# AWS Prescriptive Guidance

## Using architectural decision records to streamline technical decision-making for a software development project

# AWS Prescriptive Guidance: Using architectural decision records to streamline technical decision-making for a software development project

# Table of Contents

# Using architectural decision records to streamline technical decision-making for a software development project

*Darius Kunce and Dominik Goby, Application Architects, AWS Professional Services*

*March 2022*

This guide introduces the architectural decision records (ADR) process for software engineering projects. ADRs support team alignment, document strategic directions for a project or product, and reduce recurring and time-consuming decision-making efforts.

During project and product development, software engineering teams need to make architectural decisions to reach their goals. These decisions can be technical, such as deciding to use the command query responsibility segregation (CQRS) pattern, or process-related, such as deciding to use the GitFlow workflow to manage source code. Making these decisions is a time-consuming and difficult process. Teams must justify, document, and communicate these decisions to relevant stakeholders.

Three major anti-patterns often emerge when making architectural decisions:

- No decision is made at all, out of fear of making the wrong choice.
- A decision is made without any justification, and people don't understand why it was made. This results in the same topic being discussed multiple times.
- The decision isn't captured in an architectural decision repository, so team members forget or don't know that the decision was made.

These anti-patterns are particularly important to tackle during the development process of a product or project.

Capturing the decision, the context, and considerations that led to the decision in the form of an ADR enables current and future stakeholders to collect information about the decisions made and the thought process behind each decision. This reduces software development time and provides better documentation for future teams.

## Targeted business outcomes

ADRs target three business outcomes:

- They align current and future team members.
- They set a strategic direction for the project or product.
- They avoid decision anti-patterns by defining a process to properly document and communicate architectural decisions.

ADRs capture the context of the decision to inform future stakeholders. A collection of ADRs provide a hand-over experience and reference documentation. Team or project members use the ADR collection for follow-up projects and product feature planning. Being able to reference ADRs reduces the time required during development, reviews, and architectural decisions. ADRs also allow other teams to learn from, and gain insights into, considerations made by other project and product development teams.

# ADR process

An architectural decision record (ADR) is a document that describes a choice the team makes about a significant aspect of the software architecture they're planning to build. Each ADR describes the architectural decision, its context, and its consequences. ADRs have states and therefore follow a lifecycle. For an example of an ADR, see the appendix (p. 10).

The ADR process outputs a collection of architectural decision records. This collection creates the decision log. The decision log provides the project context as well as detailed implementation and design information. Project members skim the headlines of each ADR to get an overview of the project context. They read the ADRs to dive deep into project implementations and design choices.

When the team accepts an ADR, it becomes immutable. If new insights require a different decision, the team proposes a new ADR. When the team accepts the new ADR, it supersedes the previous ADR.

## Scope of the ADR process

Project members should create an ADR for every architecturally significant decision that affects the software project or product, including the following (Richards and Ford 2020 (p. 9)):

- Structure (for example, patterns such as microservices)
- Non-functional requirements (security, high availability, and fault tolerance)
- Dependencies (coupling of components)
- Interfaces (APIs and published contracts)
- Construction techniques (libraries, frameworks, tools, and processes)

Functional and non-functional requirements are the most common inputs to the ADR process.

