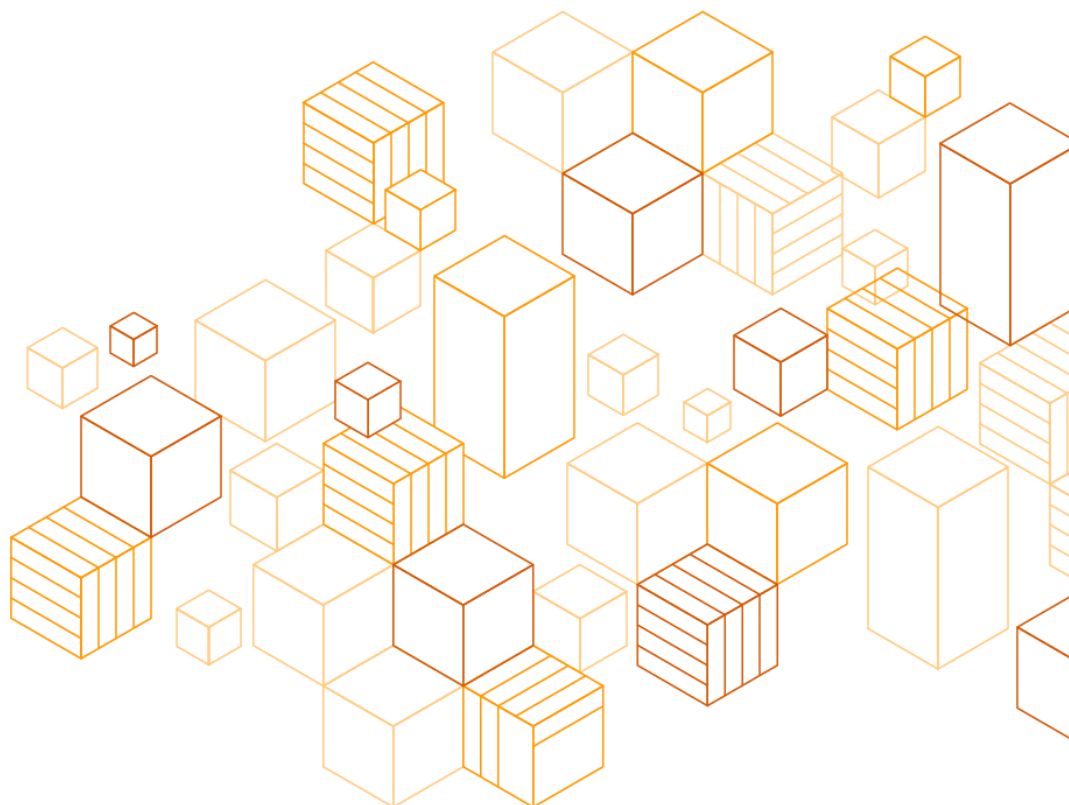# Monolithic to Microservice journey for .NET Applications

Technical Guide

October 21, 2022

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

About this guide

This guide covers considerations for refactoring a monolithic ASP.NET application into a microservice-based architecture. Database refactoring is not in scope for this guide.

Overview

Application modernization is the process of progressively transforming existing applications and infrastructure to extend to higher-value, cloud-based services to unlock new business capabilities, accelerate innovation, and reduce technical debt. One industry trend is that IT decision makers are expecting their application portfolios to grow by 50% in the next 2 years. This is because businesses need to add functionality in their applications to support improved experience and service for customers. Modernization provides a path for them to address challenges with their applications, such as increasing costs, scalability, operational resiliency, and security.

Organizations that modernize Windows workloads use a combination of strategies, including rehosting, replatforming, and refactoring to move applications to the cloud. These allow them to benefit from the economics, increased innovation, and the ability to deliver new functionality for customers. Organizations with .NET Framework applications, such as [DraftKings](), [CoStar](), and [AgriDigital](), plan to move their applications to the cloud by refactoring them to use the cross-platform version of .NET, and replatforming them to the Linux operating system.

Customers like [EposNow]() refactor to microservice-based architectures to help address business challenges related to slow scaling and complex management caused by monolithic applications that are not built using modern architectural practices.

The path to achieve modernization can be challenging. This guide will provide architects, developers, and IT professionals with the information needed to assess and begin modernizing their monolithic .NET Framework applications to the cross-platform .NET using a microservice-based architecture.

Starting with the introduction of .NET Framework in 2002, over ten million developers have built applications using .NET-based tools and frameworks. Initially, .NET applications could only run on the Windows operating system. This led to a landscape of portfolios of enterprise applications running on Windows.

In the past 10 years, changes in the technology industry have influenced the way developers build applications. These include trends such as increased adoption of Linux, the growing influence of open source, and DevOps practices, as well as and technologies like public cloud, containers, and serverless. These trends caused Microsoft to focus their .NET investments on a new version of .NET, initially named .NET Core and now simply called ".NET" (versions 5 and greater). This version of .NET is open source and cross-platform, adding the capability for .NET developers to run their applications anywhere, using a lightweight, modular framework.

There are benefits gained by businesses by refactoring .NET Framework applications to the cross-platform .NET on Linux, including the reduction of Windows licensing and access to the latest innovations from the .NET community. However, refactoring .NET Framework applications to the latest version of .NET presents challenges, particularly for applications that depend on libraries without cross-platform equivalents, including ASP.NET Web Forms, Windows Communication

Foundation (WCF), .NET Remoting, or Windows Workflow Foundation (WF).

Besides porting to the latest version of .NET, many organizations also want to break down their monolithic applications into a more manageable and scalable microservice architecture. The following sections of this guide walk through the use cases, patterns and considerations to modernize a monolithic ASP.NET to a microservice-based architecture.

Monolithic applications

Monolith vs. microservice approach

There are two categories of application architecture considered in this technical guide: monolithic and microservice-based. A monolithic application is one in which all functionality lives in one source repository and is delivered as a single unit. Over time, architectural patterns emerged to help break up applications into smaller, independent units of functionality, each delivered separately. The decomposition of applications into independent units of functionality began with the concept of Service Oriented Architecture and eventually grew into the microservices pattern.

In the past, building monolithic applications was the primary means of developing software applications, regardless of industry. An organization would develop a single application with sole control over all the data and functionality of the solution. Complexity and functional overlap would be introduced when there were multiple applications with similar purpose or data in use by different groups within an organization.

Designing applications using a microservice-based architecture decomposes a system so that each part handles a single business capability. Calls to the different microservices using well-defined interfaces comprise the entire system. Approaching application design as a series of microservices requires an initial planning effort, but enables applications to reap benefits of scaling, agility, and stability for the lifetime of the system.

Both of the approaches have considerations that help determine the suitability for a particular application. Although this technical guide focuses on decomposing monolithic applications microservices, that is not necessarily the right path for all architectures. Teams should choose the system architecture and design patterns for the project with a long-term plan in mind. Since design patterns change over time, system designs that were optimal at a project's inception may need to evolve.

Monolithic application architecture benefits

An application is monolithic when it contains components such as a user interface (UI), business logic, and data access code in a singular unit that is developed, tested, and deployed together. This guide considers two common types–the single-tiered monolith and the modular monolith.

Single-tiered monolith

A single-tiered monolith has the UI, business logic, and data access logic written as a single application that connects to a database. A change to any one part of the application requires testing and redeployment of the whole application. Such applications do not require well-defined modules related to functionality or business domains.

Modular Monolith

A modular monolith is like a single-tiered monolith because the entire application's functionality is deployed together.

However, modular monoliths have logical boundaries defined between modules that correspond to business or functional requirements. Because of tight code coupling, modules can interact with each other. The database of a modular monolith generally has similar structure and application access patterns as a single-tiered monolith.

There are some advantages to designing applications as monoliths. They include:

- **Development Model** - Many integrated development environments (IDEs) use a project structure that reflects the monolithic architecture. These IDEs attempt to simplify developer experience by combining the code for an application as a single project or solution structure. For smaller projects, this structure offers the benefit of being able to load the whole application into the development environment at once.

- **Test and Debug** - Developers can test and debug the entire application launching it in the IDE. They can also use test automation tools to run a single integration test or an entire suite to validate end-to-end functionality.

- **Application Deployment** - Deployment of monolithic applications is simple because there is only one application that deploys as a single unit, eliminating the need to coordinate the deployment of multiple applications and components at the same time.

- **Operational Overhead -** Operating monolithic applications is straight-forward, since there are few external dependencies, communication between modules is internal to the application and diagnosing issues involves troubleshooting a single application.

Drawbacks of a monolith architecture

While the simplicity of the monolithic architecture initially provides some benefits, there are inherent drawbacks. These become more pronounced as the application grows. Developers need to be increasingly diligent about the impact that changes may have. A change to one part of an application may result in unforeseen effects in other parts of the application. If the developer of a module is not aware of all the usages of their code, then updating that code in a single location can cause regressions in seemingly unrelated functionality. Even small change to a monolithic architecture requires end to end testing of the entire application. As the application becomes more complicated, it is less likely that a developer will be able fully understand impact on the entire code base.

Teams rarely design monolithic architectures with scaling as a primary concern. Monolithic applications run under the assumption that all modules of the application are immediately available as a part of the application process. This means that the way to scale out an application is to create a full copy on separate infrastructure. A load balancer or similar mechanism is required to manage traffic distribution. This method of scaling can be inefficient since there is no way to scale individual components. Parts of the application with low usage are scaled at the same rate as those that are heavily used.

By keeping all functionality together, an application can suffer from "noisy neighbor syndrome", in which the heavy usage of one part of the application can cause performance bottlenecks for the entire application, starving unrelated parts of the application of resources.

