Decomposing monoliths into microservices

AWS Prescriptive Guidance

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Prescriptive Guidance: Decomposing monoliths into microservices

Copyright © 2026 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 3
Patterns for decomposing monoliths **4**
Decompose by business capability 4
Decompose by subdomain 6
Decompose by transactions 8
Service per team pattern 10
Strangler fig pattern 12
Branch by abstraction pattern 14
FAQ **18**
Can you use multiple patterns to break down one monolith? 18
How does decomposing a monolith into microservices affect the DevOps process? 18
Resources **19**
Related guides 19
Other resources 19
Document history **20**

Decomposing monoliths into microservices

Tabby Ward and Dmitry Gulin, Amazon Web Services (AWS)

April 2023 (document history)

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.

Modernization typically involves two types of projects:

- *Brownfield projects* involve developing and deploying a new software system within the context of existing or legacy systems.
- *Greenfield projects* involve creating a system from scratch for a completely new environment, without any legacy code involved.

For brownfield projects, 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 are designed for a specific business use case. If the monolith's architecture doesn't enforce modular design, a monolith can remain a valid choice for applications that don't have clearly defined responsibilities within the boundaries of 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 or your user base grows.

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. A microservices architecture structures an application as a series of loosely coupled services. Microservices are designed to accelerate software development by enabling continuous delivery and continuous deployment (CI/CD) processes.

Before you begin the decomposition process, you should evaluate which monoliths to decompose. Make sure to include monoliths that have reliability or performance issues, or include multiple components in a tightly coupled architecture. We also recommend that you fully understand the business use case for the monolith, its technology, and its inter-dependencies with other applications.

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

- Decompose by business capability
- Decompose by subdomain
- Decompose by transactions
- Service per team pattern
- Strangler fig pattern
- Branch by abstraction pattern

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

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 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 and then use the subdomain pattern to break it down more.

Topics

- Decompose by business capability
- Decompose by subdomain
- Decompose by transactions
- Service per team pattern
- Strangler fig pattern
- Branch by abstraction pattern