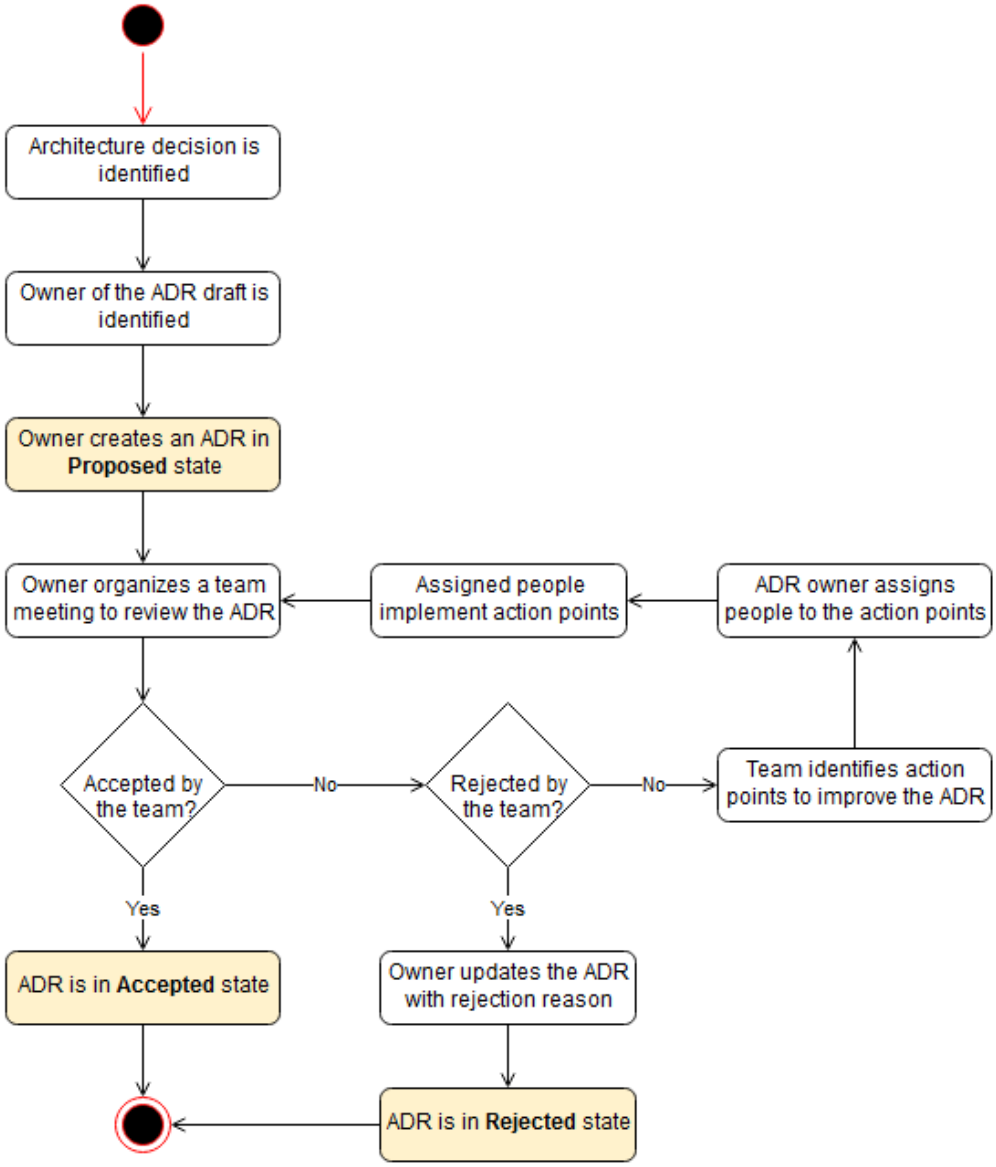
## ADR contents

When the team identifies a need for an ADR, a team member starts to write the ADR based on a projectwide template. (See the ADR GitHub organization for example templates.) The template simplifies ADR creation and ensures that the ADR captures all the relevant information. At a minimum, each ADR should define the context of the decision, the decision itself, and the consequences of the decision for the project and its deliverables. (For examples of these sections, see the appendix (p. 10).) One of the most powerful aspects of the ADR structure is that it focuses on the reason for the decision rather than how the team implemented it. Understanding why the team made the decision makes it easier for other team members to adopt the decision, and prevents other architects who weren't involved in the decision-making process to overrule that decision in the future.

## ADR adoption process

Every team member can create an ADR, but the team should establish a definition of ownership for an ADR. Each author who is the owner of an ADR should actively maintain and communicate the ADR content. To clarify this ownership, this guide refers to ADR authors as *ADR owners* in the following sections. Other team members can always contribute to an ADR. If the content of an ADR changes before the team accepts the ADR, the owner should approve these changes.

The following diagram illustrates the ADR creation, ownership, and adoption process.




After the team identifies an architectural decision and its owner, the ADR owner provides the ADR in the **Proposed** state at the beginning of the process. ADRs in the **Proposed** state are ready for review.

