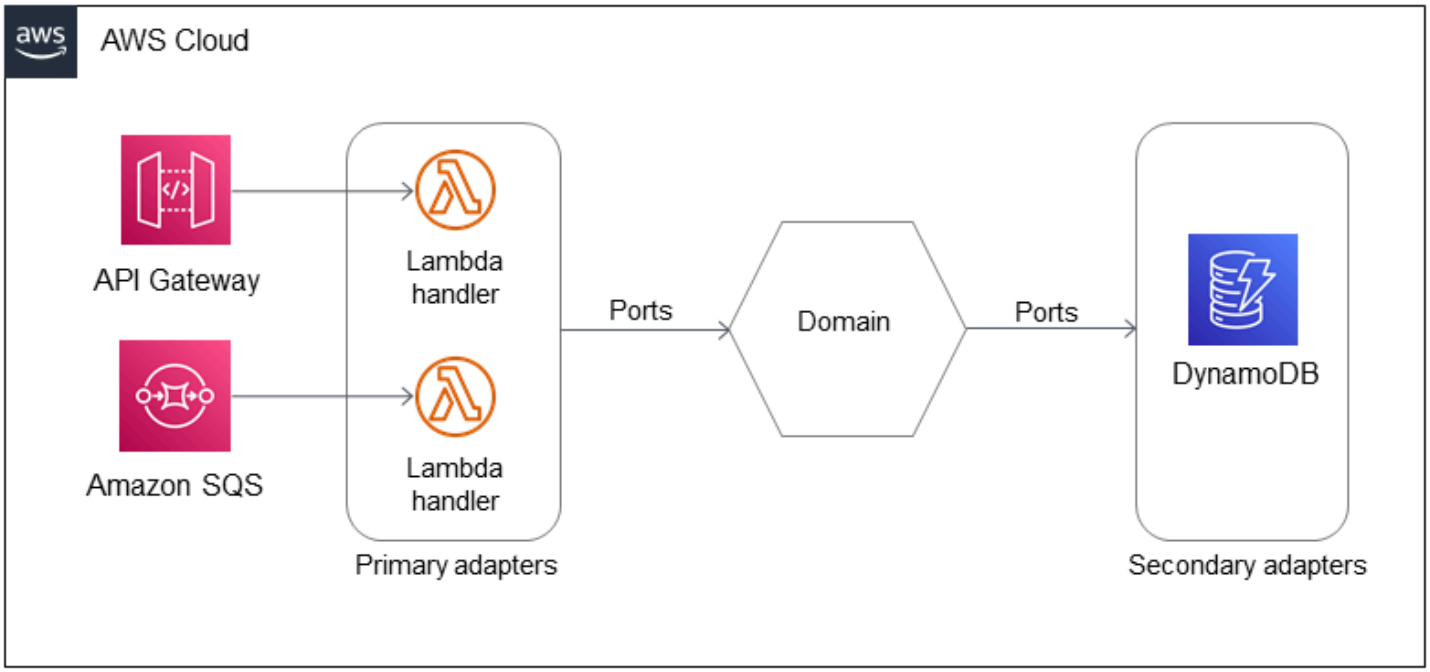
We recommend that you start simple when you architect an application by using hexagonal architecture. In this example, API Gateway is used as the client (REST API), Lambda is used as the primary adapter (compute), and DynamoDB is used as the secondary adapter (persistence). The gateway client calls the entry point, which, in this case, is a Lambda handler.



This architecture is fully serverless and gives the architect a good starting point. We recommend that you use the command pattern in the domain because it makes the code easier to maintain, and it adapts to new business and non-functional requirements. This architecture could be sufficient for building simple microservices with a few operations.

Apply the CQRS pattern

We recommend that you switch to the CQRS pattern if the number of operations on the domain is going to scale. You can apply the CQRS pattern as a fully serverless architecture in AWS by using the following example.

