AWS Migration Plan: C# Monolithic to Microservices-Based Architecture

1. Introduction

This document outlines the plan for migrating an existing C# monolithic service to a microservices-based architecture on AWS. The migration adds values in unlocking new business capabilities, accelerating innovation, and reducing technical debt, and aims to improve costs, scalability, maintainability, operational resiliency, and security.

Not all applications need the microservice-based architecture. The microservice-based architecture is suitable for applications in complex domains that require complex solutions, which become unmanageable due to the size and complexity of the domain itself. While microservices split that complex domain into simpler bounded contexts, encapsulating the complexity and reducing the scope of changes, they add complexity to the solution of implementation. When the reduction in complexity of the domain outweighs the increase in complexity of the solution, we evaluate our application to be suitable to be migrated to the microservice-based architecture.

2. Current Monolithic Architecture Overview

- Built with C# and .NET Framework/Core
- Single codebase handling multiple business functionalities
- Hosted on on-premises servers or a single cloud VM
- Relational database as the backend

3. Target Microservices Architecture on AWS

To illustrate with an example of a C# service for managing products, here is what a user requesting a product's information look like before the migration:

1. The user sends a login request with their credentials to the monolithic application.
2. The application validates the credentials and, if valid, generates an authentication token for the user.
3. The user sends a request to view a product, including the authentication token in the request header.
4. The application checks the authentication token and retrieves the product details from the Products table in the database.
5. The application retrieves the product inventory details from the Inventory table in the database.
6. The application responds with the basic information and inventory details for the product, which are displayed by the UI..

And here is what it looks like after the migration:

1. The user sends a login request with their credentials to the authentication service.
2. The authentication service validates the credentials and, if valid, generates an authentication token for the user.
3. The user sends a request to view a product, including the authentication token in the request header, to the Product Catalog microservice through API Gateway.
4. The Product Catalog microservice checks the authentication token and retrieves the product details from its Products table in DynamoDB.
5. The Product Catalog microservice responds with the product details.
6. The application UI sends a request to access the inventory information, including the authentication token in the request header, to the Inventory Management microservice through API Gateway.
7. The Inventory Management microservice checks the authentication token and retrieves the inventory details for that product from its Inventory table in DynamoDB.
8. The Inventory Management microservice responds with the inventory details for the product.
9. The application UI displays the basic information and inventory details for the product.

3.1 Key Design Principles

- **Service Decomposition:** Break down the monolith into independent, domain-specific microservices. A typical C# monolithic service
- **API Gateway:** Centralized entry point for managing API requests.
- **Event-Driven Communication:** Utilize messaging for inter-service communication.

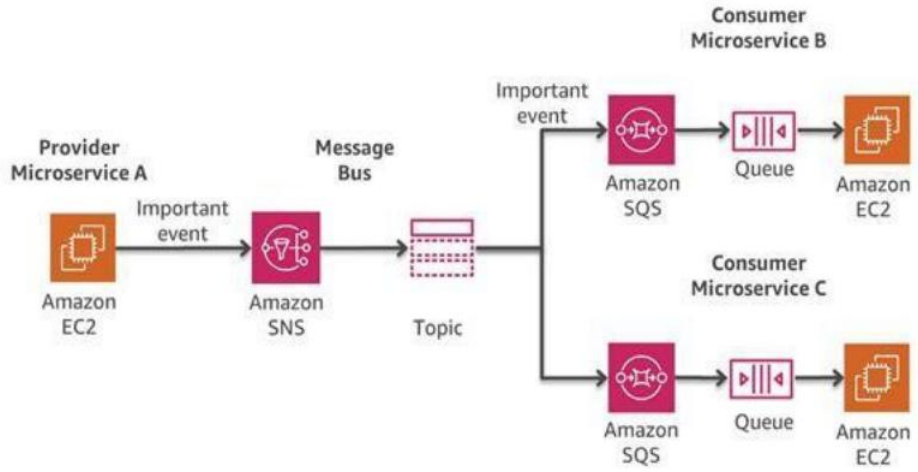

Figure 1: Asynchronous messaging and event passing between services

- **Database Per Service:** Each microservice should own its data to ensure loose coupling.