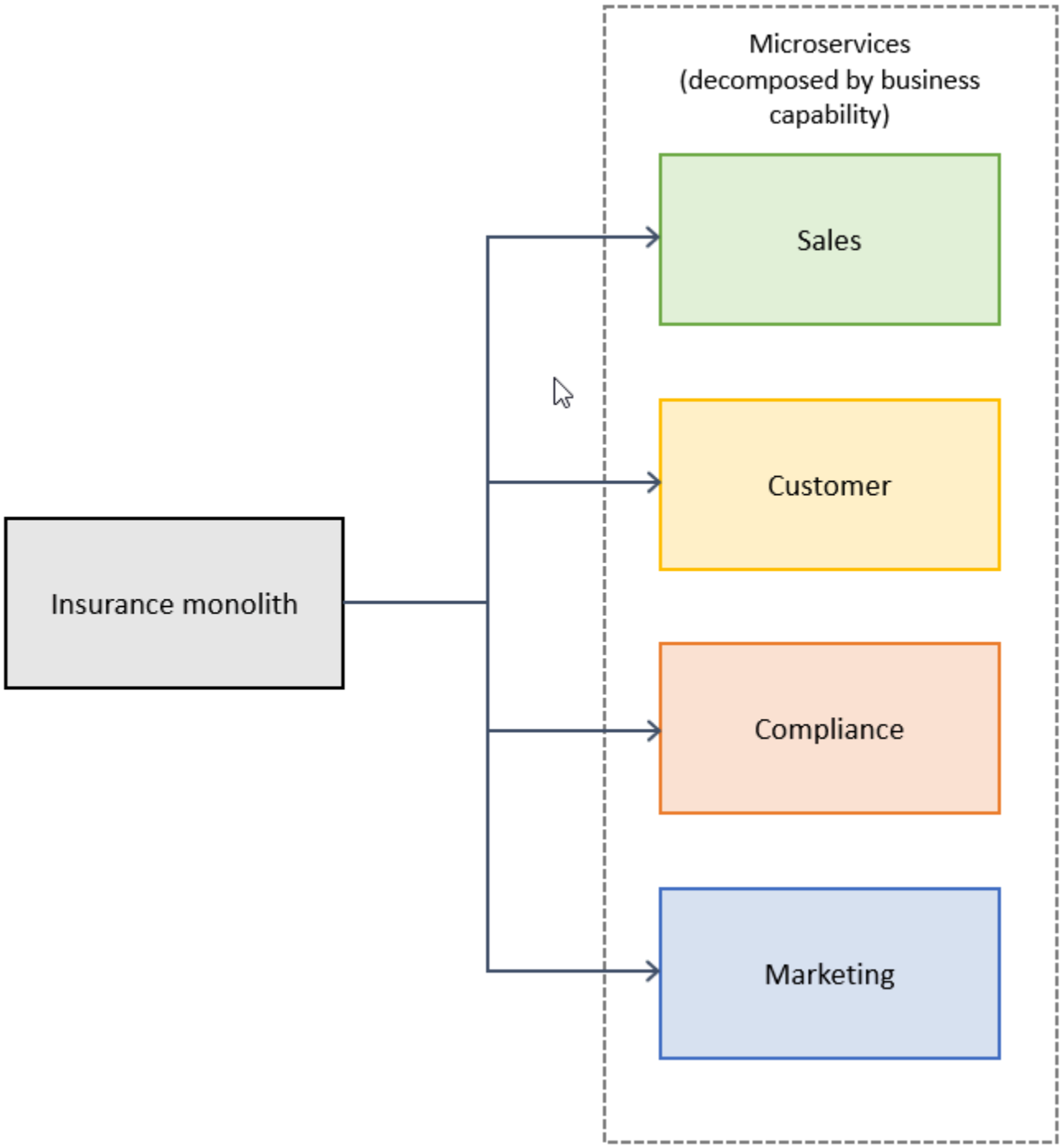
Decompose by business capability

You can use your organization's business process or capabilities to decompose a monolith. 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 explains the advantages and disadvantages of using this pattern.

Advantages	Disadvantages
• Generates a stable microservices architect ure if the business capabilities are relatively stable.	• Application design is tightly coupled with the business model.

Advantages	Disadvantages
• Development teams are cross-functional and organized around delivering business value instead of technical features. • Services are loosely coupled.	• 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 microservices based on business capabilities.