Monoliths are often stateful, relying on techniques like in-memory user session management. Applications may lose some benefits of scaling if users must remain bound to a particular instance, such as through the use of "sticky sessions". If one instance of the application fails, then all the users tied to it must restart their sessions. Changing a monolithic application to be stateless can involve a significant effort by developers, and may be prohibitively costly.

Because of the single code base of a monolith, updating to new technologies or coding patterns becomes increasingly risky as the application grows. Introduction of new technologies requires thorough testing of the entire application and increases the risk of unexpected regressions. This results in delays for even minor changes, such as upgrading the version of a library the application, or trivial database structure changes. Significant changes, such as shifting from a relational database to a non-relational database, often become too risky to even attempt.

There are well-known limitations of a monolithic architecture. They include:

- **Reduced agility**: Application becomes too large and complex for the team to make changes quickly without risk of high defects.

- **Reduced performance**: The size of the application can affect application start-up time and overall performance characteristics.

- **Deployment risk:** The team must deploy the entire application for each change. The effects of small changes have the risk to be wide-ranging.

- **CI/CD:** Large application size and specific runtime environment requirements make continuous deployment difficult (if not impossible). It is not unusual for large monolithic applications to have manual steps required in the deploy process.

Microservices

Microservice architectures

Microservice-based architectures comprise multiple fine-grained and loosely coupled services acting together as an application. Each microservice has a bounded context for a single business domain. Services communicate with each other either synchronously using REST or RPC, or asynchronously using a publish and subscribe (pub/sub) messaging model.
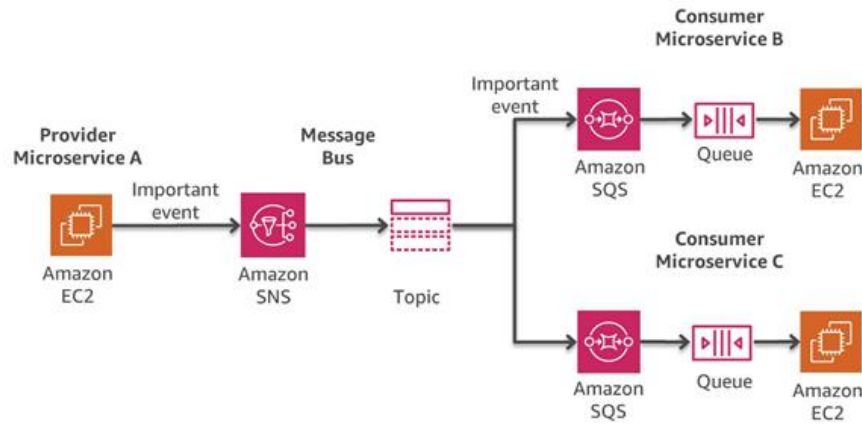
Figure 1: Asynchronous messaging and event passing between services

Each microservice has independent data storage separate from other services. The data store is part of the service implementation; other services cannot access it directly. Because data stores are specific to each service, the type of database can vary based on the service needs, and it is common for each microservice to use a purpose-built database.

The data store-per-service model can cause some challenges, such as data duplication, the atomicity of distributed transactions, and data consistency. When implementing transactions spanning services, the Saga pattern can help manage data transactions and consistency, and API composition or Command Query Responsibility Segregation (CQRS) can reduce the overhead of data retrieval.
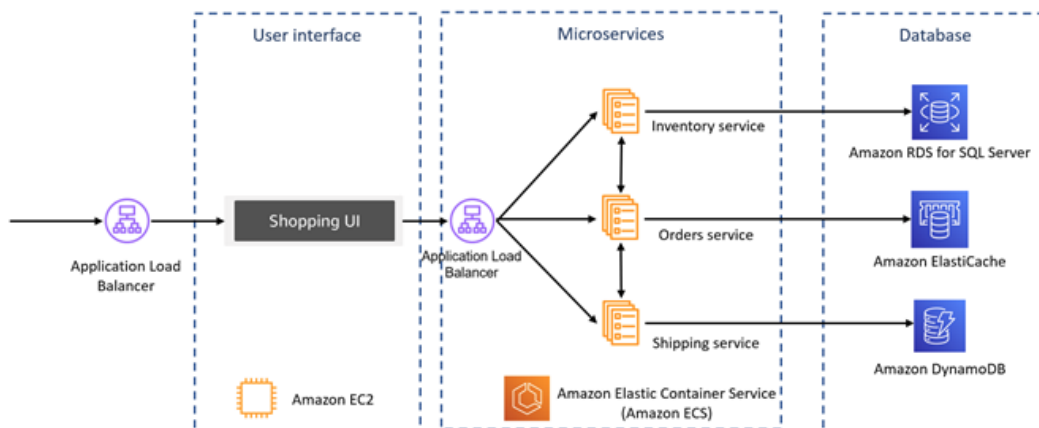

Figure 2: Synchronous communication between services using REST.
Each service uses an independent purpose-built data store

Benefits of microservice architectures

Migrating to a microservice-based architecture addresses challenges related to monolithic architectures through a combination of architectural changes and modern techniques, such as automated CI/CD processes.

A microservice-based architecture lets you to configure your application to scale efficiently, as opposed to monolithic architectures, which scale by creating additional copies of the entire application.

Th ability to separate out the functionality of your application into components that can scale independently addresses the "noisy neighbor syndrome" effect previously discussed. Consider the example in which the order processing portion of an e-commerce application needs additional resources to accommodate the required load. The increased usage could cause the order processing components to consume all the system resources. This could cause other parts of the system, such as the UI, to exhibit performance issues, despite not experiencing additional usage. In a microservice-based architecture, the order processing service can scale out as needed, leaving the rest of the application components unaffected.

The unit of deployment for each microservice is smaller than a full monolithic application. Each piece of functionality, a subset of the overall application, can be deployed on lightweight infrastructure such as containers or serverless functions (such as AWS Lambda).

Smaller units of deployment limit the scope of changes, reducing the risk of inadvertently affecting other areas of an application because functionality within the microservice is only accessible via well-defined interfaces. Microservices are simpler than monoliths to build, test and deploy, allowing more frequent delivery through CI/CD pipelines.

When to avoid microservice-based architectures

The choice to use microservice must be intentional and required to achieve business outcomes. Some examples of circumstances to avoid microservice architectures are:

- **The application is small or simple**–For less complex applications, the overhead and complexity of implementing and maintaining a microservice-based architecture may overshadow any benefit from using it.

- **The application domain does not fit in the microservice model**–If it is not possible to model an application as separate services represented by bounded contexts, then there may not be enough independence among components to facilitate a microservice design.

- **The application is static or has a limited life span-**One benefit of the microservice architecture is the ability to add additional services as the business domains change or increase. If the application is unlikely to change substantially, or is temporary, then the additional effort of building and maintaining microservices may outweigh the benefit.

- **Lack of expertise to build and maintain microservices -** If a team is small or does not have resources dedicated to building and operating microservices, then the organization may want to avoid a microservice architecture. Teams need adequate budget, skill, and capacity to build and operate microservice-based applications effectively.

Operational Considerations for Microservices

Distributed microservice-based applications introduce many benefits for businesses. However, a microservice architecture is not necessarily a "one size fits all" strategy. Along with the advantages that the pattern offers, there are operational considerations for ensuring that microservices are the right option for the business, team, and application.

Team Expertise

For teams that are experienced in traditional forms of software development, such as building single monolithic applications, the transition to building microservices may present challenges. The distributed nature of the architectures requires concepts and skills that may be unfamiliar to some development teams.

Building microservice applications requires the adoption of modern patterns such as CQRS, event sourcing, and Saga, requiring additional training and skill sets. Because there is no requirement to use the same language or framework for each microservice, teams could find themselves in a position where additional resources are required to maintain the applications over the long term because of the diverse technologies.

Microservices running in the cloud also introduce concerns for teams, requiring additional skills such as containerization, infrastructure as code, and a distributed security model.

When adopting microservice-based architectures for applications, the adoption of a DevOps culture within an organization is important. Active adherence to it by team members is important to help maintain effective collaboration, agility, and mechanisms needed to build and operate microservice-based applications successfully. A DevOps culture requires support from every level of the organization to be successful.

Operational Complexity

Microservice architectures comprise independent services that interact with each other. This introduces more moving parts than are usually present with monolithic applications.

Continuous integration and continuous delivery (CI/CD) automation for the build, test and deploy of individual microservices is critical to minimize the risk of the deployment process. The complexity of CI/CD for a microservice-based application increases both with the number of services and the different technologies used.

Inter-service communication poses a separate challenge. Because many microservices will communicate via HTTP or other network-based RPC protocols, latency may be an issue in microservice architectures, and service discovery becomes necessary. Implementation of components such as a service meshes (such as Envoy) and a service discovery catalog (such as AWS Cloud Map) allows services to find and communicate with each other, but also introduces another layer to operate and monitor.

Security

In monolithic architectures, access to an application is controlled by a single security mechanism that provides authorization for user access to functionality, and users usually accessed it through a single UI or API interface. In microservice applications, security plays an even more important role, since each independent service represents attack

surface area. Unauthorized access to one microservice must not enable unauthorized access to any other part of the system.

Microservice security design must be comprehensive and followed without exception. In microservice architectures, security boundaries need to exist where none did previously, and a centralized security mechanism (such as secure JWT validation or OIDC provider) is important to ensure the integrity of the application and its data.

Monitoring and troubleshooting

Traditional monitoring and troubleshooting techniques used in monolithic applications are not sufficient for microservice-based applications. The distributed nature of the applications can make pinpointing the cause of errors and performance bottlenecks complex. Individual errors may be difficult or impossible to reproduce reliably, and exception stack traces that display the entire call stack are not available.