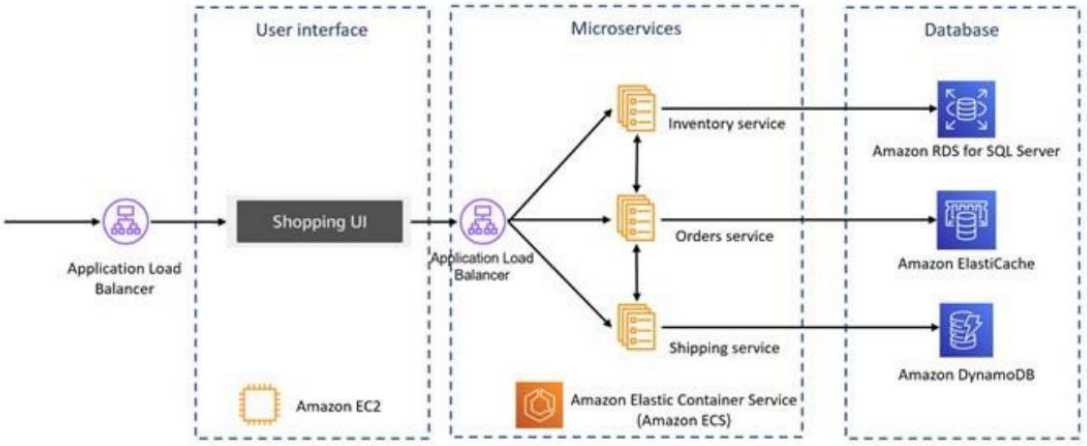

Figure 2: Synchronous communication between services using REST. Each service uses an independent purpose-built data store

- **CI/CD Pipelines:** Automate build, test, and deployment processes.

3.2 AWS Services Selection

- **Compute:** AWS Fargate (ECS) or AWS Lambda (for event-driven services)
- **API Gateway:** Amazon API Gateway for routing external requests
- **Service Discovery:** AWS App Mesh or AWS Cloud Map
- **Messaging:** Amazon SQS, SNS, or Amazon EventBridge for decoupling
- **Databases:** Amazon RDS, DynamoDB, or Aurora for separate data stores

- **Observability:** Amazon CloudWatch, AWS X-Ray, and AWS OpenSearch for monitoring
- **CI/CD:** AWS CodePipeline, AWS CodeBuild, and AWS CodeDeploy for automation
- **IAM & Security:** AWS IAM, AWS Secrets Manager, and AWS WAF for security

4. Migration Approach

4.1 Strangler Fig pattern

The Strangler Fig pattern is a refactoring approach to gradually replace monolithic functionalities with microservices, without affecting the overall functionality. This 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 microservices using asynchronous messaging or synchronous API calls. Usually prioritize components being extracted in the following ways: (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.

For example, for a C# monolithic application / service for managing products including inventory, order, shipping functionalities, and a UI, the first step could be designing the shipping service, the remaining application using API to request the shipping service; then, designing the order service, and finally designing the inventory service so the UI is eventually solely rely on API calls to request services.

1. **Identify independent components in the monolith.** A monolith can be decomposed according to an organization's business capabilities.
    1. One approach uses domain-driven design (DDD) to decompose existing monolithic systems that have well-defined boundaries between subdomain-related modules. Each user interaction goes through the minimum possible number of services (ideally only 1).
    2. The other pattern is to decompose by transaction, so the system avoids latency issues or two-phase commit problems, when it needs to call multiple services to complete one business transaction. This pattern is appropriate if response times are important and different modules do

not create a monolith after packaging.

3. Calls to the different microservices using well-defined communicating interfaces comprise the entire system. Using the adapter pattern, **AWS Microservice Extractor for .NET** makes it easy to refactor older monolithic applications into smaller code projects and create adapters which are wrappers that make synchronous calls over the network to the microservice on behalf of the application logic, when it produces code of abstract interfaces that implements the adapter pattern, demonstrating how to access the newly extracted service.