The ADR owner then initiates the review process for the ADR. The goal of the ADR review process is to decide whether the team accepts the ADR, determines that it needs rework, or rejects the ADR. The project team, including the owner, reviews the ADR. The review meeting should start with a dedicated time slot to read the ADR. On average, 10 to 15 minutes should be enough. During this time, each team member reads the document and adds comments and questions to flag unclear topics. After the review phase, the ADR owner reads out and discusses each comment with the team.

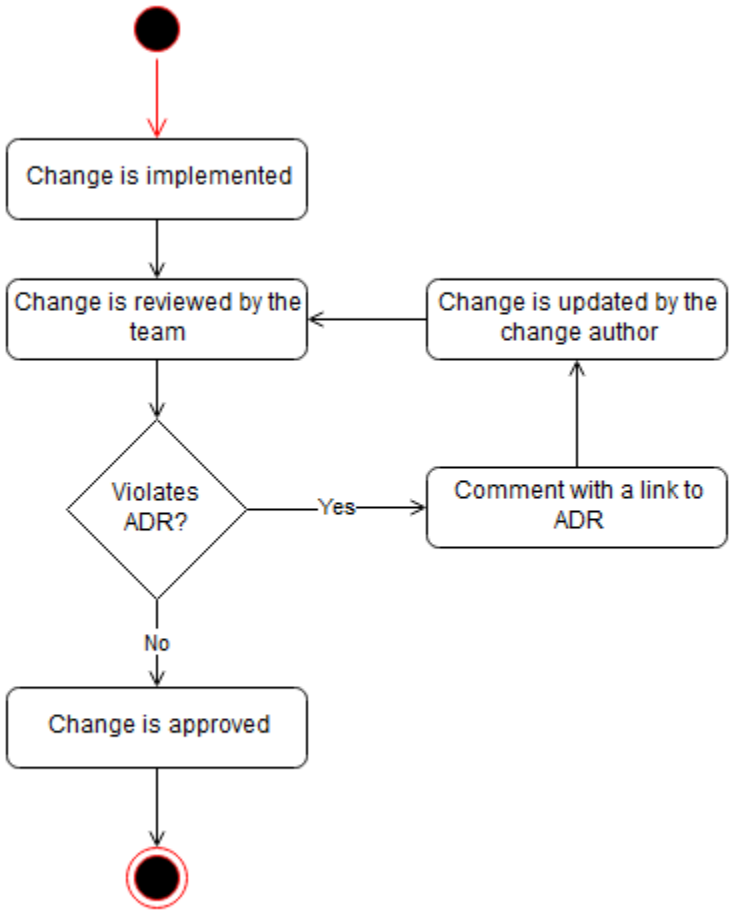
If the team finds action points to improve the ADR, the state of the ADR stays **Proposed**. The ADR owner formulates the actions, and, in collaboration with the team, adds an assignee to each action. Each team member can contribute and resolve the action points. It is the responsibility of the ADR owner to reschedule the review process.

The team can also decide to reject the ADR. In this case, the ADR owner adds a reason for the rejection to prevent future discussions on the same topic. The owner changes the ADR state to **Rejected**.

If the team approves the ADR, the owner adds a timestamp, version, and list of stakeholders. The owner then updates the state to **Accepted**.

ADRs and the decision log they create represent decisions made by the team and provide a history of all decisions. The team uses the ADRs as a reference during code and architectural reviews where possible. In addition to performing code reviews, design tasks, and implementation tasks, team members should consult ADRs for strategic decisions for the product.