Organizations need to implement observability tools that are suitable for microservice architectures. For application logs, teams can use services such as Amazon CloudWatch and Logstash for log collection, and OpenSearch for log aggregation and analysis. Distributed tracing tools such as AWS X-Ray can provide critical information about activity that span multiple microservices, increasing visibility and traceability. AWS Logging for .NET integrates popular logging frameworks such as ASP.NET Core logging, Log4Net, and NLog with AWS logging systems.

Data Consistency

One fundamental characteristic of a microservice architecture is the use of an independent data source for each service. Data for a single transaction will be in a consistent state only after all the involved microservices have completed their work. This is in contrast to a monolithic application using a single relational database, in which ACID (atomic, consistent, isolated, and durable) transactions and foreign key relationships help to maintain data consistency and integrity.

Designers of microservice applications need to take special precautions to allow data to exist in a consistent state not only between, but also during, transactions. Consider a microservice-based application where inventory and order data exist in separate data sources. Decrementing the amount of an item before placing an order may be required to prevent overselling. However, returning the item to inventory if the order is canceled is equally important to ensure that the stock levels remain accurate.

Patterns such as saga can help coordinate transactions in microservice-based architectures. These help teams to maintain consistency of data stored across an entire distributed application. Implementing the saga pattern can lead to more complex workflows and testing scenarios.

Monolith to microservices

Challenges of breaking up a monolith into microservices

One of the first challenges that teams face when decomposing an application into microservices is a lack of firsthand knowledge of the overall application. This lack of knowledge often manifests in three ways:

1) The original developers may no longer be with the company, so firsthand knowledge of the original design of the application and its design patterns may be lost.

2) The application may have grown, with add additional functionality added via "Scope Creep". This often results in the original design and architecture being compromised as the code base expands and changes.

3) The lack of application knowledge results in outdated or non-existent application documentation. Poor documentation often exacerbates missing application expertise.

As monolithic applications grow, modules are sometimes invoked in ways that were never intended. It becomes difficult to identify boundaries in code that define a microservice candidate. An example of this is when some parts of an application use a dependency injection (DI) framework to use a class, but other areas of the application use the constructor to instantiate objects. When classes intended as internal to a code module get called directly by other parts of the application, a "leaky abstraction" occurs. Because of these types of intra-connections, difficulties arise with defining the calling patterns for the code to replace with a microservice.

Monolithic architectures frequently require that code is running as a single instance on a server. This "singleton" behavior encourages coding practices in opposition to microservice decomposition. A monolithic web application end may store session state data in process memory on the web server. Although this approach is simple to implement and use, it results in an extracted microservice no longer having access to session data. Such design issues require upfront development effort to prepare the monolithic application before decomposition can start. In this example, moving the session state from process memory to external storage such as a cache or a NoSQL database can mitigate the issue.

With microservice extraction from the monolithic application, new challenges with deploying and testing present themselves. Because microservices are deployed separately, potentially by different teams or parts of the business, each will have its own requirements for testing and deployment. Transitioning to microservice-based architectures increases the number of deployments, so the automation of deployments is essential. For teams that do not use CI/CD, the resources required by the additional deployment efforts can quickly outpace the resources available. Therefore, each microservice should support a CI/CD process from the initial stages of development.

Test coverage

Test coverage for the existing code base helps with code refactoring. Test coverage identifies defects and functional anomalies during the refactoring process. An iterative refactor/test loop can reduce large refactoring tasks into smaller steps, each validated with test cases. If a code base does not have test coverage, developers can divide it into smaller logical units. They can then write test cases for the most critical functional units first, repeating the process for the remaining functional units.

Strategies to split a monolith

It is common for traditional ASP.NET applications to use layered architecture with code organized in logical layers of functionality, the most common ones being User Interface (UI), Business Logic/Application Logic, and Data Access layers.
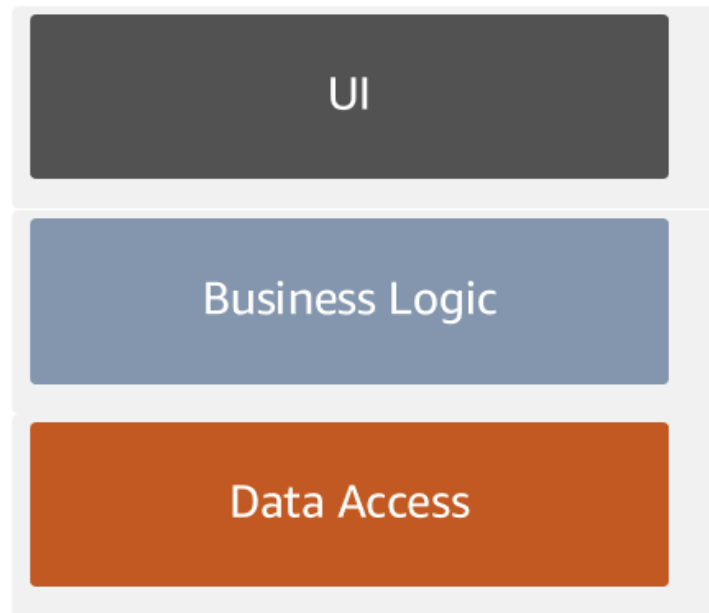
Figure 3: Typical layers of an ASP.NET monolithic application.

Domain Driven Design

One way to decompose a monolith application is to refactor the application's functionality according to business domains to create a bounded context for each model. A bounded context is the logical boundary of a business domain, inside of which a domain model applies. Modeling an application in this way is known as Domain Driven Design (DDD).
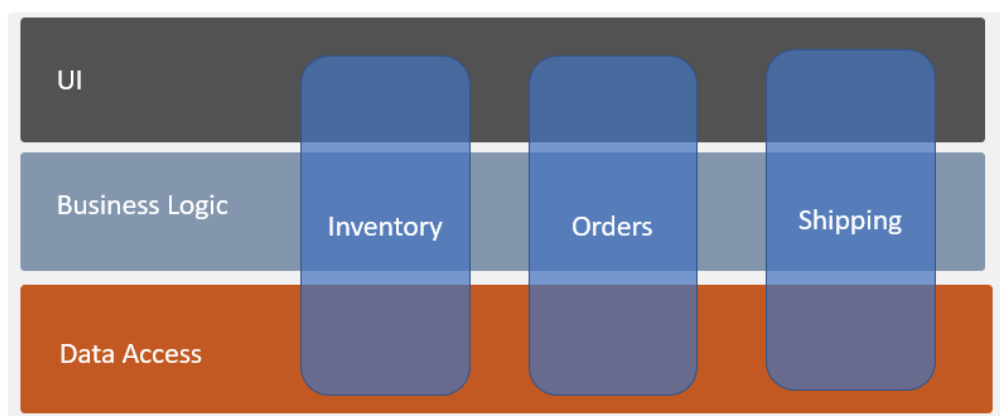

Figure 4: Separating application functionality by business domain.

Strangler fig pattern

The strangler fig pattern is a refactoring approach to replace the functionality of a legacy application incrementally with new services, without affecting the overall functionality. The includes identifying existing components and dependencies between them, grouping interdependent components into separate bounded contexts, and extracting them as an independent service. The remaining monolithic application communicates with the extracted services using asynchronous messaging or synchronous API calls.
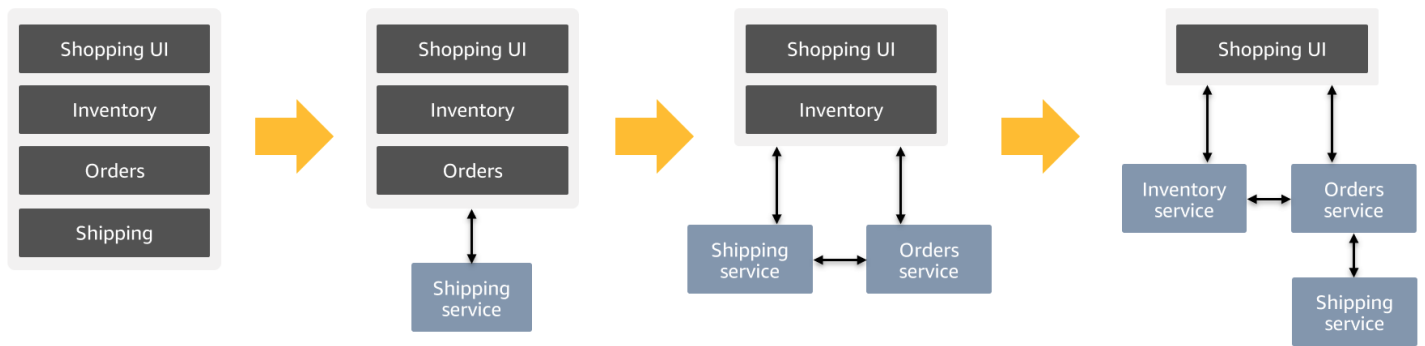
Figure 5: Example of the Strangler Fig Pattern

The following best practices for component selection will help you choose components to extract into microservices that will independently scale and deploy:

1. Select components that have good test coverage and low technical debt to reduce the upfront work required.

2. Select components that are likely to have scaling requirements independent of the rest of the monolithic application.

3. Select components that have frequent business requirement changes and are apt to require frequent deployments as a microservice.

The strangler fig pattern is one of the most common patterns to decompose existing monolithic applications into independent services.

Communication in microservices architecture

Once a team has started to decompose a monolithic application into microservices, the next step is to define the methods of communication between the services that compose of the application. It is important to choose the right communication platform for an application, both for the short and long term. Different communications protocols have different requirements to be considered when considering them for a microservice application.