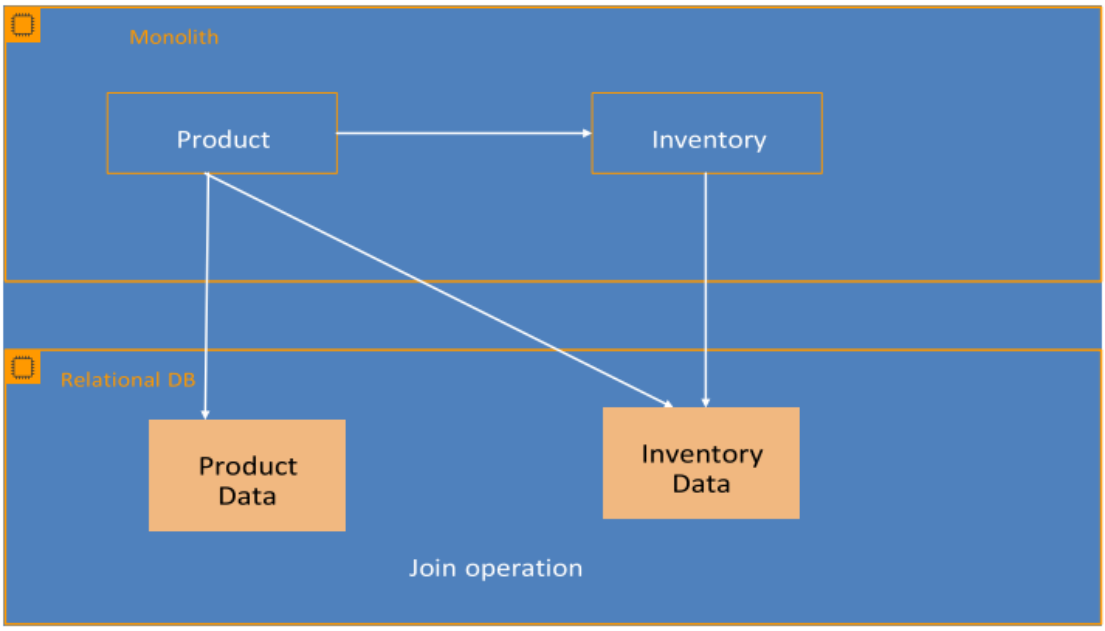

Figure 10: Typical Monolith

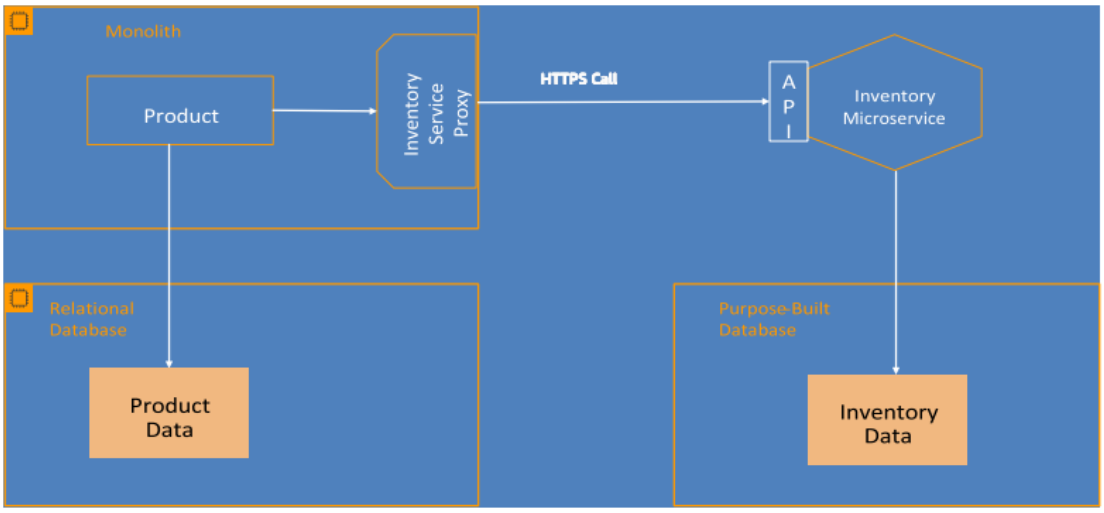

Figure 11: Monolith Following Extraction

2. **A key characteristic of microservice-based architectures is the use of an independent data store for each individual service.** Restructuring the

monolithic database requires breaking down the database according to each microservice's directly related data. To determine where to split a monolithic database, first analyze the database mappings. Like the service extraction analysis, analyze database usage and map tables to the new microservices, and keep in mind issues like data synchronization, transactional integrity, joins, and latency. When separating the core functionalities and modules into microservices, typically use APIs to share and expose data or replicate data in the dependent service's database. Based on the existence of scenarios in which a service needs to use multiple databases to complete a transaction, The Saga architectural pattern can be leveraged to provide an approach to maintain data consistency in distributed applications by coordinating transactions among multiple microservices. In the pattern, a microservice publishes an event for every operation, and the decision on the next operation is based on the success or failure of the individual events. AWS Step Functions can be used to implement such a system. Steps with multiple potential outcomes, such as "ProcessPayment", have subsequent steps that handle both success and failure conditions, for example, "UpdateCustomerAccount" and "SetOrderFailure", respectively.