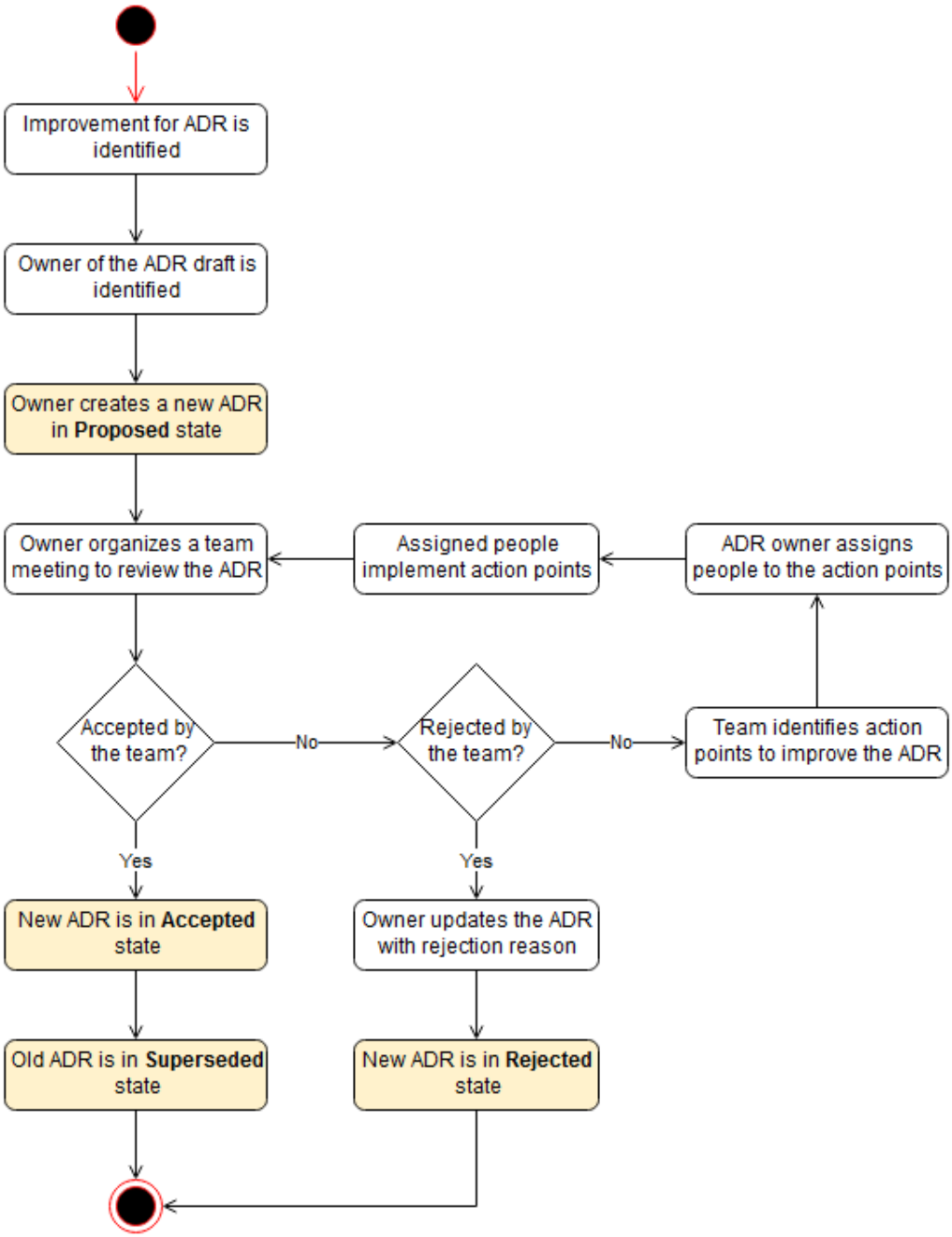
The following diagram shows the process of applying an ADR to validate if a change in a software component conforms to the agreed decisions.




As a good practice, each software change should go through peer reviews and require at least one approval. During the code review, a code reviewer might find changes that violate one or more ADRs. In this case, the reviewer asks the author of the code change to update the code, and shares a link to the ADR. When the author updates the code, it is approved by peer reviewers and merged into the main code base.

# ADR review process

The team should treat ADRs as immutable documents after the team accepts or rejects them. Changes to an existing ADR requires creating a new ADR, establishing a review process for the new ADR, and approving the ADR. If the team approves the new ADR, the owner should change the state of the old ADR to **Superseded**. The following diagram illustrates the update process.



# Best practices

**Promote ownership.** Each project team member should be empowered to create and own an ADR. This practice distributes architectural research work among team members and offloads that work from the solutions architect or team lead. It also fosters a sense of ownership in the decision-making process. This helps the team adopt those decisions faster instead of treating them as decisions that were imposed from higher levels of the organization.