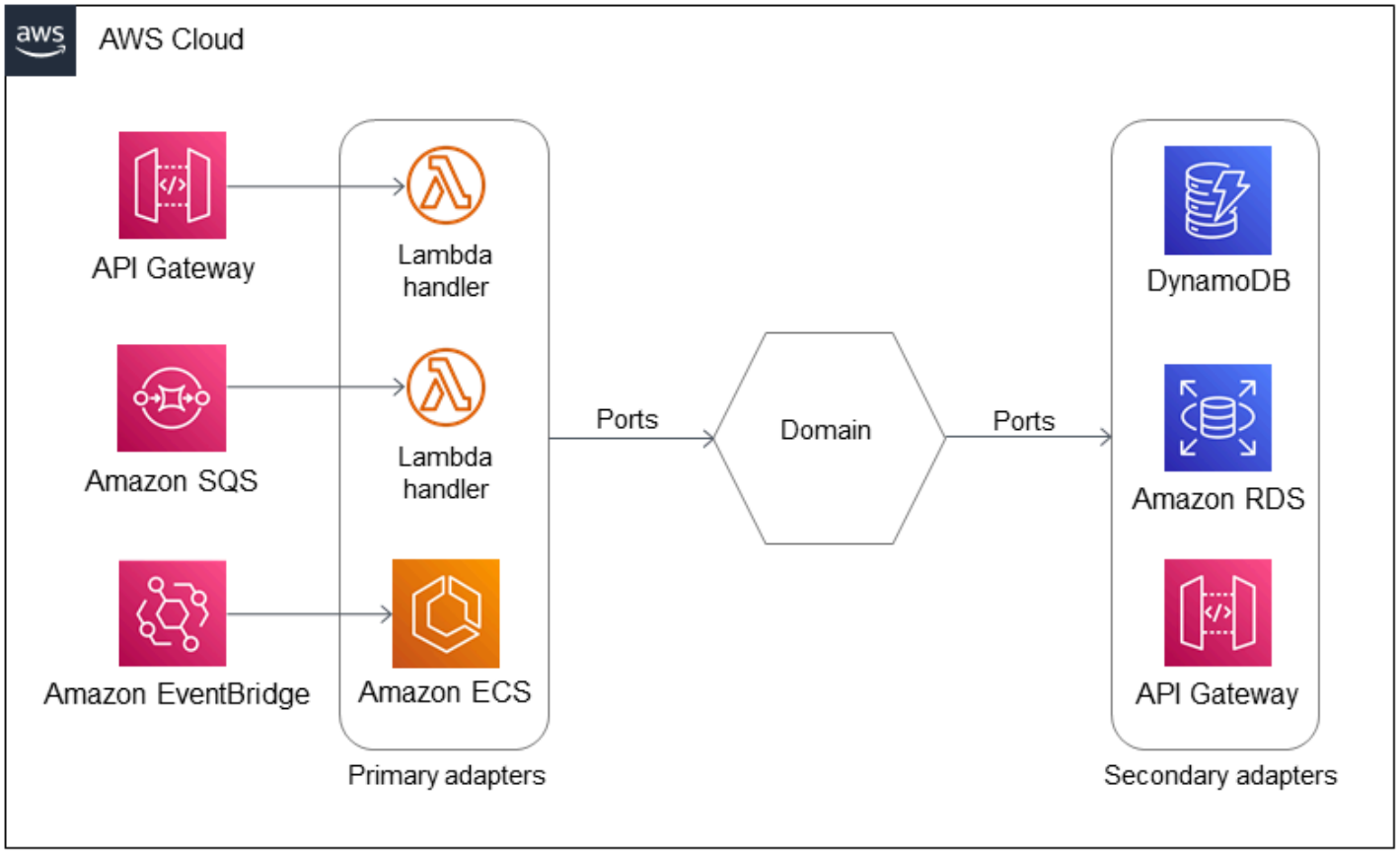


This example uses two Lambda handlers, one for queries and one for commands. Queries are run synchronously by using an API gateway as the client. Commands are run asynchronously by using Amazon SQS as the client.

This architecture includes multiple clients (API Gateway and Amazon SQS) and multiple primary adapters (Lambda), which are called by their corresponding entry points (Lambda handlers). All components belong to the same bounded context, so they are within the same domain.

Evolve the architecture by adding containers, a relational database, and an external API

Containers are a good option for long-running tasks. You might also want to use a relational database if you have a predefined data schema and want to benefit from the power of the SQL language. In addition, the domain would have to communicate with external APIs. You can evolve the architecture example to support these requirements as shown in the following diagram.