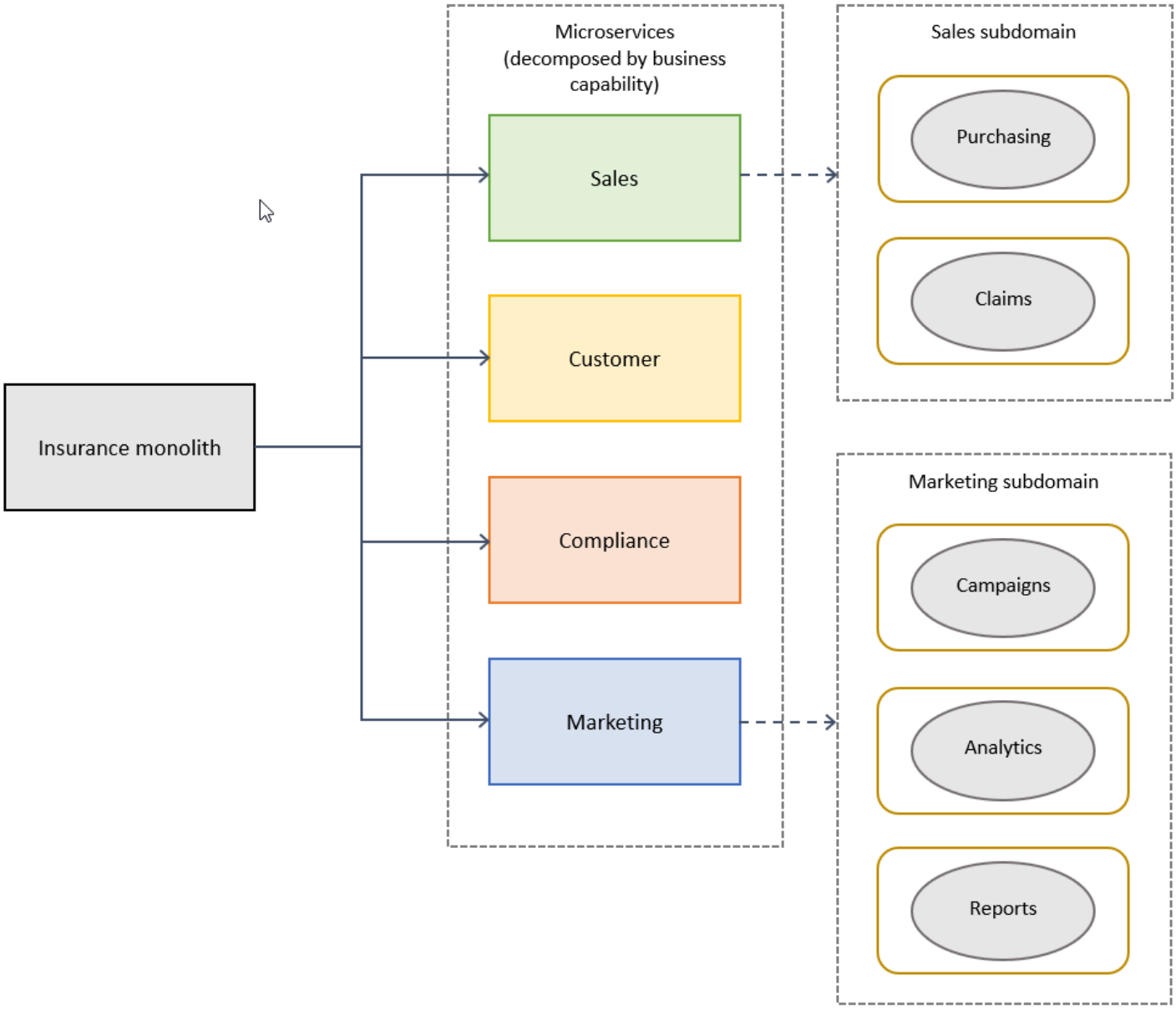
Decompose by subdomain

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 that 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 explains 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 understan ding of the overall business.

The following illustration shows how an insurance monolith can be 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*.