Monolithic applications implement synchronous communication between libraries because it is the natural communication paradigm for an application deployed as a single binary codebase. When deploying an application as a collection of microservices, communication that provides fault tolerance, high availability and the possibility of supporting scaling on a per-service basis should be considered.

A simple method for communicating between an application component and an extracted microservice is the adapter pattern. This is actually how the AWS Microservice Extractor for .NET works. In this pattern, application logic makes calls to a component (the adapter) that is a wrapper for calls to the microservice. The adapter makes synchronous calls over the network to the microservice on behalf of the application logic.

One factor to consider when determining the method of communication to use is whether the microservice calls require synchronous communication. If there is a requirement for synchronous communication, the default method of routing communications is to implement an HTTP call directly to the microservice endpoint. Synchronous communications provide the developer with a simple method of connecting to the microservice. However, some challenges associated with a monolithic architectures remain present. Direct network connections to a microservice result in a strongly coupled system, which results in a single point of failure. Replacing a direct connection by placing the microservice behind a load balancer helps to mitigate this. The load balancer routes traffic to the microservice instances that are in service, enabling it to accommodate scaling actions automatically. Another technique that can be used is a service discovery model, where the calling code looks up the endpoint of the microservice.

For communication that is not required to be a synchronous, developers can implement a loose coupling model for interaction with the microservice. A typical method of loosely coupling a microservice involves the use of a queuing mechanism, such as Amazon Simple Queue Service (SQS). Developers integrate microservices with SQS by placing items into a queue for asynchronous processing by microservices, which periodically retrieve and process them. Implementing loose coupling through a queuing mechanism insulates applications from failures in microservices. Microservices can then independently scale based on the number of queued items. An event publication service such as Amazon Simple Notification Service (SNS) can be used to asynchronously notify many subscribers simultaneously.

Decompose the Database

There are multiple ways to structure your databases as you go from monolith to microservice. Following are two commonly used patterns.

Shared-database-per-service pattern

In the "shared-database-per-service" pattern, multiple microservices use a single database. This pattern is normally an interim architecture choice before fully adopting independent data stores for each microservice. Teams should consider avoiding this pattern for new microservices, since it is anti-pattern in that scenario. Teams should carefully assess the application data access patterns adopting the "shared-database-as-a-service" pattern, avoiding "hot tables, which are individual frequently-accessed shared among multiple microservices.

When using this pattern, database changes must also be backward-compatible to prevent unexpected application failure, because there are potentially dependencies in multiple codebases. For example, developers can drop columns or tables only if none of the running microservice versions reference them.

In Figure 6, all the microservices in a serverless [AWS Lambda](#) application share an insurance database, with an AWS Identity and [Access Management (IAM)](#) role controlling access to each of them. A change in the "Sales" service that requires a database schema change requires coordination with the developers of the "Customer" and "Compliance" services to ensure compatibility of the code with the updated database structure.
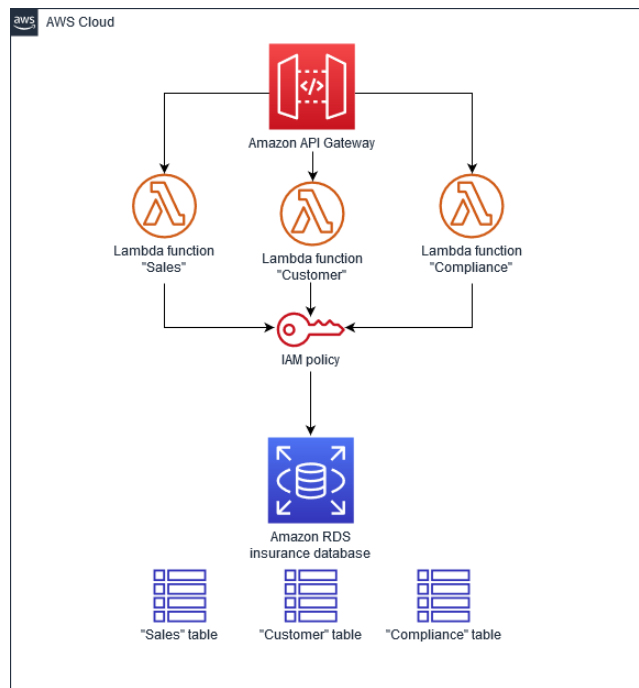

Figure 6: Illustration of Shared-Database-Per-Service Pattern

This pattern does not reduce dependencies between development teams, and introduces runtime coupling because all microservices share the same database. A long-running transaction in one service against one table, for instance "Customer" could indefinitely block access to that table by the other services.

Teams can consider "shared database-per-service pattern" in cases such as:

1. They are not able to redesign the existing data access layer.

2. They a need to maintain data consistency through ACID transactions, and are not ready to implement patterns such as saga.

3. They want to maintain only one database server instance for the entire application.

4. They have complex data dependencies among the microservices that prevent implementation of an independent for each microservice.

One-database-per-service

A defining characteristic of the microservice architecture pattern is the use of an independent data source for each service. Decomposing monolith's database into multiple service specific databases can achieve this. By design, only a microservice that owns a database can access it. In the previous example, ideally the "Customer" service will be the only one accessing a separate customer database, while the "Product" service will be the only one accessing the product database and so on.
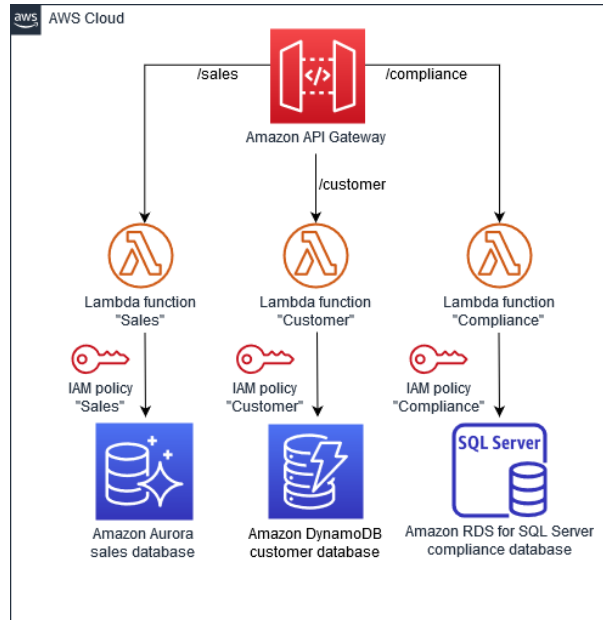

Figure 7.1: Illustration of One-Database-Per-Service Pattern

"One database-per-service pattern" introduces advantages such as:

1. The ability to scale performance independently across data stores.

2. The choice of database engine based on the data structure and access patterns of each microservice.

3. The capability to restrict access to each data store to independently.

Transforming an application to the "one-database-per-service" model allows teams to take advantage of these benefits, but the database refactoring process requires a similar degree of planning and effort as with the applications themselves. This guide does not cover database refactoring explicitly, but AWS offers prescriptive guidance to help teams get started in that journey.

Saga pattern

When decomposing a monolith into independent services, there may be scenarios in which a service needs to use multiple databases to complete a transaction. The saga pattern is a workflow that can achieve this.

The saga architectural pattern provides an approach to maintain data consistency in distributed applications by coordinating transactions among multiple microservices. In the saga pattern, microservice publishes an event for every operation, and the next operation starts based on the event's outcome. The overall transaction takes a path that depends on the success or failure of the individual events.

Figure 8 illustrates how the saga pattern can use AWS Step Functions to implement an order processing system. Steps with multiple potential outcomes, such as "ProcessPayment", have subsequent steps that handle both success and failure conditions, in this example "UpdateCustomerAccount" and "SetOrderFailure", respectively.
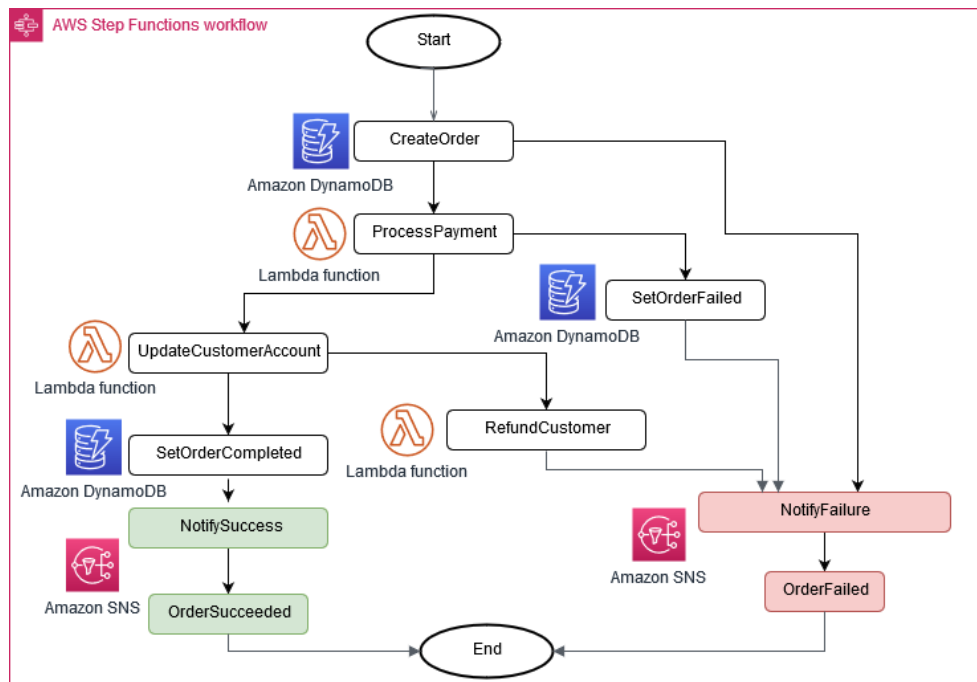

Figure 8: Illustration of the Saga pattern

The sage pattern may be useful for an application in the following cases:

1. The application needs to maintain data consistency across multiple microservices without tight coupling.

2. Long-lived transactions should not block operations on data by other microservices.

3. Transactions need to roll back if a single operation in the sequence fails.

Microservices technology choices

When moving to a cloud-based microservice architecture, it is important to make technology choices that will best meet business and technical requirements for the workload. Teams may come with pre-conceived notions of what they should choose, based on factors such as their own experience and current trends. This may cause situations such as a

team of developers selecting .NET solely because they have previous experience, or a choosing Kubernetes for container orchestration because of its popularity.