**Preserve ADR history.** ADRs should have a change history, and each change should have an owner. When the ADR owner updates the ADR, they should change the status of the old ADR to **Superseded**, note their changes in the change history of the new ADR, and keep the old ADR in the decision log.

**Schedule regular review meetings.** If you are on a new (greenfield) project, the ADR process can be quite intense in the beginning. We recommend that you establish a cadence of regular ADR discussion and review meetings before or after the daily standup. With this approach, the defined ADRs will stabilize in two or three sprints, and you can build a solid foundation with fewer meetings.

**Store ADRs in a central location.** Each project member should have access to the collection of ADRs. We recommend that you store the ADRs in a central location and reference them on the main page of your project documentation. There are two popular options for storing ADRs:

- A Git repository, which makes it easier to version ADRs
- A wiki page, which makes the ADRs accessible to all team members

**Address non-compliant code.** The ADR process doesn't solve the issue of non-compliant legacy code. If you have legacy code that doesn't support the established ADRs, you can either update the outdated code base or artifacts gradually, while introducing new changes, or your team can decide to refactor the code explicitly by creating technical debt tasks.

# FAQ

## What are the benefits of creating an ADR process?

The project team should create an ADR process to streamline architectural decision-making, prevent repeated discussions about the same architectural topics, and communicate architectural decisions effectively.

## When should the project team create an ADR?

The project team should create an ADR for every aspect of the software that affects structure (patterns such as microservices), non-functional requirements (security, high availability, and fault tolerance), dependencies (coupling of components), interfaces (APIs and published contracts), and construction techniques (libraries, frameworks, tools, and processes).

## How often should the project team review an ADR?

The project team should review the ADR at least once before accepting it.

## Who should create an ADR?

Every team member can create an ADR. We recommend that you promote a notion of ownership for ADRs. An author who owns the ADR should actively maintain and communicate the ADR content. Other team members can always contribute to an ADR. The ADR owner should approve changes to an ADR.

## What information should an ADR contain?

At a minimum, each ADR has to define the context of the decision, the decision itself, and the consequences of the decision for the project and its deliverables. The context should mention possible solutions the team considered. It should also contain any relevant information relating to the project, customer, or technology stack. The decision must clearly state, in imperative language, the solution the team has decided to adopt. Avoid using words such as “should,” and phrase each decision to say “We use…” or “The team has to use…” The consequences section should mention all known trade-offs of making the decision. Each ADR must have a status and a changelog that contains the change date and the person who is responsible for the change.

## Where can I find ADR templates?

There are multiple versions and variants of ADR templates available. For a public collection of commonly used ADR templates, see the ADR GitHub repository.

# Next steps and resources

We recommend that you start small and see the benefit that ADRs bring to your team. If you are working on an ongoing project, identify the next architectural change and apply the proposed ADR process to create your first ADR.

Another starting point is to document your overall software development process by using ADRs. Often, the development process is based on tacit knowledge that the team didn't capture in any documentation. Documenting this process enables a smoother experience for new members of the team.

If you are on a greenfield project, apply the ADR process and start capturing all the decisions from the beginning in a few sentences. You can then iterate on those ADRs and supplement them with new information. After you establish your ADRs, you can start using them as a reference in your code review process.

**Resources**

- Architecture Decision Records. https://adr.github.io/.
- Richards, Mark and Neal Ford. 2020. *Fundamentals of Software Architecture*. Sebastopol: O'Reilly Media.

# Appendix: Example ADR

**Title**

This decision defines the software development lifecycle approach for ABC application development.