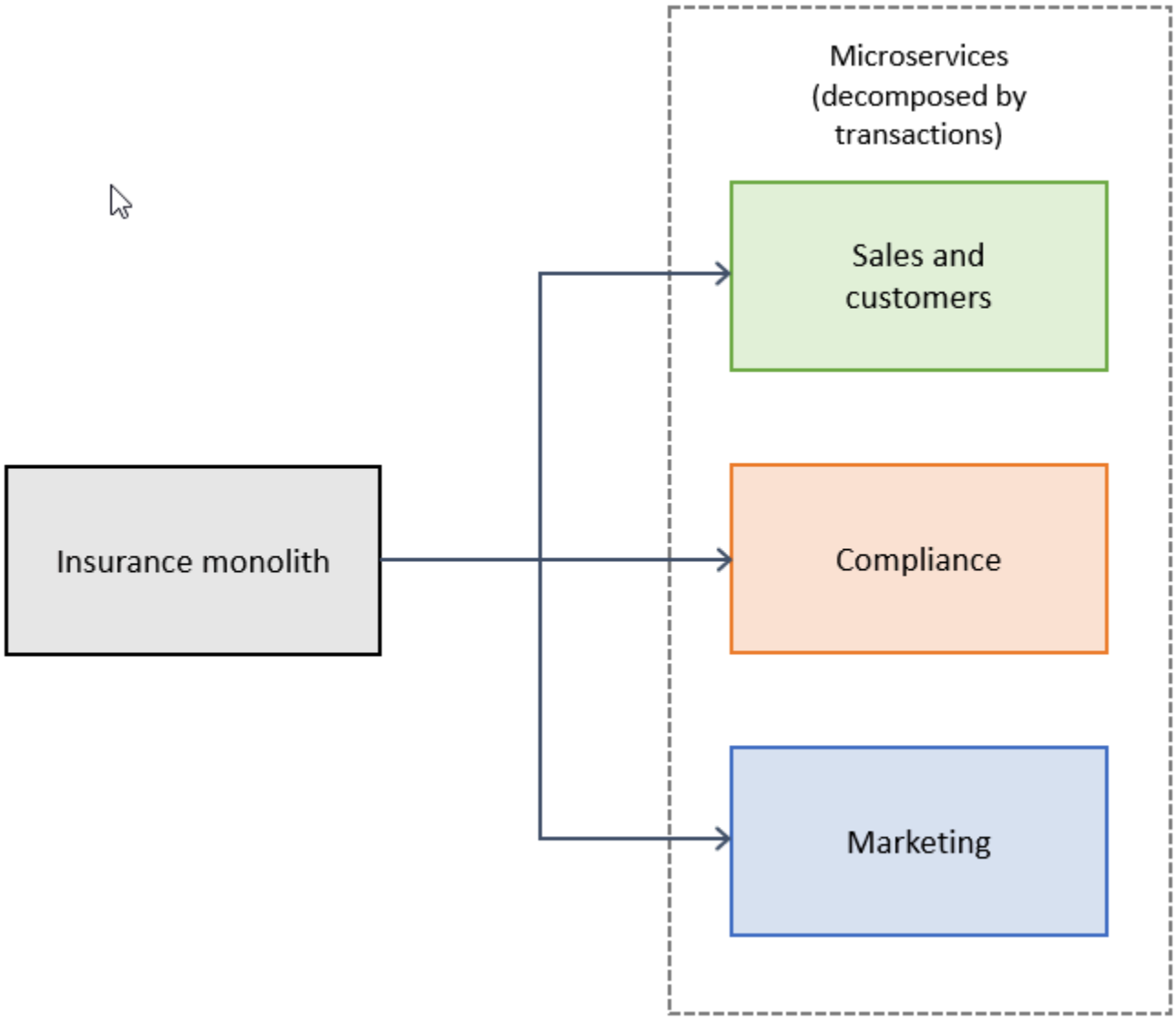
Decompose by transactions

In a distributed system, an application typically has 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 explains 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. • Multiple functionalities are implemented in a single microservice instead of separate microservices, which increases cost and complexity. • 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 coordination with both.