Compute

When building microservice-based applications in the cloud, the two most popular options are containers and serverless function-as-a-service.

Containers

Containerizing workloads allows the packaging of application code and dependencies, such as runtime libraries, together. Containerized applications have the benefit of portability, since they will run the same regardless of the underlying host infrastructure, because the application code is abstracted away from the underlying operating system. Containers are a good choice for microservice-based applications, because they are lightweight, portable, and integrate natively with modern DevOps tools.

The two most common types of containers that customers use for their applications are based on Linux and Windows, indicating both the virtualized container operating systems and the container engine host operating system. The choice of container type depends on the application code requirements.

Many runtimes, such as .NET 6.0 and NodeJS support both on Windows and Linux containers. If application code has dependencies on Microsoft Windows, such as .NET Framework, then Windows container are necessary. If an application also runs on Linux, there are advantages such as smaller container image size, faster startup time and little to no OS license costs. For applications running versions of .NET Framework, there are modernization tools available to help refactor the code to .NET 6.0 such as Porting Assistant for .NET.

On AWS, customers have a choice of two ways of hosting containers, using Amazon Elastic Compute Cloud (EC2), or as serverless containers using AWS Fargate. When using Amazon EC2, the customer provides (usually via automation) instances on which the container orchestrator place new containers. When paired with local image caching, this mechanism provides fast startup time for individual containers. In order to ensure there is always available capacity to place containers, excess capacity must always be available, or else avoid container startup delays may occur while additional hosts are provisioned. When using EC2 instance, application owners are responsible for OS patching.

AWS Fargate can run both Windows and Linux-based containers and eliminates the need to manage EC2 instances. The startup time for containers that are provisioned using AWS Fargate may be a slower than those running on Amazon EC2, which can cache container images locally.

For applications that can tolerate the slower container startup times, AWS Fargate is an optimal choice because of its more efficient use of resources and lower management overhead. However, if near-instant startup time is required, then using EC2 as the container host platform can achieve that outcome.

Container Orchestration

Each container should serve a single purpose, hosting a single application or component. Microservice-based applications that use containers comprise images deployed for the various services. Orchestrating the image, quantity, and connectivity of application containers is a critical aspect of operating microservice-based applications.

There are different container orchestration engines available to customers, such as Docker Swarm, Kubernetes, and RedHat OpenShift. Teams that run container-based applications on AWS usually select one of the cloud-based container orchestration services, rather than implementing container orchestration themselves.

Amazon Elastic Kubernetes Service (Amazon EKS) is a service that provides a managed Kubernetes control plane. Kubernetes provides capabilities such as storage management and orchestration, service discovery, load balancing, and automatic scaling. Amazon EKS can use both the EC2- or Fargate-based container engines.

For smaller applications that use few individual container images, Amazon Elastic Container Service (Amazon ECS) provides a simple orchestration mechanism. Microservices on Amazon ECS use one or more container images that run as discrete tasks, such as a web application with a reverse proxy, Amazon ECS manages the health and scale of containerized microservices and also integrates with other services that are useful in microservice-based applications, such as Elastic Load Balancing and AWS Auto Scaling. Amazon ECS can use both EC2- or Fargate-based compute.

Both Amazon ECS and Amazon EKS can use both EC2- and Fargate-based compute.

For customers building simpler microservices-based applications with fewer components, or who have little experience running containerized workloads, then Amazon ECS is a good choice to run containerized microservices. For customers that are already running Kubernetes, or are building large-scale applications with many microservices, then the advanced orchestration capabilities and managed service aspects of EKS provides make it a good choice.

Serverless functions

Another common choice for microservice-based architectures in the cloud is serverless functions. An example of this is AWS Lambda. Serverless functions provide a layer of abstraction on top of the underlying infrastructure and runtime. When deploying serverless functions, the user supplies an application code bundle and specifies a function entry point to be called upon invocation. In the serverless model, resources are allocated only when required.

Serverless functions do not run continuously, unlike containers. Instead, a function will start up in response to an event. Each instance of a function processes one request at a time. If multiple requests for a function occur simultaneously, the function will scale automatically to handle them. Examples of events are: HTTP requests, messages added to a message bus or queue, a scheduled timer, uploading of a file to object storage.

Because of the higher level of abstraction of serverless functions, the platforms are generally more opinionated than other types of compute, such as containers. For instance, in AWS Lambda, the managed runtime choice is limited to certain versions of supported (e.g., only long-term support (LTS) versions of .NET that runs on Linux such as .NET 6.0, but not .NET 7.0, which is a current release only). When using serverless functions, the code must adhere to a prescribed structure that allows it be invoked by the service. Because of the nature of this structure, that is multiple independent single purpose functions, teams will need to refactor code for compatibility with the serverless invocation model.

 Since serverless-based microservices use many cloud-based components acting together, automated deployment of both code and cloud infrastructure is necessary. Serverless-specific Infrastructure as code (IaC) tools, such as AWS Serverless Application Model (SAM), Serverless Framework, and Chalice, provide mechanisms to automate the creation and integration of serverless functions with other cloud-native services.

Making the choice of compute

When making the choice of a compute platform, there are both technology and business-based considerations to determine the right choice for the application. Examples are:

1. Will the benefits to the business and customers from choosing this technology that outweigh additional complexity or management required for the chosen service?

2. Does the team already have the skillset to build applications with this compute service? Will there be training or additional staff needed?

3. Will the technology being chosen able to accommodate future needs of the business, both in terms of functionality and scale?

4. Will the selected compute platform meet the security and compliance requirements of the business?

5. Are there automation tools can simplify and eliminate risk of human error when deploying applications, and are they compatible with the team's current CI/CD tools?

Both containers and serverless offer scalable and flexible options for building microservice architectures. Building applications on a cloud provider such as AWS also enables access to other cloud-based services. Some teams prefer to build applications to be entirely based on serverless technology, while others opt to use a more traditional application model with containers, integrating native AWS services with it.

If a team wants a traditional web application framework for their microservices, such as ASP.NET Core, Express.JS, or Flask, then using containers for microservices makes sense. By using these with AWS Fargate with Amazon ECS or Amazon EKS, then they get the resilience, scalability and high availability of the cloud, while still being able to use other AWS services. If the team is building an application that is highly distributed, event driven, or message-based, then serverless compute such as AWS Lambda as the central, along with one of the serverless deployment frameworks, may be the right decision.

If a microservice-based application needs both longer-running compute, and also relies on messaging and is event-based, then both containers and serverless functions for different parts of the application can be used. For instance, the team may choose serverless to process events related to uploading documents for OCR and publishing metadata to a database, with containers used to process background processes related to processing the extracted data in the uploaded documents.

Database

Mentioned previously, of the key characteristics of microservice-based architectures is the use of an independent data store for each individual service. This is in contrast to monolithic architectures, in which one data store contains most or all of the data, usually in a single relational database (RDBMS).

One advantage of building applications in the cloud is the selection of database types available. The availability of purpose-built databases allows teams to pick the right one for each microservice, optimized for the type of use case. In practice, this paradigm is known as "polyglot persistence". When choosing cloud-based databases (either relational or otherwise), teams use the consumption-based pricing model of the cloud, removing the need to purchase commercial database licenses. Using cloud-based managed database service also reduces operational overhead, because the cloud

provider (e.g. AWS) manages the underlying infrastructure and database platform, and the customer is only responsible for building applications and managing data.

Following are some common choices of purpose-built and cloud-based databases commonly used in microservice-based applications:

- **Relational Database Management Systems (RDBMS)** – Relational databases are usually chosen use cases data with a fixed structure. Relational databases naturally maintain data consistency through ACID transactions. Examples: MySQL, PostgreSQL, Amazon Aurora.

- **Document Databases** are NoSQL databases that store data without a prescribed structure or enforced relationships to other data. They are often used for read-intensive workloads such as product catalogs. Examples include MongoDB, Amazon DocumentDB, and CouchDB.

- **Key-Value Databases** are NoSQL databases that stores data using a hash table data structure (hence the name "key-value"). Key-value databases offer high performance storage and retrieval of unstructured data. Use cases include user profiles, session state, and shopping cart data. Example: Amazon DynamoDB.

- **In-Memory Databases** are high performance NoSQL database that store unstructured in memory, allowing for sub-millisecond access time. Teams use in-memory databases for frequently accessed, ephemeral data, such as user sessions, and as a caching layer in front of other, slower data stores. Examples include Redis, Memcached, Amazon ElastiCache (Redis/Memcached) or Amazon MemoryDB.

