AWS Prescriptive Guidance
Decomposing monoliths into microservices
## Table of Contents

- **Introduction** ...................................................................................................................................... 1  
  - Targeted business outcomes ........................................................................................................ 2
- **Patterns for decomposing monoliths** ........................................................................................................ 3  
  - Decompose by business capability ................................................................................................. 3  
  - Decompose by subdomain .................................................................................................................. 4  
  - Decompose by transactions .............................................................................................................. 5  
  - Service per team pattern ................................................................................................................... 7
- **FAQ** .................................................................................................................................................. 9  
  - Can you use multiple patterns to break down one monolith? ........................................................... 9  
  - How does decomposing a monolith into microservices affect the DevOps process? ......................... 9
- **Resources** ........................................................................................................................................ 10  
  - Related guides ................................................................................................................................. 10  
  - Other resources ............................................................................................................................... 10
- **AWS Prescriptive Guidance glossary** .................................................................................................... 11
- **Document history** .............................................................................................................................. 19
Decomposing monoliths into microservices

Hari Ohm Prasath Rajagopal, Senior Modernization Architect, Amazon Web Services
Tabby Ward, Senior Modernization Architect, Amazon Web Services
Dmitry Gulin, Modernization Architect, Amazon Web Services

January 2021

A migration to the Amazon Web Services (AWS) Cloud has many advantages, including technical and business agility, new revenue opportunities, and reduced costs. To fully benefit from these advantages, you should continuously modernize your organization’s software by refactoring your monolithic applications into microservices. This process consists of three main steps:

- Decompose monoliths into microservices – Use the decomposition patterns provided by this guide to break down monolithic applications into microservices.
- Integrate microservices – Integrate the newly created microservices into a microservices architecture by using AWS serverless services.
- Enable data persistence for microservices – Promote polyglot persistence among your microservices by decentralizing data stores.

One of the first steps in your application modernization journey is to decompose the monoliths in your portfolio into microservices. Most applications begin as monoliths that represent a specific business use case, and if the monolith's architecture doesn't enforce modular design, a monolith can remain a valid choice for applications that don't have well-established domain knowledge. The central characteristic of a monolith as a single unit of deployment can also help mitigate design flaws, such as tight coupling or a lack of internal structure.

Although a monolith can be a valid option for some use cases, it is typically not suitable for a modern application. The poorly defined internal structures of a monolith can make it difficult to maintain code, which creates a steep learning curve for new developers and causes additional support costs. High coupling and low cohesion can significantly increase the time it takes to add new features, and you might be unable to scale individual components based on traffic patterns. Monoliths also require multiple teams to coordinate for one large release, which increases the collaboration and knowledge transfer burden. Finally, you can find that adding new features or building new user experiences becomes difficult when your business grows or user numbers increase.

To avoid this, you can use decomposition patterns to break down monolithic applications, convert them into several microservices, and migrate them to a microservices architecture. Before you begin the decomposition process, you should evaluate which monoliths to decompose, and make sure to include those with reliability or performance issues, or those that include multiple components in a tightly coupled architecture. We also recommend that you fully understand the monolith’s business use case, technology, and its interdependencies with other applications.

This guide is for application owners, business owners, architects, technical leads, and project managers. It discusses the following four cloud-native patterns that are used to decompose monoliths, and describes the advantages and disadvantages of each one:

- Decompose by business capability (p. 3)
- Decompose by subdomain (p. 4)
• Decompose by transactions (p. 5)
• Service per team pattern (p. 7)

The guide is part of a content series that covers the application modernization approach recommended by AWS. The series also includes:

• Strategy for modernizing applications in the AWS Cloud
• Phased approach to modernizing applications in the AWS Cloud
• Evaluating modernization readiness for applications in the AWS Cloud
• Integrating microservices by using AWS serverless services
• Enabling data persistence for microservices

Targeted business outcomes

You should expect the following outcomes after you decompose your monoliths into microservices:

• An efficient transition of your monolithic application into a microservices architecture.
• Rapid adjustments to fluctuating business demand but without interrupting core activities, such as high scalability, improved resiliency, continuous delivery, and failure isolation.
• Faster innovation because each microservice can be individually tested and deployed.
Patterns for decomposing monoliths

After you decide to decompose a monolith in your application portfolio, you should choose the appropriate decomposition patterns and introduce them into your organization.

**Note**
You can use multiple patterns to decompose a monolith. For example, you can decompose a monolith by business capability (p. 3) and then use the subdomain pattern (p. 4) to break it down more.

**Topics**
- Decompose by business capability (p. 3)
- Decompose by subdomain (p. 4)
- Decompose by transactions (p. 5)
- Service per team pattern (p. 7)

**Decompose by business capability**

A monolith can be decomposed by using your organization's business capabilities. A **business capability** is what a business does to generate value (for example, sales, customer service, or marketing). Typically, an organization has multiple business capabilities and these vary by sector or industry. Use this pattern if your team has enough insight into your organization’s business units and you have subject matter experts (SMEs) for each business unit. The following table provides the advantages and disadvantages of using this pattern.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generates a stable microservices architecture if the business capabilities are relatively stable.</td>
<td>Application design is tightly coupled with the business model.</td>
</tr>
<tr>
<td>Development teams are cross-functional and organized around delivering business value, instead of technical features.</td>
<td>Requires an in-depth understanding of the overall business, because it can be difficult to identify business capabilities and services.</td>
</tr>
<tr>
<td>Services are loosely coupled.</td>
<td></td>
</tr>
</tbody>
</table>

In the following diagram, an insurance monolith is decomposed into four different microservices based on the business capabilities.
Decompose by subdomain

This pattern uses a domain-driven design (DDD) subdomain to decompose monoliths. This approach breaks down the organization's domain model into separate subdomains that are labeled as core (a key differentiator for the business), supporting (possibly related to business but not a differentiator), or generic (not business-specific). This pattern is appropriate for existing monolithic systems that have well-defined boundaries between subdomain-related modules. This means you can decompose the monolith by repackaging existing modules as microservices but without significantly rewriting existing code. Each subdomain has a model and the scope of that model is called a bounded context; microservices are developed around this bounded context.