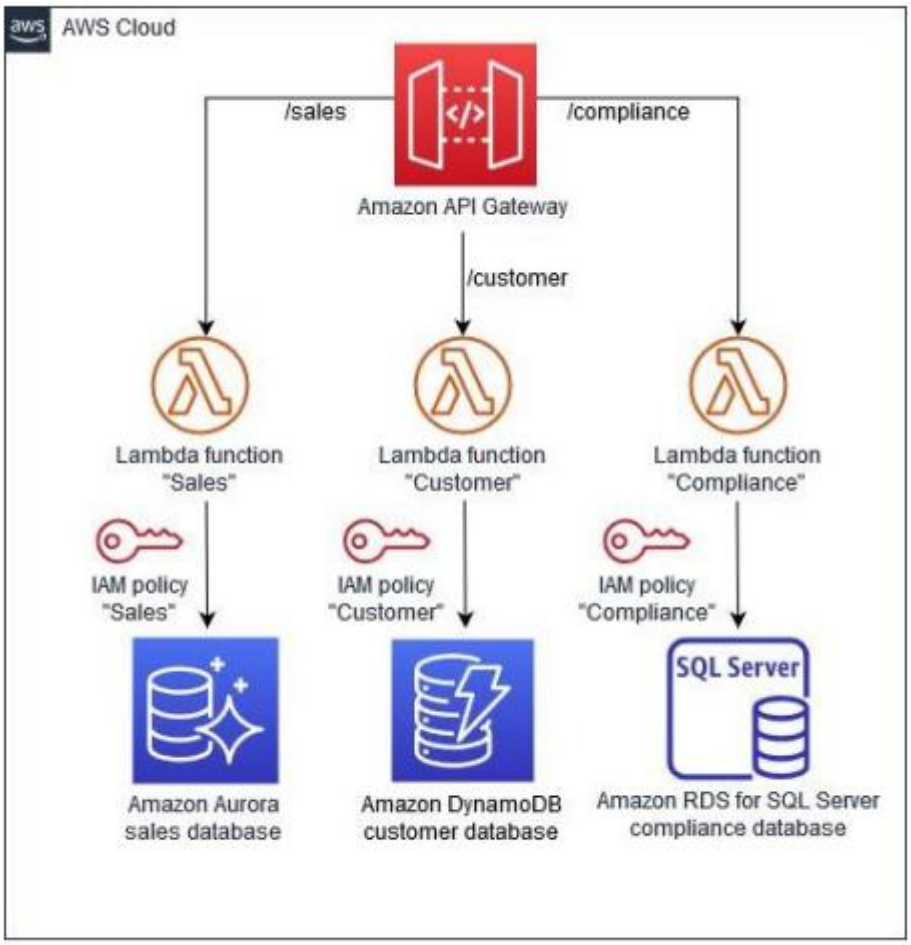

Figure 7.1: Illustration of One-Database-Per-Service Pattern

3. **Develop corresponding microservices.** Ensure the best practices for designing microservices, such as using RESTful APIs, applying the single responsibility principle, ensuring loose coupling and high cohesion, and implementing security and testing protocols. Use containerization technology, like Docker because it provides portability. Create configurations to supply environments required to run each microservice, for example, SDKs, dependencies, database choice and configurations, etc. Amazon Elastic Container Registry (ECR) is a fully managed Docker container registry that makes it easy to store, manage, and deploy Docker container images, and can be used as a private repository to host built-in Docker images. Microservices can optionally be serverless functions, which are not in need of containers, and only run in response to an event, and scale by number of requests. 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.

4. **Decide on method for inter-service communications.** Consider the following factors. If the microservice calls require synchronous communication, direct HTTP calls to the endpoints can be implemented. However, placing the microservice behind a load balancer or using a service discovery model helps to reduce strong couplings, and single point of failure. For communication that is not required to be synchronous, developers can implement a loose coupling model for interaction with the microservice.

    1. A queuing mechanism, such as Amazon Simple Queue Service (SQS), places items into a queue for asynchronous processing by microservices, which also insulates applications from failures in microservices. All services know of a message broker, and they push messages to that broker. Other services can subscribe to the messages that they care about. Microservices can then independently scale based on the number of queued items.