- **Time Series Databases** are NoSQL databases designed for high throughput data ingestion in temporal order. They are often used for Internet of Things (IoT) applications and storing metrics or telemetry data. Example: Amazon Timestream

- **Graph Databases** are NoSQL databases that both data and a representation of the connections among data items. They are often used for fraud detection, recommendation engines, and social applications. Examples: Neo4j, Amazon Neptune.

- **Ledger Databases** store transaction data and provide cryptographical verification of each transaction, providing an auditable, immutable transaction history. Financial applications often use ledger databases when a single, verifiable source of truth is required. Example: Amazon Quantum Ledger Database.

Selecting the right type of database for each microservice is important. Similar to the selection of the compute platform, there are factors when making a database choice.

1. Does the business value added by the database engine outweigh the additional expertise required and refactoring effort of refactoring the codebase that it introduces?

2. Will the functionality of the database being chosen for each microservice be able to accommodate future needs of the business?

3. Will the performance of the database be able to accommodate the projected size of data?

4. Does the database have high availability and resilience that will allow the application to meet the required SLA for customers?

Security considerations for microservices

Security is a critical consideration with any application, but the distributed nature of microservices introduces additional considerations when designing and implementing them. By introducing security best practices and standards into the foundation of the microservice-based application, companies can avoid the risk, cost, and complexity of a reactive approach to security.

Authentication and Authorization

Since monoliths get deployed as a single entity, there is usually a single authentication and authorization mechanisms for the entire application. This may include a username and password for a web application or API keys for a server-based API. Even if separate a separate identity provider (IdP) handles authentication and authorization of consumers of an application, there is still common code in the application that controls access.

In microservice-based architectures, it is critical to implement access control for each service, for both end-user client access and service-to-service communication. By ensuring that each call to any microservice is authorized, unintended access to any service will be prevented.

For user-facing services, authorization frameworks, such as OpenID Connect (OIDC) or OAuth 2.0, can be used to control access. This enables limiting the scope of access for each identity to only the allowed services and functionality with each service. Using an independent identity provider, such as Amazon Cognito, controls access to applications using both Cognito-specific and federated user identities. One benefit of choosing Amazon Cognito for identity management is its ability to integrate with other AWS services, such as Amazon API Gateway and Application Load Balancer.

Service-to-service communication must also be secured to maintain defense in depth. One such technique for communications is mutual TLS (mTLS), where x.509 certificates are used to both authenticate inter-service communication and encrypt the traffic. Many service mesh products, such as Amazon App Mesh, support mTLS.

Another method is to use JSON web tokens to validate identity and authorize access. Using this technique, each service will attempt to verify the token's digital signature by a trusted certificate, in order to trust the data contained within the JWT.

Increased Attack Surface

In monolithic applications, there are usually only a few ways to access the system, such as a single web application or API endpoint. For applications that accept external (Internet-facing) traffic, only certain network ports and protocols are allowed by firewalls, either into the corporate data center or cloud-based virtual network. Using Network Address Translation (NAT) avoids exposing services directly, limiting the public-facing part of the application to a single IP address and port.

When designing microservice-based applications, using the principle of defense in depth enables teams to reduce the number of attack vectors. Defense in depth includes securing the perimeter of the application at each layer, making it more difficult for an attacker to gain unauthorized access to any services.

For example, when running microservices in an Amazon Virtual Private Cloud (VPC), the following steps can reduce the risk of a larger attack surface area:

- Services endpoints such as containers and API endpoints that do not require a public-facing presence should use in private subnets, disabling direct access from the Internet.

- Network Access Control Lists (NACLs) should to restrict access between subnets, limiting only the required traffic for the application.

- Security group rules should restrict traffic to each service from the intended clients only.

- Non-public microservices accessed from on-premises clients should be accessible via a secure, dedicated network connection, such as AWS Site-to-Site VPN or AWS Direct Connect.

- Services that need to be accessed directly from the public Internet should use perimeter security tools like AWS Web Application Firewall (WAF) to detect and protect against malicious traffic, and AWS Shield to protect against distributed denial of service attacks (DDoS).

Data Encryption and Protection

All traffic in a microservice-based application should be encrypted. This includes not only traffic between client applications and the individual services, but also inter-service communication. Obsolete TLS protocols and cipher suites should be avoided.

When running microservice-based applications in AWS, certificates can be issued, managed, and automatically renewed using Amazon Certificate Manager (ACM) for public facing services, or AWS Certificate Manager Private Certificate Authority (Private CA) for internal-only services. ACM and Private CA both integrate with many services commonly used in microservice-based architectures, such as Amazon API Gateway, Amazon Cognito, and Elastic Load Balancing.

It is also important to enforce encryption at rest for all application data. Data should only be able to be accessed by parties with access to the correct encryption keys. Amazon Key Management Service (KMS) integrates with most other AWS services and allows customers to either bring their own encryption keys or use allow AWS to manage them.

Enforce isolation at the database layer

In monolithic applications, there is often no security boundary between different parts of the application or within its database(s). This means that if a malicious actor gains access to the monolithic application, they can gain access to any data in the application. An example of this is with SQL injection vulnerabilities, where an attacker runs arbitrary queries against any table in the application database, regardless of part of the application accessed.

Providing both application and database-level isolation in microservice applications reduces the "blast radius". If an attacker exploits a vulnerability in a microservice, they will only have access to data belonging to that service, reducing the impact of a security event.

This adds additional complexity to microservice architectures, as opposed to monolithic applications, because there will be many additional databases to maintain and monitor, along with access control for each unit of compute, by controlling access to the data via database user or AWS IAM role.

Monitoring and Logging

Logging and monitoring are central to detecting, understanding, and preventing security issues within microservice applications. By collecting data related to microservice access and usage and by whom, teams can characterize usage patterns across microservices and detect unusual activity.

Besides instrumenting microservice application with logging functionality, teams running cloud-based applications can also analyze of logs emitted by services such as API gateways, container orchestration services, and cloud-based databases without having to write additional code.

There are many applications available to teams that enable the monitoring of entire microservice applications. When operating microservice applications in AWS, Amazon CloudWatch is a common choice, because of the ability to collect both logs and performance metrics from applications and AWS services. Amazon CloudWatch logs and metrics can be consumed by applications like Splunk, NewRelic, and Amazon OpenSearch Service for analysis and threat detection. AWS services such as Amazon Managed Service for Prometheus offers Prometheus-compatible monitoring service for containers, and Amazon Managed Grafana can be used to visualize and analyze data from multiple sources

Cost Considerations

With cloud computing, companies have access to a scalable platform, low-cost storage, database technologies, and tools on which to build enterprise-grade solutions. Cloud computing helps businesses reduce costs and complexity, adjust capacity on-demand, accelerate time to market, increase opportunities for innovation, and enhance security.

Weighing the financial considerations of operating an on-premises data center compared to using cloud infrastructure is not as simple as comparing hardware, storage, and compute costs. Whether you own your own data center or rent space at a colocation facility, you have to manage investments that include capital expenditures, operational expenditures, staffing, opportunity costs, licensing, and facilities overhead. AWS prescriptive guidance for cost can help choose right pricing model that works the best for organizations.

Decomposing a monolithic ASP.NET monolith application

A monolith can be decomposed by using organization's business capabilities. A business capability is what a business does to generate value (for example, sales, customer service, or marketing). This approach uses 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). This pattern is appropriate for existing monolithic systems that have well-defined boundaries between subdomain-related modules.

The other pattern is to decompose by transaction. 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.

Irrespective of the decomposition strategy, breaking up a monolith requires understanding the co-relation between various parts of the application and re-factoring the code into independent services. This process can be tedious and manual. Assistive refactoring tools such as AWS Microservice Extractor for .NET can simplify the process of refactoring older monolithic applications into smaller code projects to build a microservices-based architecture.

AWS Microservice Extractor for .NET

AWS Microservice Extractor for .NET is an assistive tool that can help with the process, providing an understanding of a monolithic .NET application's structure. It provides teams with information to help identify candidate microservice functionality and will provide automated code transformation to help with the process.
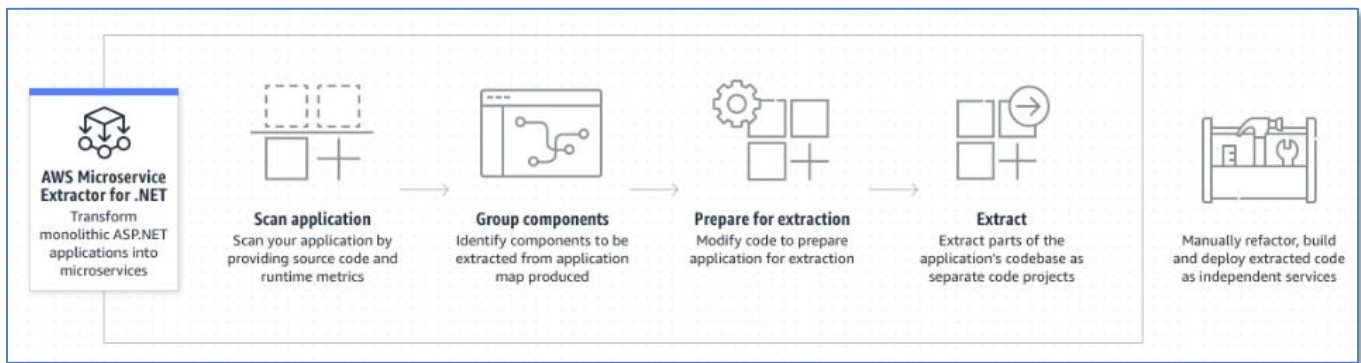
Figure 9: Microservice Extractor for .NET Steps

