Application Modernization (Cloud Optimized Application)

Migrating Applications to the Cloud

Application modernization is a well-known term across organizations, with an appreciation for leveraging hyper-scalers to drive the adoption of this keyword. Applications born in the cloud can be architected as cloud-native from the outset. However, when it comes to brownfield custom applications that are migrated to the cloud, every customer wants to modernize and migrate. The trend is shifting away from the lift-and-shift approach, which is becoming a last resort option.

While the term "application modernization" resonates across teams, there are nuanced interpretations between development and infrastructure perspectives. The infrastructure team typically focuses on migrating applications by leveraging cloud-native services to achieve modernization goals. In contrast, the application development team prioritizes modernizing the application stack itself by adopting the latest frameworks, practices like domain-driven design, and decomposing monolithic applications into microservices architectures. Anticipation of effort or kind of effort that require to remediate might not align with goal of either team. This create unmapped thin gap.

New though arise by business.

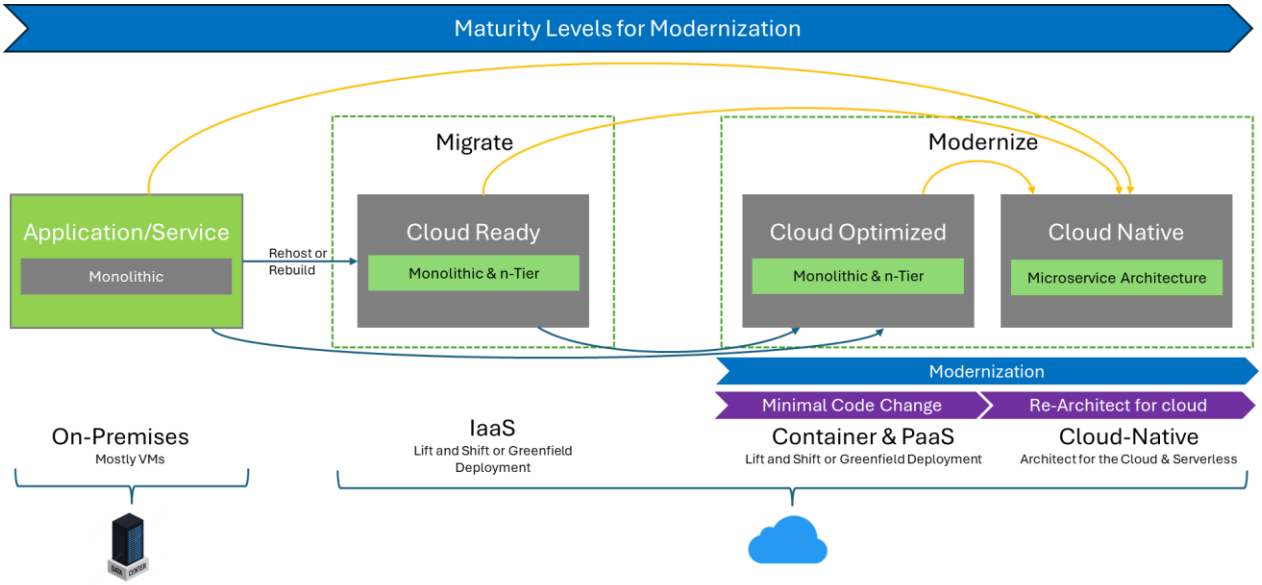
Should Rebuild the application when moving to Cloud?

Utilizing cost/benefit analysis, it's likely that certain individuals may not endorse the initiative. The expenses associated with transitioning to a cloud-native approach would likely surpass the business benefits derived from the application.

Cloud optimization

During my visit to any customer, I tend not to use the term “Application Modernization” when discussing about migrating to the cloud rather I use “Cloud Optimized Migration”. This keyword keeps a clear understanding that the focus is on migrating the application to the cloud with the least changes.

Each application has its own use case but on general use case to summarise its cloud journey I follow the below flow.



Legacy applications, while valuable assets, often present limitations in agility, scalability, and performance, hindering organizations from fully capitalizing on emerging technologies and market opportunities. To overcome modernization challenges by providing access to advanced cloud services and infrastructure. Different applications require tailored migration strategies to optimize performance, cost-effectiveness, and alignment with business objectives.

Key Migration Strategies:

1. Lift-and-Shift (Cloud Infrastructure-Ready) Migration:
    - Ideal for non-critical monolithic applications.
    - Quick migration to cloud-based virtual machines (VMs) without architectural changes.
    - Utilizes Infrastructure as a Service (IaaS) model.

This approach offers a low-complexity, rapid path to the cloud but provides limited benefits in terms of scalability, performance, and cost optimization.

2. Cloud-Optimized Migration:
    - Suited for legacy applications critical to business operations.
    - Enhances deployment with cloud services without altering core architecture.
    - Examples include containerization and deployment to container orchestrators like Azure Kubernetes Services.
    - Enables consumption of cloud-backing services such as databases, message queues, and monitoring.

This strategy represents a mid-point on the modernization continuum, providing incremental benefits while laying the foundation for further cloud-native transformation.

3. Cloud-Native Approach:
    - Best for strategic enterprise applications requiring agility and velocity.
    - Involves re-platforming, rearchitecting, and rewriting code for cloud-native architecture.
    - Enables decomposition into microservices and containerization.