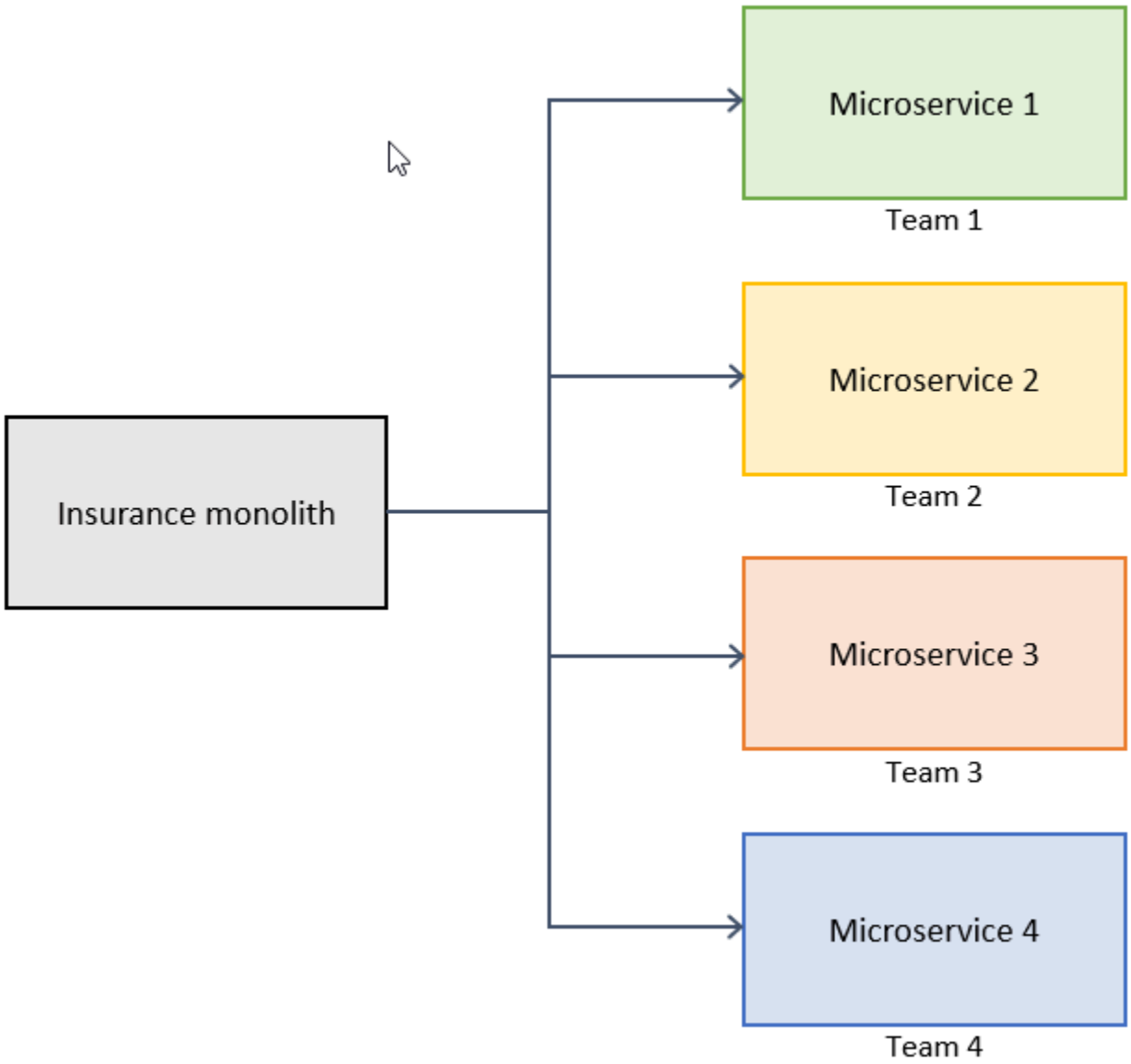
Service per team pattern

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 you assign each microservice to a single team. However, if the team is large enough, multiple subteams could own separate microservices within the same team structure. The following table explains 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 on product features. • Different teams can use different technolog ies, 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 can be split into microservices that are managed, maintained, and delivered by individual teams.



Strangler fig pattern

The design patterns discussed so far in this guide apply to decomposing applications for greenfield projects. What about brownfield projects that involve big, monolithic applications? Applying the previous design patterns to them will be difficult, because breaking them into smaller pieces while they're being used actively is a big task.

The strangler fig pattern is a popular design pattern that was introduced by Martin Fowler, who was inspired by a certain type of fig that seeds itself in the upper branches of trees. The existing tree initially becomes a support structure for the new fig. The fig then sends its roots to the

ground, gradually enveloping the original tree and leaving only the new, self-supporting fig in its place.

This pattern is commonly used to incrementally transform a monolithic application into microservices by replacing a particular functionality with a new service. The goal is for the legacy and new, modernized versions to coexist. The new system is initially supported by, and wraps, the existing system. This support gives the new system time to grow and to potentially replace the old system entirely.

The process to transition from a monolithic application to microservices by implementing the strangler fig pattern consists of three steps: transform, coexist, and eliminate:

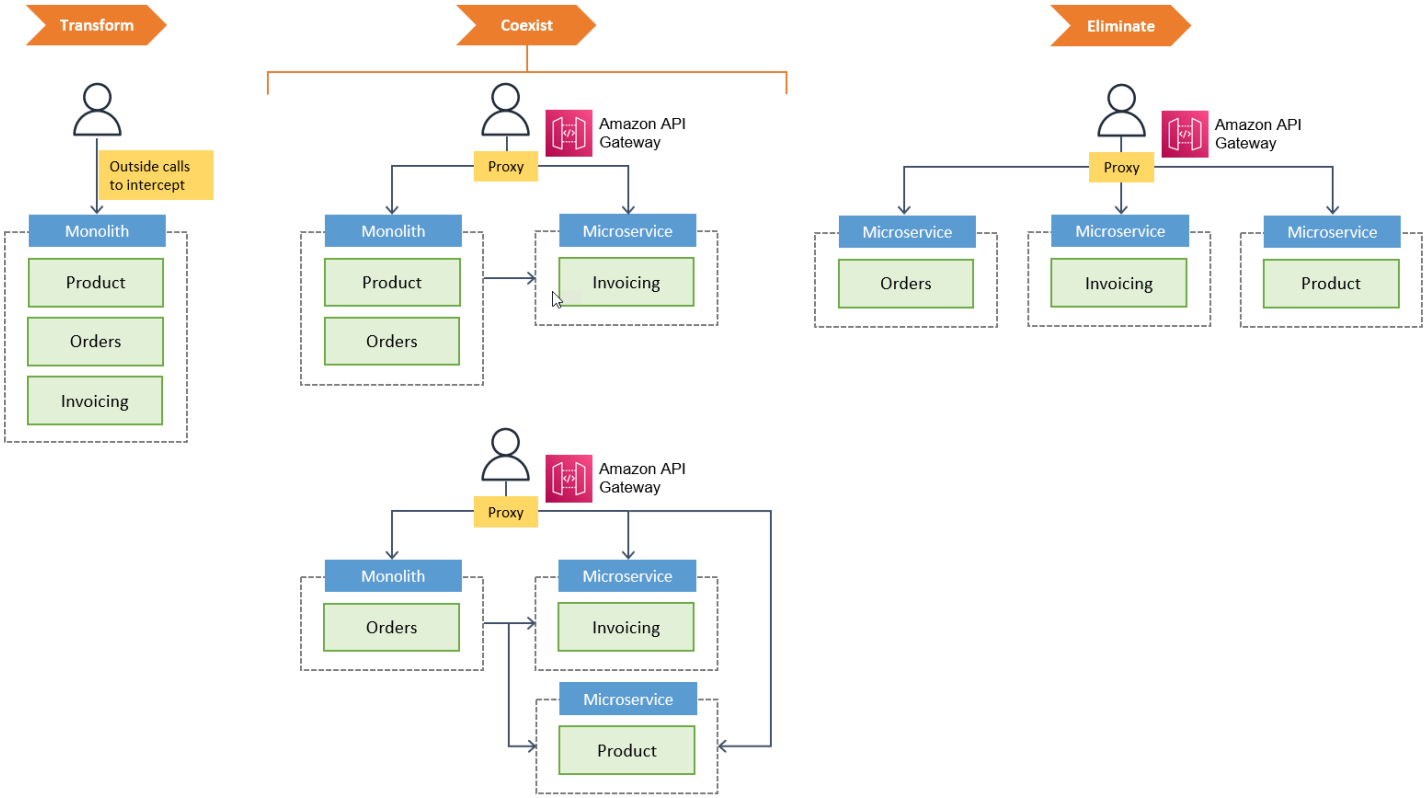
- *Transform* – Identify and create modernized components either by porting or rewriting them in parallel with the legacy application.
- *Coexist* – Keep the monolith application for rollback. Intercept outside system calls by incorporating an HTTP proxy (for example, Amazon API Gateway) at the perimeter of your monolith and redirect the traffic to the modernized version. This helps you implement functionality incrementally.
- *Eliminate* – Retire the old functionality from the monolith as traffic is redirected away from the legacy monolith to the modernized service.

The following table explains the advantages and disadvantages of using the strangler fig pattern.

Advantages	Disadvantages
• Allows for graceful migration from a service to one or more replacement services. • Keeps old services in play while refactoring to updated versions. • Provides the ability to add new services and functionalities while refactoring older services. • The pattern can be used for versioning of APIs.	• Isn't suitable for small systems where the complexity is low and the size is small. • Cannot be used in systems where requests to the backend system cannot be intercept ed and routed. • The proxy or facade layer can become a single point of failure or a performance bottleneck if it isn't designed properly. • Requires a rollback plan for each refactore d service to revert to the old way of doing things quickly and safely if things go wrong.