A typical monolithic application allows components to call other components, and does not limit access to the shared data store (Figure 10).
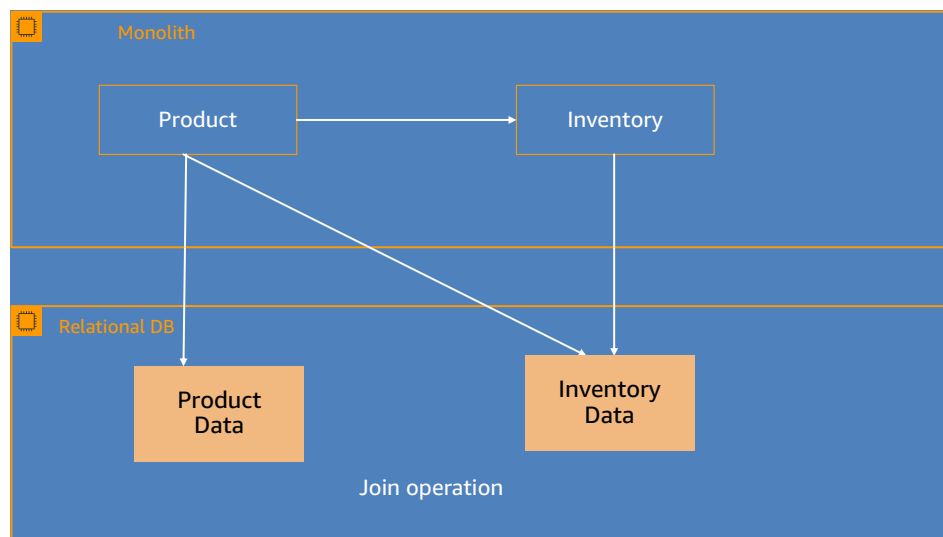

Figure 10: Typical Monolith

Microservice Extractor can help model the monolith as one or more separate microservices. This includes the understanding of dependencies and automation of migrating functionality to separate services (Figure 11).
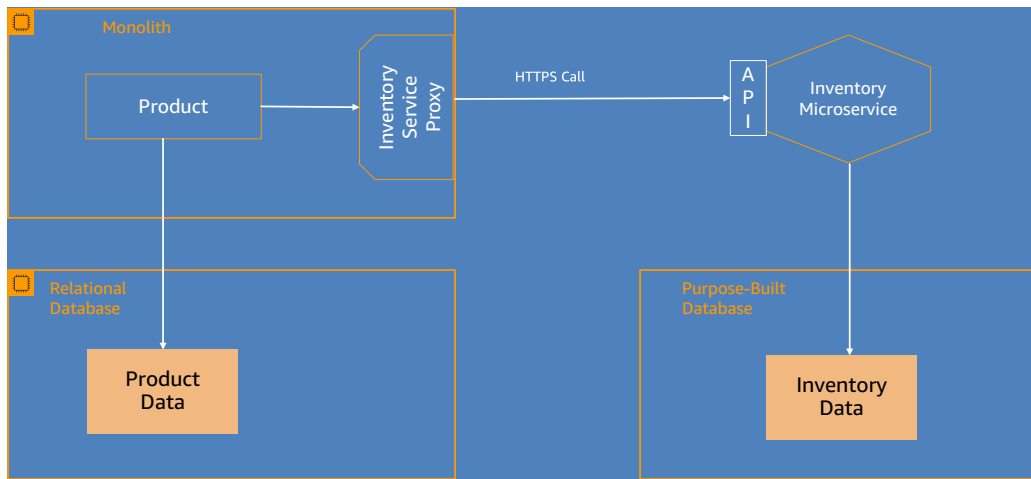
Figure 11: Monolith Following Extraction

Analyze application

Microservice Extractor's supported use cases are in the product documentation. The first step in the process is to onboard the monolithic application into the tool for analysis.

There are additional options available to assist in the analysis of the monolithic application. These include runtime profiling data, which will add call count to the application visualization, providing to understand not only the direction of class dependencies but also the relative importance. Information on collecting runtime information for IIS based applications can be found in the Microservice Extractor documentation. Runtime profiling data is optional.

If upgrading the new microservice from .NET Framework to .NET 6.0 is planned, then Microservice Extractor can also use the Porting Assistant for .NET program, to include information about .NET compatibility when visualizing the monolithic application.

Visualize

The Microservice Extractor for .NET enables visualization of individual classes that comprise the monolith, along with the dependencies that they have on each other. Microservices Extractor displays the visualization as a series of nodes connected by edges. Each node represents a class or interface in the monolithic codebase.

The individual nodes provide information about the type of each component. These include things such as models, data access code, and UI components. The edges connecting the nodes are directional, with the plain end showing the source component (i.e. caller) and the arrow end showing the target component (i.e. the called). Nodes that are "starting points", meaning that they have dependencies, but are not dependencies themselves, may good candidates for a microservice extraction.
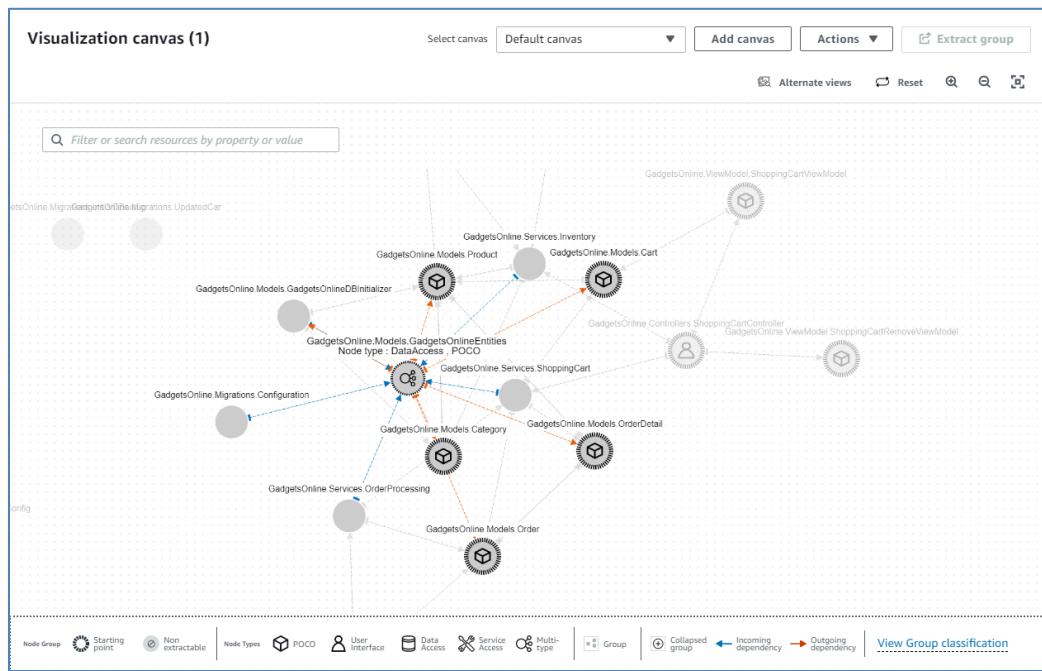
Figure 12: Visualization Canvas

Visualization of the monolithic application serves several purposes in when moving from a monolithic to a microservices-based architecture.

Visualizing the entire structure of the application is helpful for planning the refactoring effort by providing information about the current structure of the application. This type of structural information about the application may not otherwise be available. Without an understanding of the application structure, determining how to decompose the application is difficult.

Also, visualization enables the understanding of the dependencies between the components. This gives visibility to dependencies that may not be obvious when browsing the monolith's source code directly. For example, the tool shows all code referencing an Entity Framework data context, which normally is accessed from a repository service. If there are components that reference the data context directly, rather than through the repository, it will be obvious, and a refactoring plan can be put into place to update code to use the repository instead.

When visualizing the monolithic application structure, Microservice Extractor provides three views, providing different types of insight into the application structure. The first is the standard visualization that gives a view of the entire application.

The second is namespace view (Figure 13). This shows all the components, grouped by top-level namespace. This provides understanding of the high-level organization of the codebase.
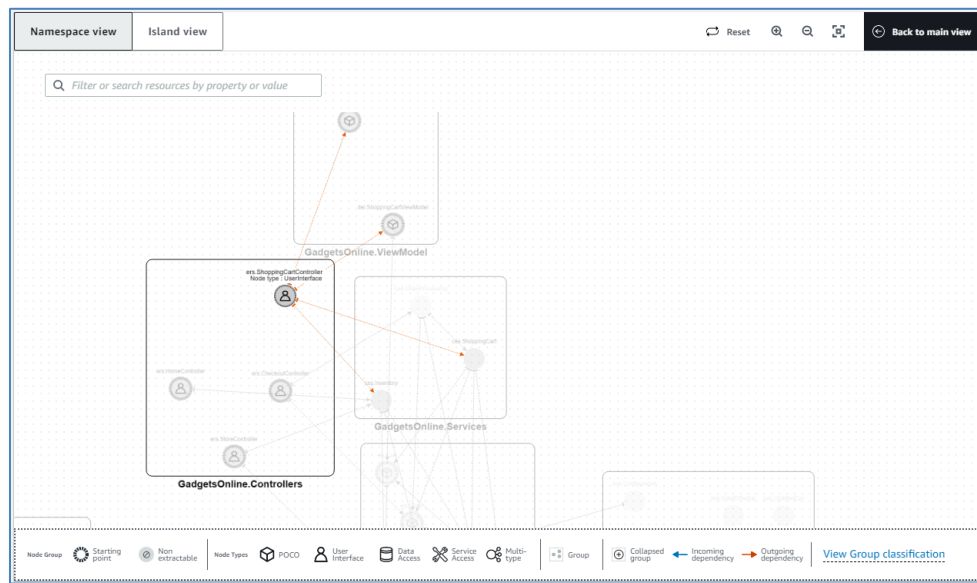
Figure 13: Namespace View

