AWS Prescriptive Guidance

Decomposing monoliths into microservices

AWS Prescriptive Guidance: Decomposing monoliths into microservices

Copyright © 2023 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

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
- Document history 11
- Glossary 12
  - Modernization terms 12

Decomposing monoliths into microservices

Hari Ohm Prasath Rajagopal, Tabby Ward, and Dmitry Gulin, Amazon Web Services (AWS)

January 2021 (document history (p. 11))

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

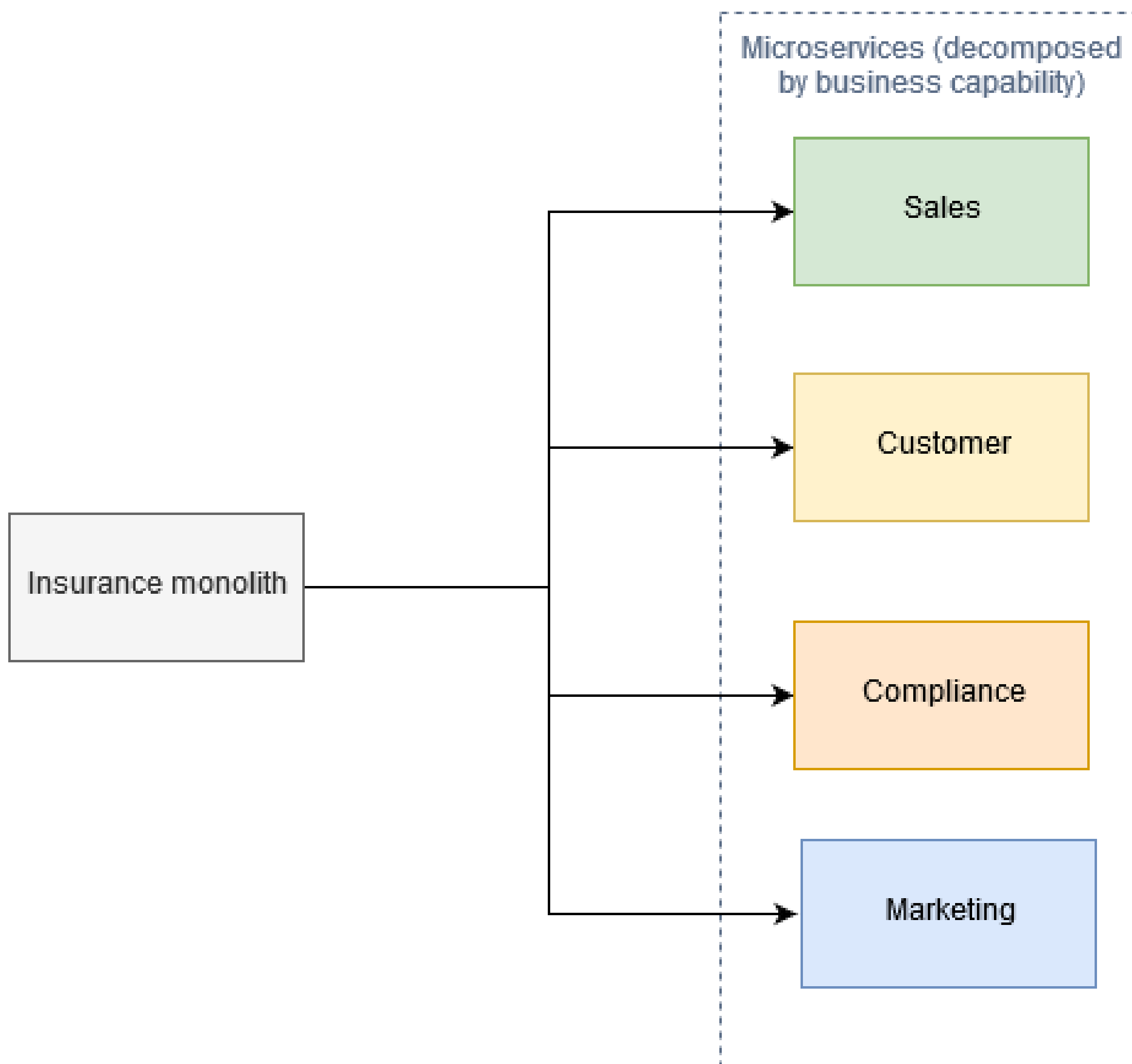
- Decompose by business capability (p. 3)
- Decompose by subdomain (p. 4)
- Decompose by transactions (p. 5)
- Service per team pattern (p. 7)

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.