The following table provides the advantages and disadvantages of using this pattern.
Decompose by transactions

In a distributed system, an application typically needs to call multiple microservices to complete one business transaction. To avoid latency issues or two-phase commit problems, you can group your microservices based on transactions. This pattern is appropriate if you consider response times important and your different modules do not create a monolith after you package them. The following table provides the advantages and disadvantages of using this pattern.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Loosely coupled architecture provides scalability, resilience, maintainability, extensibility, location transparency, protocol independence, and time independence.</td>
<td>• Can create too many microservices, which makes service discovery and integration difficult.</td>
</tr>
<tr>
<td>• Systems become more scalable and predictable.</td>
<td>• Business subdomains are difficult to identify because they require an in-depth understanding of the overall business.</td>
</tr>
</tbody>
</table>
Advantages | Disadvantages
---|---
• Faster response times.  
• You don’t need to worry about data consistency.  
• Improved availability.  

• Multiple modules can be packaged together, and this can create a monolith.  
• Increased cost and complexity due to multiple functionalities being implemented in a microservice, instead of as a separate microservice.  
• Transaction-oriented microservices can grow if the number of business domains and dependencies among them is high.  
• Inconsistent versions might be deployed at the same time for the same business domain.

In the following illustration, the insurance monolith is broken down into multiple microservices based on transactions.

In an insurance system, a claim request is typically tagged to a customer after it is submitted. This means that a claims service cannot exist without a “Customers” microservice. “Sales” and “Customers” are packaged together in one microservice package, and a business transaction requires coordinating with both of them.
Service per team pattern

Instead of decomposing monoliths by business capabilities or services, the service per team pattern breaks them down into microservices that are managed by individual teams. Each team is responsible for a business capability and owns the capability's code base. The team independently develops, tests, deploys, or scales its services, and primarily interacts with other teams to negotiate APIs. We recommend that each individual microservice is owned by only one team. However, if the team is large enough, it’s possible that multiple subteams could own separate microservices within the same team structure. The following table provides the advantages and disadvantages of using this pattern.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Teams act independently with minimal coordination.</td>
<td>• It can be difficult to align teams to end-user functionality or business capabilities.</td>
</tr>
<tr>
<td>• Code bases and microservices are not shared by multiple teams.</td>
<td>• Additional effort is required to deliver larger, coordinated application increments, especially if there are circular dependencies between teams.</td>
</tr>
<tr>
<td>• Teams can quickly innovate and iterate product features.</td>
<td></td>
</tr>
<tr>
<td>• Different teams can use different technologies, frameworks, or</td>
<td></td>
</tr>
<tr>
<td>programming languages.</td>
<td></td>
</tr>
<tr>
<td><strong>Important</strong>: These should be hidden behind a well-defined and stable public API.</td>
<td></td>
</tr>
</tbody>
</table>

The following illustration shows how a monolith is split into microservices that are managed, maintained, and delivered by individual teams.
AWS Prescriptive Guidance Decomposing monoliths into microservices
Service per team pattern
Decomposing monoliths FAQ

This section provides answers to commonly raised questions about decomposing monoliths.

Can you use multiple patterns to break down one monolith?

Yes, you can use multiple patterns to decompose a monolith. The most common way is to decompose a monolith with the Decompose by business capability (p. 3) pattern and then use the Decompose by subdomain (p. 4) pattern to break it down more.

How does decomposing a monolith into microservices affect the DevOps process?

Because you do not have to redeploy everything after a change is made to the application, you must have support and ownership of newly created microservices that are added to the deployment process. This could make the DevOps process more complex.
Resources

Related guides

- Strategy for modernizing applications in the AWS Cloud
- Phased approach to modernizing applications in the AWS Cloud
- Evaluating modernization readiness for applications in the AWS Cloud
- Integrating microservices by using AWS serverless services
- Enabling data persistence in microservices

Other resources

- Break down a monolithic application into microservices with Amazon ECS, Docker, and Amazon EC2
AWS Prescriptive Guidance glossary

AI and ML terms (p. 11) | Migration terms (p. 12) | Modernization terms (p. 16)

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:

- 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. 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.

• 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.

**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
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS Prescriptive Guidance Decomposing monoliths into microservices</td>
<td>Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.</td>
</tr>
<tr>
<td>business continuity planning (BCP)</td>
<td>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.</td>
</tr>
<tr>
<td>Cloud Center of Excellence (CCoE)</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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. |
| 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](#) 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
Decomposing monoliths into microservices

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.

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial publication (p. 19)</td>
<td>—</td>
<td>January 11, 2021</td>
</tr>
</tbody>
</table>