Advantages	Disadvantages
• The pattern can be used for legacy interacti ons for solutions that aren't or won't be upgraded.	

The following illustration shows how a monolith can be split into microservices by applying the strangler fig pattern to an application architecture. Both systems function in parallel, but you'll start moving functionality outside the monolith code base and enhance it with new capabilities. These new capabilities give you the opportunity to architect microservices in a way that best suits your needs. You'll continue stripping out capabilities from the monolith until it's all replaced by microservices. At that point, you can eliminate the monolith application. The key point to note here is that both the monolith and the microservices will live together for a period of time.



Branch by abstraction pattern

The strangler fig pattern works well when you can intercept the calls at the perimeter of the monolith. However, if you want to modernize components that exist deeper in the legacy application stack and have upstream dependencies, we recommend the branch by abstraction

pattern. This pattern enables you to make changes to the existing code base to allow the modernized version to safely coexist alongside the legacy version without causing disruption.

To use the branch by abstraction pattern successfully, follow this process:

1. Identify monolith components that have upstream dependencies.
2. Create an abstraction layer that represents the interactions between the code to be modernized and its clients.
3. When the abstraction is in place, change the existing clients to use the new abstraction.
4. Create a new implementation of the abstraction with the reworked functionality outside the monolith.
5. Switch the abstraction to the new implementation when ready.
6. When the new implementation provides all necessary functionality to users and the monolith is no longer in use, clean up the older implementation.

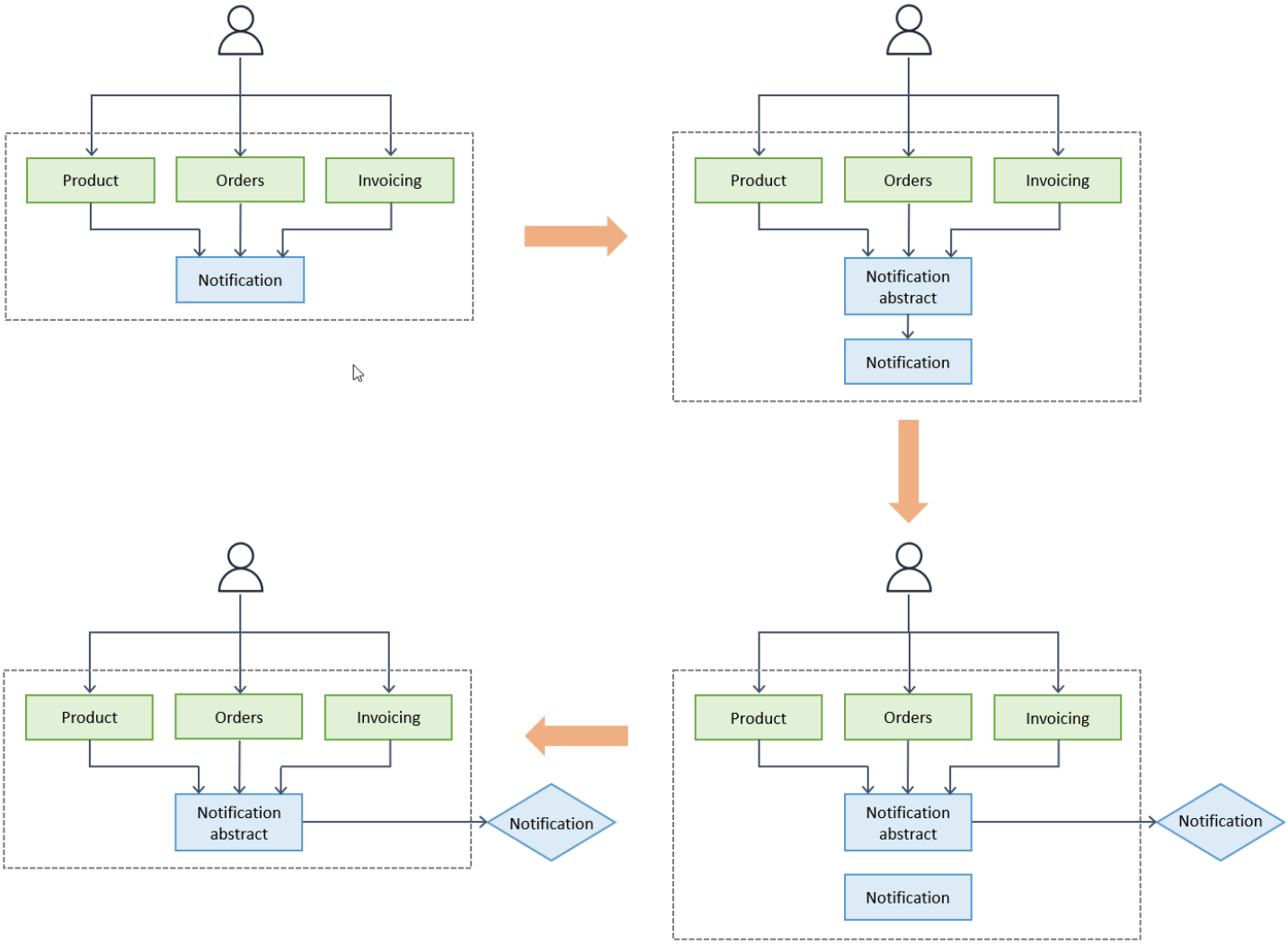
The branch by abstraction pattern is often confused with feature toggles, which also allow you to make changes to your system incrementally. The difference is that feature toggles are intended to allow the development of new features and keep those features invisible to users when the system is running. Feature toggles are thus used at deploy time or runtime to choose whether a particular feature or set of features is visible in the application. Branch by abstraction is a development technique and can be combined with feature toggles to switch between the old and new implementation.