    2. An event publication service such as Amazon Simple Notification Service (SNS) can be used to asynchronously notify many subscribers. Individual services write their events to a message broker. Services can listen to the events of interest. This pattern keeps services loosely coupled because the events don't include payloads.

5. **Migrate contents in the monolith's database to their corresponding microservices.** There is a wide range of options of database technologies on the AWS platform, depending on the characteristics of the data, Relational Databases like Amazon Aurora, NoSQL Databases like MongoDB, Key-Value Databases like Amazon DynamoDB, In-Memory, Time Series and other Databases can be used as database solutions.

6. **Ensure to implement access control for each microservice, for both end-user client access and service-to-service communication.** AWS Identity and Access Management (IAM) manages access to AWS services and resources securely. Depending on the scope of the service, choose Amazon Cognito as identity providers for user-facing services, which integrates with other AWS services like API Gateway, and an Application Load Balancer. Protect service-to-service communications with the technique of mutual TLS (mTLS) which both authenticates inter-service communication and encrypts 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 using a trusted certificate, in order to trust the data contained within the request.

7. **Ensure test coverage for microservices.** Test coverage for the existing codebase helps with code refactoring. Test coverage identifies defects and functional anomalies during the refactoring process. It is important to have test coverage, by dividing the code base into smaller logical units and have test cases for the most critical functional units first, then the remaining functional units. Microservice architecture require more automation as it reduces the effort needed to verify functionalities and discover potential issues. Think of automated CI/CD including tests, logging, and monitoring in advance.
8. **Deploy the microservices on AWS** using Amazon Elastic Container Service (ECS) to orchestrate containers that are insulated and secure.
9. **Route traffic incrementally via API Gateway.** Fargate uses automatic scaling to achieve elasticity and scale resources in or out as the workload requires. When the load breaches the memory utilization threshold of 90 percent, horizontal scaling takes place and launches one more Docker container, attaching it to the load balancer. When the memory utilization threshold is less than 50 percent, it scales in automatically and removes the corresponding resources gracefully from the load balancer.
10. **Decommission monolithic components as their microservice replacements become stable.**

4.2 Migration Phases (Illustrated using the C# service for managing products)

Phase 1: Assessment & Planning

- Identify domain boundaries in the monolith.
    - Product Catalog: manages products and their information.
    - Inventory Management: handles stock and distribution of products.
    - Order Management: manages user order process.
- Define microservices scope and data ownership.
    - Product Catalog:
        - GET /api/products Returns all products.
        - GET /api/products/{id} Returns a specific product.
        - POST /api/products Adds a new product.

        - PUT /api/products/{id}Updates a product.
        - DELETE /api/products/{id}Deletes a product.
    - Inventory Management:
        - GET /api/inventory/{id} Returns inventory of a product.
        - POST /api/inventory/{id} Increase or decrease the inventory of a product.
    - Order Management:
        - GET /api/orders/{id}: Returns an order.
        - GET /api/orders/{userId}: Returns a user's orders.
        - POST /api/orders: Submits an order.
        - PUT /api/orders/{id}: Updates an order.
    - Product Catalog: stores product name, description, and normal price.
    - Inventory Management: stores inventory details for all products.
    - Order Management: stores orders with fields for user ID, order ID, payment details, etc.
- Choose AWS services and design the target architecture.
- Set up AWS accounts and IAM roles.

Phase 2: Infrastructure Setup

- Set up individual repositories of C#.NET projects for Product Catalog, Inventory Management, and Order Management microservices. Structure the projects using best practices, with separate folders for models, controllers, and database access code. Refactor the monolithic application code, moving the relevant functionality for each microservice into its respective project.
- Deploy foundational AWS services (VPC, IAM, networking, security groups, and monitoring).
- Set up container orchestration (ECS with Fargate or EKS for Kubernetes). Amazon ECS integrates with other services that are useful, such as Elastic Load Balancing and AWS Auto Scaling. For large-scale applications with many microservices, Amazon Elastic Kubernetes Service (EKS) provides managed Kubernetes capabilities, such as advanced orchestration capabilities and

managed service aspects.

- Configure CI/CD pipelines. The new microservice codebases need to be added to version control and integrated into the company's CI/CD strategy, automating as much of the process as possible. Amazon Elastic Compute Cloud (Amazon EC2) provides compute capacity in the cloud, and can be used to host Jenkins and JFrog Artifactory as a container for the CI/CD pipeline.