The third visualization type displays islands of functionality within the monolith (Figure 14). Islands are independent pieces of functionality with no external dependencies.. Island view is a useful way to find parts of the application that are unused, or can be easily separated out into separate projects.
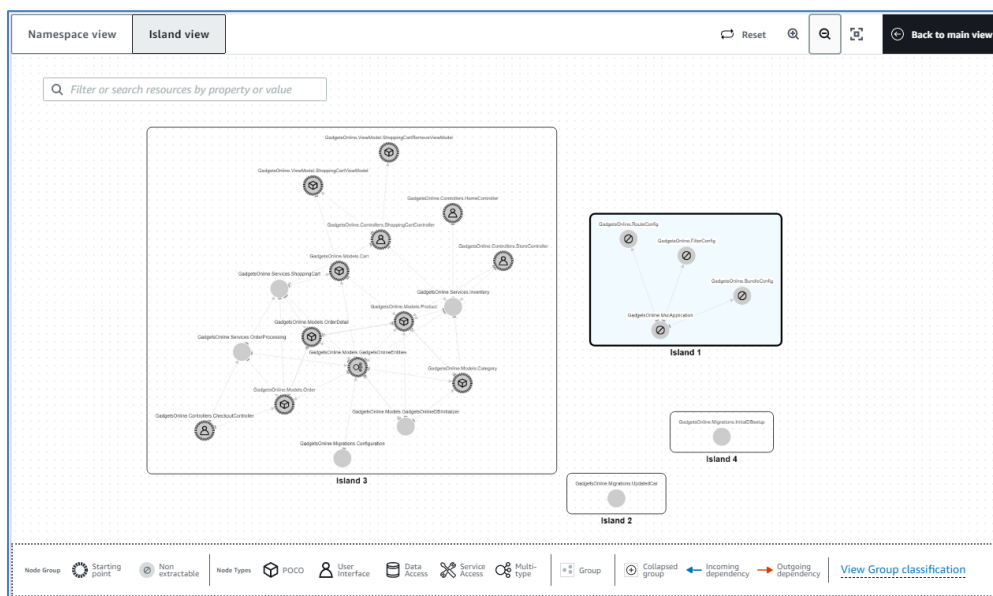

Figure 14: Island View

Although the Microservice Extractor does not support not all project types for extraction, it supports all .NET projects written in C# for visualization.

Group

The grouping functionality of Microservice Extractor (Figure 15) helps teams to plan what functionality to separate as a microservice. Microservice extractor enables the manual grouping of components to prepare for extraction. When you group classes together, they will display together in the visualization canvas, and remain together during automated extraction.

Determining the groups for the individual microservices can be challenging for even moderately complex monolithic applications can be a challenging ordeal. In order to help make the process simpler, Microservice Extractor provides machine-learning automated recommendations to suggest component grouping. This provides a starting point for planning the decomposition of the monolith into microservices. (Note: this feature is in preview at the time of this writing. To request access to it, please send a note to mailto:microservice-extractor-preview@amazon.com with a valid AWS account number.)

When grouping classes, Domain Driven Design (DDD) is the general guiding principle followed. With DDD functionality groups according to business domain rather than arbitrary functionality. In an e-Commerce example, functional domains might involve inventory management, the shopping cart experience, or order management.
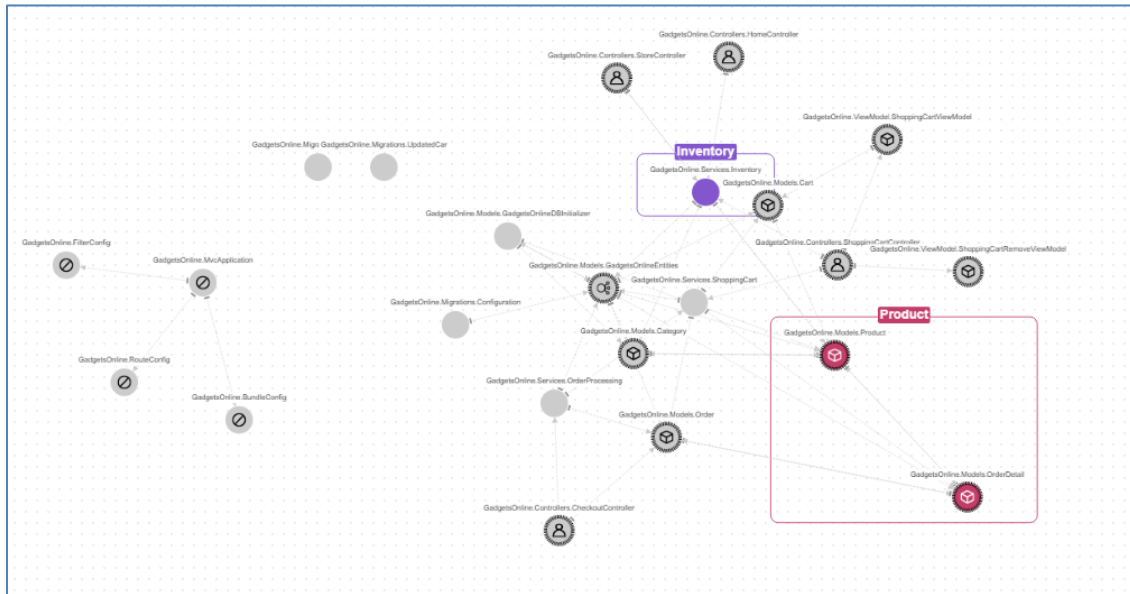

Figure 15: Grouping Classes Prior to Extraction

Extract

When decomposing a monolithic application to a microservices architecture using the strangler fig pattern, functionality is removed from the monolith and created as a new service. When extracting components in this way, it can eliminate some of the overhead of starting from scratch and allow the use of code that has already been written and tested. Extraction of existing functionality gets this advantage when the source monolith and destination microservice using compatible runtimes (e.g. .NET monolith with .NET microservices, or Java monolith with Java microservices).

The extraction process (either automated or manual) includes the creation of the destination microservice, plus the modification of the monolith to make calls to it. For instance, in a monolith, business logic may exist in a .NET assembly statically linked with the main application. Calls to the business logic occur natively within the process itself. However, when replacing those calls with calls to a HTTP-based microservice, the monolith must be updated to include functionality that can make the network calls.

When using the Microservice Extractor automated extraction feature, it will allow the extraction of the individual groups into separate microservices, adding code to the original monolith for invoking the microservice.

For example, if inventory data were accessed as a repository in monolithic code, the application might look like this:

```
Inventory inventory = new Inventory();
```

Microservice Extractor will produce code that implements the adapter pattern, demonstrating how to access the newly extracted service:

```
IInventory inventory = InventoryEndpointFactory.GetEndPointAdapter();
```

The extraction process is only the beginning of breaking up a monolith into a microservice-based application. This document has mentioned the challenges and benefits of adopting a microservice extractor, and the business transformation required. Microservice extractor is one way to accelerate adoption, but really represents the starting point.

Next Steps

Once a team has successfully extracted one or more separate microservices (either manually or by using a tool like Microservice Extractor for .NET), they will need to adapt the applications to meet the needs and standards of the organization. This includes the following steps:

- The new microservice code base(s) needs to be added to version control and integrated into the company's CI/CD strategy, automating as much of the process as possible.

- The teams responsible for the new microservices should to familiarize themselves with the new codebase, understand the business use cases and plan for any additional refactoring required to use the microservices. This includes updates to implement team code and security standards, of approved libraries, or the refactoring of duplicated code into common packages, such as through a private team NuGet repository.

- The entire application, both the monolith and extracted microservices need to be thoroughly tested for functionality, performance, and security. Adding microservices will introduce additional factors to be aware of, such as the latency introduced by HTTP communication. The entire application must be tested to verify that it will run as expected in production.

- Teams should document the process used to separate microservices from the original monolith. The strangler fig pattern intends for the pattern to be repeated for each microservice. Having a repeatable process for a monolithic application will reduce the amount of work required to add future microservices.

To get hands-on experience with Microservice Extractor, an online workshop walks through the extraction of a microservice from a monolith application from beginning to end, providing examples of several types of refactoring required for implementation. It uses the Gadgets Online open source sample project to illustrate the entire process.

Conclusion

The benefits of using microservice architecture are scalability, optimized cost, adoption of modern software development practices, and quicker time to market. While microservice-based applications have advantages over monoliths, not every application is a suitable candidate for a microservice architecture. Developers should evaluate tradeoffs between architectures before refactoring a monolith application to microservice architecture. Anyone considering refactoring a monolithic .NET application into independent services can use this guide to understand the patterns and process better to accelerate the adoption of microservice architecture.

 AWS Microservice Extractor for .NET is a tool to help customers decompose monolithic ASP.NET applications into smaller code projects that are deployed as independent services. It is primarily meant to help developers who will make code changes and refactor applications. It also assists software architects and business analysts with identifying the business processes that need to be modeled as separate services. AWS Microservice Extractor for .NET simplifies refactoring older monolithic applications into smaller code projects to build a microservices-based architecture.

Contributors

Contributors to this document include:

- Aditi Sharma, WW GTM Specialist Sr Manager, Amazon Web Services.

- Craig Bossie, Specialist Sr Solutions Architect, Amazon Web Services.

- Runeet Vashisht, Specialist Principal Solutions Architect, Amazon Web Services.

- Tom Moore, Sr .NET Developer Advocate, Amazon Web Services.

Document Revisions

Date	Description
October 2022	First publication.