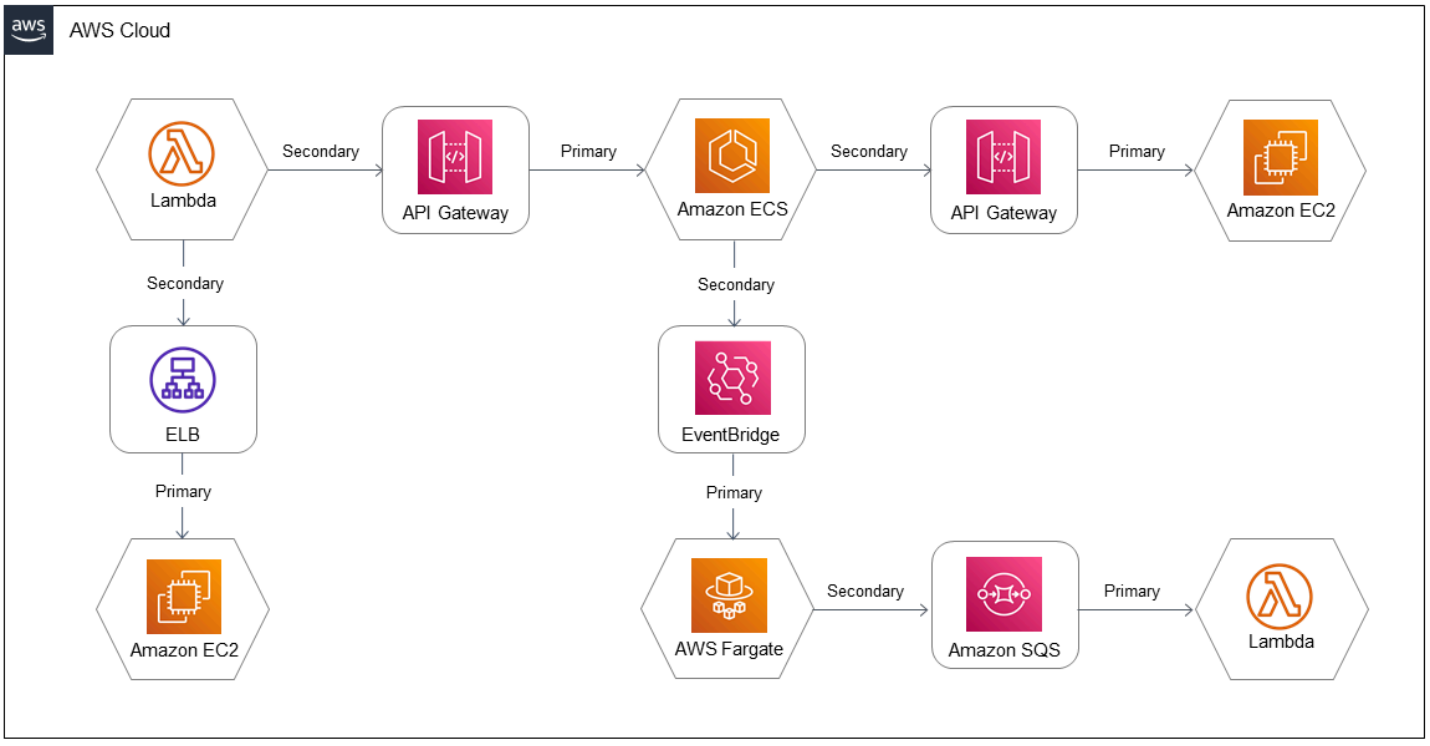
This example uses Amazon ECS as the primary adapter for launching long-running tasks in the domain. Amazon EventBridge (client) initiates an Amazon ECS task (entry point) when a specific

event happens. The architecture includes Amazon RDS as another secondary adapter for storing relational data. It also adds another API gateway as a secondary adapter for invoking an external API call. As a result, the architecture uses multiple primary and secondary adapters that rely on different underlying compute layers in one business domain.

The domain is always loosely coupled with all primary and secondary adapters through abstractions called *ports*. The domain defines what it requires from the outside world by using ports. Because it is the adapter's responsibility to implement the port, switching from one adapter to another doesn't affect the domain. For example, you can switch from Amazon DynamoDB to Amazon RDS by writing a new adapter, without affecting the domain.