Phase 3: Incremental Migration

- Develop and deploy the first microservices.
    - Write Docker Files for the microservice, specifying the base .NET image, copying the source code, and setting the appropriate entry point.
    - Build and push the Docker images to Amazon Elastic Container Registry (ECR) for the microservice.
    - Create separate task definitions in Amazon ECS for the microservice, specifying the required CPU, memory, environment variables, and the ECR image URLs.
    - Create ECS services for the microservice, associating them with the corresponding task definitions and API Gateway endpoints.
- Implement API Gateway for routing. Apply request validation to ensure incoming requests are well-formed. Options to use additional services for routing are as follows. AWS VPN establishes a secure and private tunnel from the on-premises data center to the AWS global network. Amazon Route 53 is a Domain Name Server (DNS), which routes global traffic to the application using Elastic Load Balancing. Amazon Virtual Private Cloud (Amazon VPC) sets up a logically isolated, virtual network where the application can run securely. Application Load Balancer is a product of Elastic Load Balancing, which load balances HTTP/HTTPS applications and uses layer 7-specific features, like port and URL prefix routing for containerized applications.
- Migrate database components with minimal downtime using AWS DMS. For example, it migrates on-premises Oracle databases to cloud-native Aurora databases. An example for migrating live data to AWS's DynamoDB looks like:
    - First, add a timestamp to the data.
    - Next, create the new tables.
    - Then set up DynamoDB Streams to replicate all writes to the

corresponding tables.

    - Then copy the old data, either with a script or with export / import tools. Make sure this can handle duplicates.
    - Finally, switch over to the new tables.

- Implement logging and monitoring.
- The entire application - the monolith and extracted microservices - needs 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.
- The team 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.

Phase 4: Optimization & Scaling

- Use appropriate tools, for example, AWS App Mesh, or AWS X-Ray to monitor the health status, data and request traffic volume, and security, and respond to issues by troubleshooting microservice interactions promptly.
- Enhance auto-scaling policies for microservices. To prevent different scaling speeds causing issues, when service dependencies exist, decouple the services when we can.
- Optimize database performance. Adjust the size of the ECS tasks and database tables. Check the resource usage of each microservice and databases, and fine-tune them independently.
- Perform security audits and compliance checks. Use the principle of defense in depth. Defense in depth includes securing the perimeter of the application at each layer. 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:
    - Endpoints of services hosted in containers, and APIs that do not require a public-facing interface should be placed in private subnets, disabling direct access from the Internet.
    - Network Access Control Lists (NACLs) should restrict access between

subnets, limiting only the required traffic for the application.

    - Security group rules should allow traffic to each service from the intended clients only.
    - Non-public microservices called 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).

- Logging and Monitoring also help with detecting, understanding, and preventing security issues. Amazon CloudWatch is a common choice 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. Amazon Managed Service for Prometheus offer a Prometheus-compatible monitoring service for containers, and Amazon Managed Grafana can be used to visualize and analyze data from multiple sources. The tool collects and visualizes the metrics and logs of the microservices, and alerts of any issues or anomalies.
- Refactor and optimize as needed. Determine how much capacity each service needs, and optimize for it. Consider that different microservices have different needs. A user-facing microservice might need to scale really fast, at the speed of Fargate or Lambda. A service that only processes asynchronous transactions, such as a payments-processing service, probably doesn't need to scale as fast, and can get away with an Auto Scaling Group. A batch processing service could even use Spot Instances. This way the team may optimize the cost that needs to be spent in operating the services. Also, increased management efforts of multiple AWS service categories matter to the cost.

5. Risks and Mitigation Strategies

With the use of an independent data source for each microservice, data for a single transaction will be in a consistent state only after all the involved microservices have completed their work. Designers of microservice applications need to take special precautions to allow data to exist in a consistent state both between and during transactions.

Risk	Mitigation Strategy
Data inconsistency	Use AWS DMS for seamless data migration
Service communication failures	Implement retries and circuit breakers
Security vulnerabilities	Use IAM, encryption, and security best practices
Operational complexity	Automate monitoring and deployments

6. Conclusion

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.

This migration plan provides a structured approach to breaking down a C# monolith into microservices on AWS. By leveraging AWS's managed services, we ensure scalability, security, and maintainability while minimizing downtime and risk.

Thank you!

References:

1. AWS Inc. Monolithic to Microservice journey for .NET Applications

2. Guille Ojeda, Microservices in AWS: Migrating from a Monolith