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

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



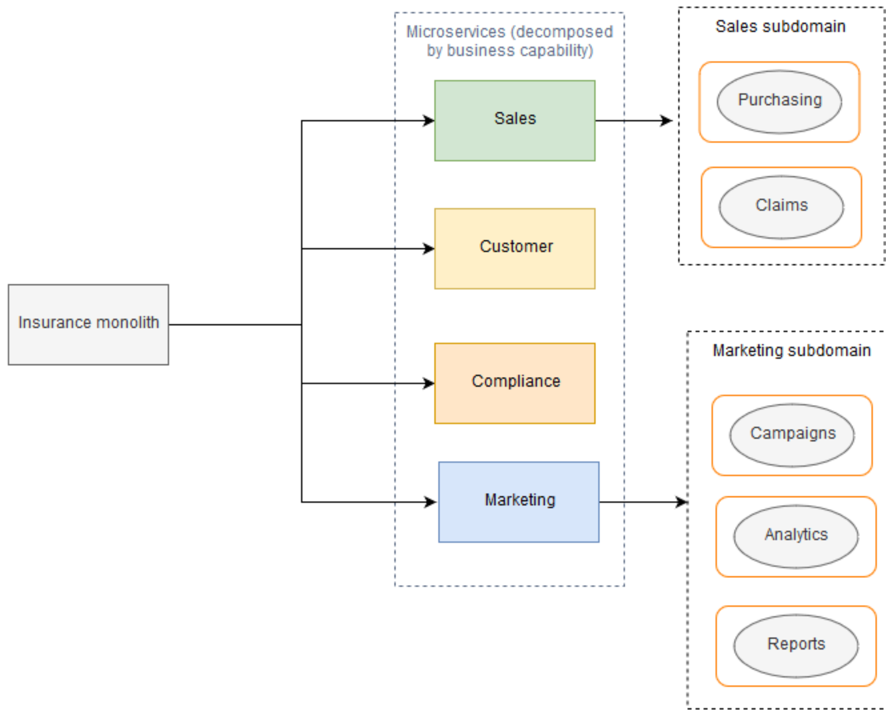
Decompose by subdomain

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.

Advantages	Disadvantages
• Loosely coupled architecture provides scalability, resilience, maintainability, extensibility, location transparency, protocol independence, and time independence. • Systems become more scalable and predictable.	• Can create too many microservices, which makes service discovery and integration difficult. • Business subdomains are difficult to identify because they require an in-depth understanding of the overall business.

The following illustration shows how an insurance monolith is decomposed into subdomains after it was decomposed by business capabilities.



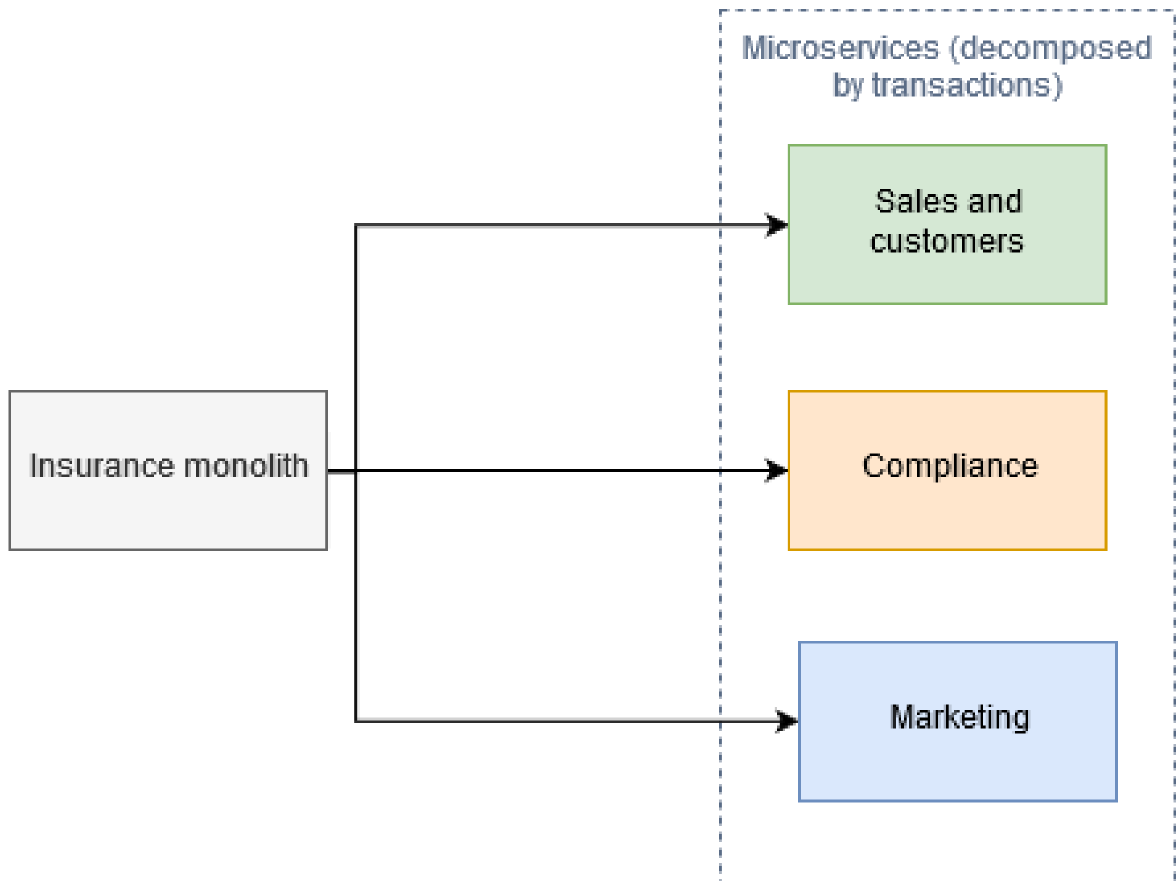
The illustration shows that the “Sales” and “Marketing” services are broken down into smaller microservices. The "Purchasing" and "Claims” models are important business differentiators for "Sales," and are split into two separate microservices. "Marketing" is decomposed by using supporting business functionalities, such as "Campaigns," "Analytics," and "Reports."