**Status**

Accepted

**Date**

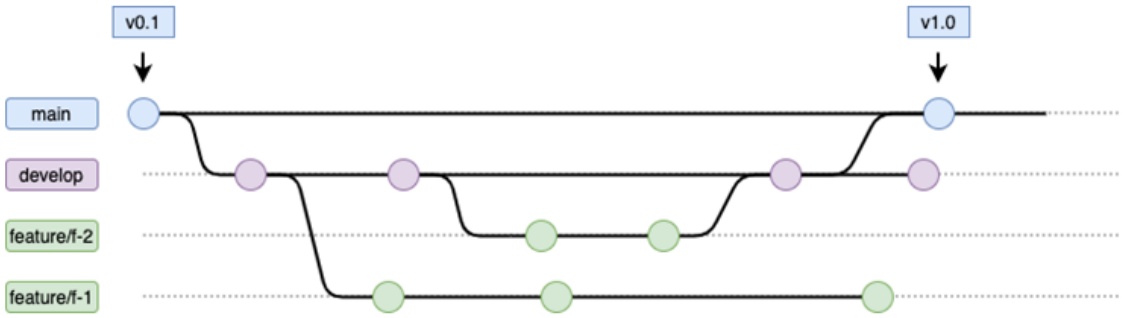
2022-03-11

**Context**

ABC application is a packaged solution, which will be deployed to the customer's environment by using a deployment package. We need to have a development process that will enable us to have a controllable feature, hotfix, and release pipeline.

**Decision**

We use an adapted version of the GitFlow workflow to develop ABC application.




For simplicity, we will not be using the `hotfix/*` and `release/*` branches, because ABC application will be packaged instead of being deployed to a specific environment. For this reason, there is no need for additional complexity that might prevent us from reacting quickly to fix bugs in production releases, or testing releases in a separate environment.

The following is the agreed branching strategy:

- Each repository must have a protected `main` branch that will be used to tag releases.
- Each repository must have a protected `develop` branch for all ongoing development work.

**Consequences**

Positive:

- Adapted GitFlow process will enable us to control release versioning of the ABC application.

Negative:

- GitFlow is more complicated than trunk-based development or GitHub flow and has more overhead.

**Compliance**

- The `main` and `develop` branches in each repository must be marked as `Protected`.
- Changes to the `main` and `develop` branches must be propagated by using merge requests.
- At least one approval is required for every merge request.

**Notes**

- Author: Jane Doe
- Version: 0.1
- Changelog:
  - 0.1: Initial proposed version

# AWS Prescriptive Guidance glossary

AI and ML terms (p. 12) | Migration terms (p. 13) | Modernization terms (p. 17)

## AI and ML terms

The following are commonly used terms in artificial intelligence (AI) and machine learning (ML)-related strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

| | |
|---|---|
| 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?" |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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?" |
| 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). |
| 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. |
| 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. |
| 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. |
| 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. |

# Migration terms

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

| | |
|---|---|
| 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:<br><br>• 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.<br>• 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.<br>• 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.<br>• Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. This migration scenario is specific to VMware Cloud on AWS, which supports virtual machine (VM) compatibility and workload portability between your on-premises environment and AWS. You can use the VMware Cloud Foundation technologies from your on-premises data centers when you migrate your infrastructure to VMware Cloud on AWS. Example: Relocate the hypervisor hosting your Oracle database to VMware Cloud on AWS.<br>• 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.<br>• Retire – Decommission or remove applications that are no longer needed in your source environment. |
| 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 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. |
| 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 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. |
| 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. |
| 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. |
| 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 stages of adoption | The four phases that organizations typically go through when they migrate to the AWS Cloud:<br>• Project – Running a few cloud-related projects for proof of concept and learning purposes<br>• Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)<br>• Migration – Migrating individual applications<br>• Re-invention – Optimizing products and services, and innovating in the cloud<br><br>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. |
| configuration management database (CMDB) | A database that contains information about a company's hardware and software products, configurations, and inter-dependencies. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| large migration | A migration of 300 or more servers. |
| 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 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 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 CloudEndure 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 strategy | The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs (p. 13) entry in this glossary and see Mobilize your organization to accelerate large-scale migrations. |
| 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). |
| 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. |

| | |
|---|---|
| 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. |
| 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. |
| 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. |
| responsible, accountable, consulted, informed (RACI) matrix | A matrix that defines and assigns roles and responsibilities in a project. For example, you can create a RACI to define security control ownership or to identify roles and responsibilities for specific tasks in a migration project. |
| 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. |
| service-level agreement (SLA) | An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance. |
| 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. |
| 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. |
| 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. |

# Modernization terms

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

| | |
|---|---|
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. For more information, see the Two-pizza team section of the Introduction to DevOps on AWS whitepaper. |

# 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.

| update-history-change | update-history-description | update-history-date |
| --- | --- | --- |
| Initial publication (p. 20) | — | March 16, 2022 |