The following table explains the advantages and disadvantages of using the branch by abstraction pattern.

Advantages	Disadvantages
• Allows for incremental changes that are reversible in case anything goes wrong (backward compatible). • Lets you extract functionality that's deep inside the monolith when you can't intercept the calls to it at the edge of the monolith.	• Isn’t suitable if data consistency is involved. • Requires changes to the existing system. • Might add more overhead to the developme nt process, especially if the code base is poorly structured. (In many cases, the upside is worth the extra effort, and the larger the restructuring, the more important it is to

Advantages	Disadvantages
• Allows multiple implementations to coexist in the software system. • Provides an easy way to implement a fallback mechanism by using an intermedi ate verification step to call both new and old functionality. • Supports continuous delivery, because your code is working at all times throughout the restructuring phase.	consider using the branch by abstraction pattern.)

The following illustration shows the branch by abstraction pattern for a *Notification* component in the insurance monolith. It starts by creating an abstract or interface for notification functionality. In small increments, existing clients are changed to use the new abstraction. This may require searching the code base for calls to APIs related to the *Notification* component. You create the new implementation of notification functionality as a microservice outside your monolith and host it in the modernized architecture. Inside your monolith, your newly created abstraction interface acts as a broker and calls out the new implementation. In small increments, you port notification functionality to the new implementation, which stays inactive until it's fully tested and ready. When the new implementation is ready, you switch your abstraction over to use it. You would want to use a switching mechanism that can be flipped easily (such as a feature toggle) so you can switch back to the old functionality easily if you encounter any problems. When the new implementation starts to provide all notification functionality to your users and the monolith is no longer in use, you can clean up the older implementation and remove any switching feature flag that you might have implemented


Product
Orders
Invoicing
Notification
Product
Orders
Invoicing
Notification abstract
Notification
Product
Orders
Invoicing
Notification abstract
Notification
Product
Orders
Invoicing
Notification abstract
Notification
Notification

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 pattern and then use the decompose by subdomain 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

Other resources

- Break a Monolithic Application into Microservices with AWS Copilot, Amazon ECS, Docker, and AWS Fargate

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
Added new patterns	Added information about the strangler fig and branch by abstraction patterns.	April 6, 2023
Initial publication	—	January 11, 2021