Add more domains (zoom out)

Hexagonal architecture aligns well with the principles of a microservices architecture. The architecture examples shown so far contained a single domain (or bounded context). Applications typically include multiple domains, which need to communicate through primary and secondary adapters. Each domain represents a microservice and is loosely coupled with other domains.



In this architecture, each domain uses a different set of compute environment(s). (Each domain might also have multiple compute environments, as in the previous example.) Each domain defines its required interfaces to communicate with other domains through ports. Ports are implemented

by using primary and secondary adapters. This way, the domain is unaffected if there is a change in the adapter. In addition, domains are decoupled from one another.

In the architecture example shown in the previous diagram, Lambda, Amazon EC2, Amazon ECS, and AWS Fargate are used as primary adapters. API Gateway, ELB, EventBridge, and Amazon SQS are used as secondary adapters.

FAQ

Why should I use a hexagonal architecture?

Hexagonal architecture shifts developers' focus to the domain logic, simplifies test automation, and improves code quality and adaptability. These improvements result in a faster time to market and easier technical and organizational scaling.

Why should I use domain-driven design?

Domain-driven design (DDD) enables you to build software components and constructs by using a common language between business stakeholders and engineers. DDD helps you manage software complexity and is an effective strategy for maintaining software products in the long term.

Can I practice test-driven development without hexagonal architecture?

Yes. Test-driven development (TDD) isn't limited to specific software design patterns. However, hexagonal architecture makes it easier to practice TDD.

Can I scale my product without hexagonal architecture and domain-driven design?

Yes. Technical and organizational product scaling can be achieved with most design patterns. However, hexagonal architecture and DDD make it easier to scale and are more effective for large projects in the long term.

Which technologies should I use to implement hexagonal architecture?

Hexagonal architecture isn't limited to a specific technology stack. We recommend that you choose technology that supports dependency inversion and unit testing.

I am developing a minimum viable product. Does it make sense to spend time thinking about software architecture?

Yes. We recommend that you use design patterns that are familiar to you for MVPs. We encourage you to try and practice hexagonal architecture until your engineers are comfortable with it. Establishing a hexagonal architecture for new projects doesn't require a significantly bigger time investment than starting without any architecture.

I am developing a minimum viable product and have no time to write tests.

If your MVP contains business logic, we strongly recommend writing automated tests for it. This will reduce the feedback loop and save time.

Which additional design patterns can I use with hexagonal architecture?

Use the CQRS pattern to support scaling of the overall system. Use the repository pattern to store and restore your domain model. Use the unit of work pattern to manage transactional process steps. Use composition over inheritance to model domain aggregates, entities, and value objects. Do not build complex object hierarchies.

Next steps

- Familiarize yourself further with domain-driven design concepts by reading the links collected in the Resources section.
- If you're implementing a new project, use the project structure template provided in this guide and implement a few features.
- If you're in the process of implementing an existing project, identify code that can be split between read-only and write-only operations. Abstract read-only code into query services, and place write-only code into command handlers.
- When your base project structure is in place, write unit tests, establish continuous integration (CI) with test automation, and follow test-driven development (TDD) practices.

Resources

References

- Structure a Python project in hexagonal architecture using AWS Lambda (AWS Prescriptive Guidance pattern)
- Agile Teams (Scaled Agile Framework website)
- Architecture patterns with Python, by Harry Percival and Bob Gregory (O'Reilly Media, March 31, 2020), specifically the following chapters:
  - Commands and Command Handler
  - Command-Query Responsibility Segregation (CQRS)
  - Repository Pattern
- Event storming: The smartest approach to collaboration beyond silo boundaries, by Alberto Brandolini (Event Storming website)
- Demystifying Conway's Law, by Sam Newman (Thoughtworks website, June 30, 2014)
- Developing evolutionary architecture with AWS Lambda, by James Beswick (AWS Compute Blog, July 8, 2021)
- Domain Language: Tackling Complexity in the Heart of Software (Domain Language website)
- Facade, from *Dive Into Design Patterns* by Alexander Shvets (ebook, December 5, 2018)
- GivenWhenThen, by Martin Fowler (August 21, 2013)
- Implementing Domain-Driven Design, by Vaughn Vernon (Addison-Wesley Professional, February 2013)
- Inverse Conway Maneuver (Thoughtworks website, July 8, 2014)
- Pattern of the Month: Red Green Refactor (DZone website, June 2, 2017)
- SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples, by Thorben Janssen (Stackify website, May 7, 2018)
- SOLID Principles: Explanation and examples, by Simon Hoiberg (ITNEXT website, Jan 1, 2019)
- The Art of Agile Development: Test-Driven Development, by James Shore and Shane Warden (O'Reilly Media, March 25, 2010)
- The SOLID Principles of Object-Oriented Programming Explained in Plain English, by Yigit Kemal Erinc (freeCodeCamp object-oriented programming posts, August 20, 2020)
- What is an Event-Driven Architecture? (AWS website)