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.

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.



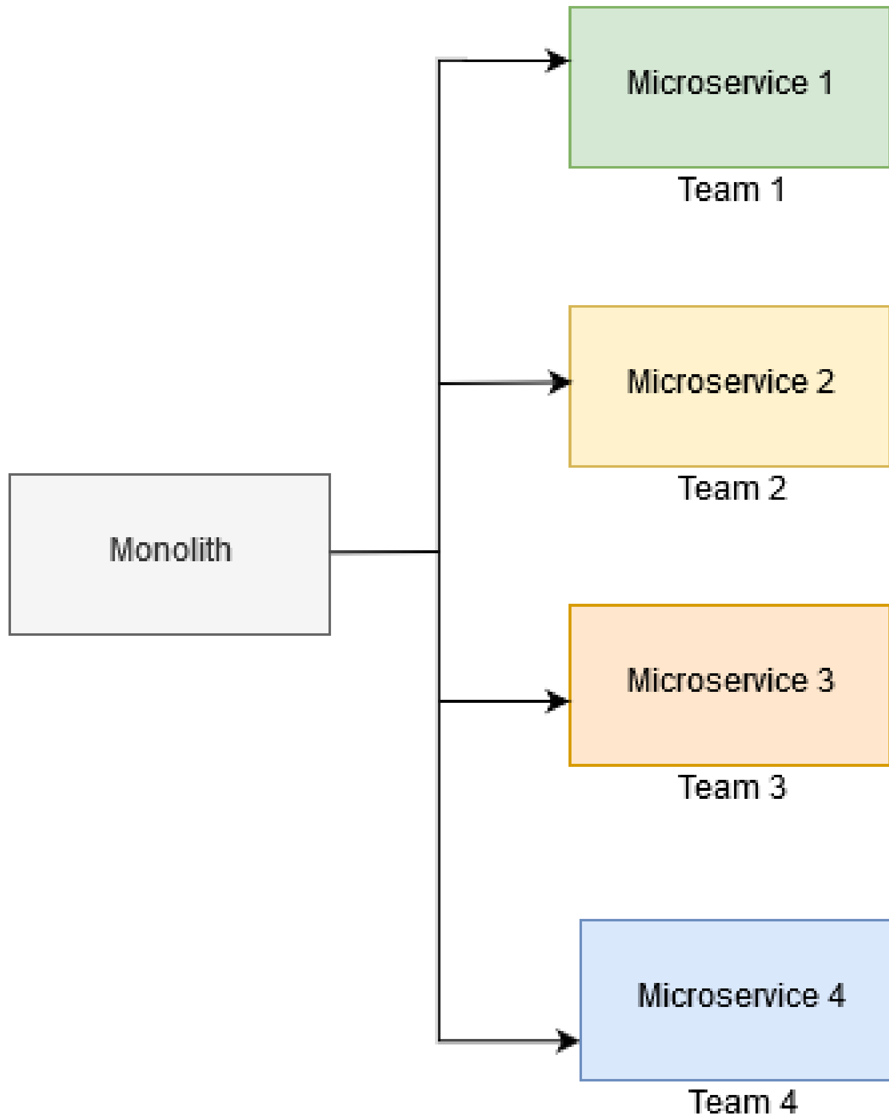
In an insurance system, a claim request is typically tagged to a customer after it is submitted. This means that a claims service cannot exist without a “Customers” microservice. "Sales" and "Customers" are packaged together in one microservice package, and a business transaction requires coordinating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

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

The following illustration shows how a monolith is split into microservices that are managed, maintained, and delivered by individual teams.


Microservice 1
Team 1
Microservice 2
Team 2
Monolith
Microservice 3
Team 3
Microservice 4
Team 4

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

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 (p. 11)	—	January 11, 2021

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.

Modernization terms

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.