AWS services

- Amazon API Gateway
- Amazon Aurora
- Amazon DynamoDB
- Amazon Elastic Compute Cloud (Amazon EC2)
- Amazon Elastic Container Service (Amazon ECS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- ELB
- Amazon EventBridge
- AWS Fargate
- AWS Lambda
- Amazon Relational Database Service (Amazon RDS)
- Amazon Simple Notification Service (Amazon SNS)
- Amazon Simple Queue Service (Amazon SQS)

Other tools

- Moto
- LocalStack

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	June 15, 2022

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or Bitbucket Cloud. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, Amazon SageMaker AI provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

EDI

See electronic data interchange.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

electronic data interchange (EDI)

The automated exchange of business documents between organizations. For more information, see What is Electronic Data Interchange.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more

information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the “2021-05-27 00:15:37” date into “2021”, “May”, “Thu”, and “15”, you can help the learning algorithm learn nuanced patterns associated with different data components.

few-shot prompting

Providing an LLM with a small number of examples that demonstrate the task and desired output before asking it to perform a similar task. This technique is an application of in-context learning, where models learn from examples (*shots*) that are embedded in prompts. Few-shot prompting can be effective for tasks that require specific formatting, reasoning, or domain knowledge. See also zero-shot prompting.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

FM

See foundation model.

foundation model (FM)

A large deep-learning neural network that has been training on massive datasets of generalized and unlabeled data. FMs are capable of performing a wide variety of general tasks, such as understanding language, generating text and images, and conversing in natural language. For more information, see What are Foundation Models.

G

generative AI

A subset of AI models that have been trained on large amounts of data and that can use a simple text prompt to create new content and artifacts, such as images, videos, text, and audio. For more information, see What is Generative AI.

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

golden image

A snapshot of a system or software that is used as a template to deploy new instances of that system or software. For example, in manufacturing, a golden image can be used to provision software on multiple devices and helps improve speed, scalability, and productivity in device manufacturing operations.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries.

Detective guardrails detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub CSPM, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

holdout data

A portion of historical, labeled data that is withheld from a dataset that is used to train a machine learning model. You can use holdout data to evaluate the model performance by comparing the model predictions against the holdout data.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large language model (LLM)

A deep learning AI model that is pretrained on a vast amount of data. An LLM can perform multiple tasks, such as answering questions, summarizing documents, translating text into other languages, and completing sentences. For more information, see What are LLMs.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

LLM

See large language model.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners,

migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic `PUT` and `DELETE` requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

privacy by design

A system engineering approach that takes privacy into account through the whole development process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

prompt chaining

Using the output of one LLM prompt as the input for the next prompt to generate better responses. This technique is used to break down a complex task into subtasks, or to iteratively refine or expand a preliminary response. It helps improve the accuracy and relevance of a model's responses and allows for more granular, personalized results.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

RAG

See Retrieval Augmented Generation.

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

Retrieval Augmented Generation (RAG)

A generative AI technology in which an LLM references an authoritative data source that is outside of its training data sources before generating a response. For example, a RAG model might perform a semantic search of an organization's knowledge base or custom data. For more information, see What is RAG.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata.

The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security by design

A system engineering approach that takes security into account through the whole development process.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your

organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

system prompt

A technique for providing context, instructions, or guidelines to an LLM to direct its behavior. System prompts help set context and establish rules for interactions with users.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zero-shot prompting

Providing an LLM with instructions for performing a task but no examples (*shots*) that can help guide it. The LLM must use its pre-trained knowledge to handle the task. The effectiveness of zero-shot prompting depends on the complexity of the task and the quality of the prompt. See also few-shot prompting.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.