- Offers benefits such as rapid feature releases, fine-grained scalability, improved resiliency, and performance.

This approach represents the most complex and comprehensive modernization effort but yields the greatest long-term business value.

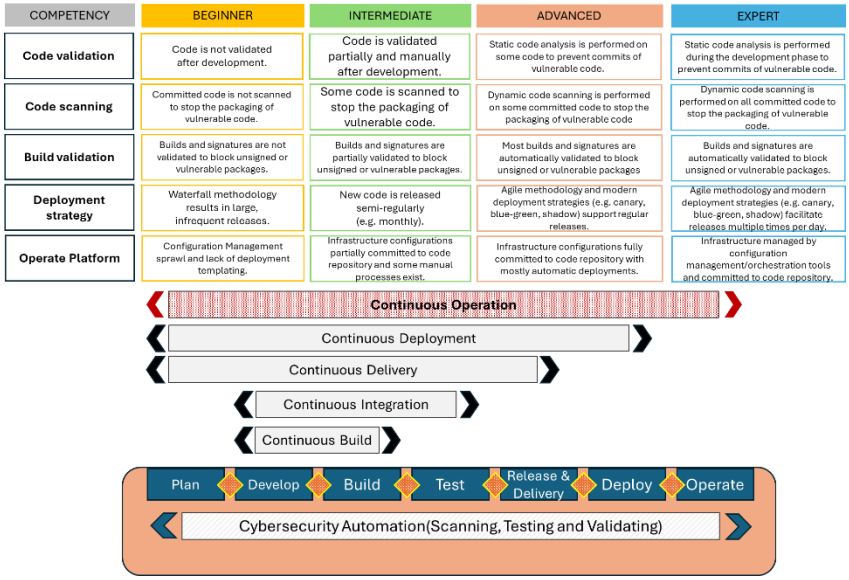
Decision-Making

Considerations for Decision-Making simplified table:

Cloud Infrastructure-Ready Lift and shift	**Cloud-Optimized Modernize**	**Cloud-Native Modernize, rearchitect, and rewrite**
ROI		
Operation Maturity		
Application		
Data		
Complexity		
Challenges		
Advantages		

DevSecOps:

Integrate security practices throughout the modernization lifecycle. Automate security testing and vulnerability scanning in the CI/CD pipeline. Implement security as code to enforce policies and controls. Foster a culture of shared responsibility for security across development, operations, and security teams. Continuously monitor and respond to security threats in the cloud-native environment.



Conclusion:

As organizations embark on the journey of modernizing legacy applications, the choice of migration strategy plays a pivotal role in determining the success of their cloud initiatives. By understanding the strengths and limitations of each approach and aligning them with organizational goals, businesses can unlock the full potential of cloud computing while mitigating risks and maximizing returns on investment.

There are a few points to keep in mind while executing.

1. Customer Confusion on Application Prioritization:
    - Conduct a thorough assessment of the existing application landscape and business priorities.
    - Engage with the customer to understand their key pain points, strategic objectives, and dependencies across different applications.
    - Develop a prioritization framework that considers factors such as business impact, technical complexity, and ROI to guide the modernization sequence.
    - Establish clear communication channels and decision-making processes to address the customer's concerns and reach a consensus on the modernization roadmap.
2. Determining the Starting Point:
    - Analyze the application architecture, dependencies, and technical debt to identify the most appropriate starting point for modernization.
    - Consider factors such as the criticality of the application, the potential for quick wins, and the ability to leverage the modernized components across the application portfolio.
    - Begin with a pilot or a small-scale modernization effort to demonstrate the feasibility and benefits of the approach, building confidence and momentum for the broader initiative.
3. Adherence to Best Practices:
    - Ensure the modernization effort follows well-established design patterns, architectural principles, and industry best practices.
    - Adopt an iterative and incremental approach to modernization, allowing for continuous feedback and course corrections.
    - Emphasize the importance of maintaining application resilience, security, and overall quality throughout the modernization process.
    - Leverage cloud-native and microservices-based architectures to enhance scalability, flexibility, and maintainability.
4. Cost Considerations:
    - Understand that the initial phases of application modernization may involve higher costs due to the investment required in areas such as:
        - Comprehensive assessment and planning
        - Migration and re-platforming efforts
        - Development of new cloud-native components
        - Training and change management
    - Carefully plan the modernization roadmap and budget, considering long-term operational cost savings and business benefits.
    - Explore cost-optimization strategies, such as leveraging cloud-based pricing models, optimizing resource utilization, and implementing automation.
5. Legacy vs. Effort:
    - Carefully evaluate the technical debt and complexity associated with the legacy applications.
    - Assess the effort required to modernize the existing applications versus the potential benefits, such as improved performance, scalability, and maintainability.
    - Consider the opportunity cost of continuing to maintain and support legacy systems versus the investment required to modernize.
    - Develop a strategic plan that balances the need to modernize critical applications with the pragmatic approach of retaining certain legacy components where the effort outweighs the benefits.

By addressing these key considerations during the project execution phase, organizations can navigate the application modernization journey more effectively, ensuring a successful and sustainable transformation.

Use Case Approach

1. Start with understanding.

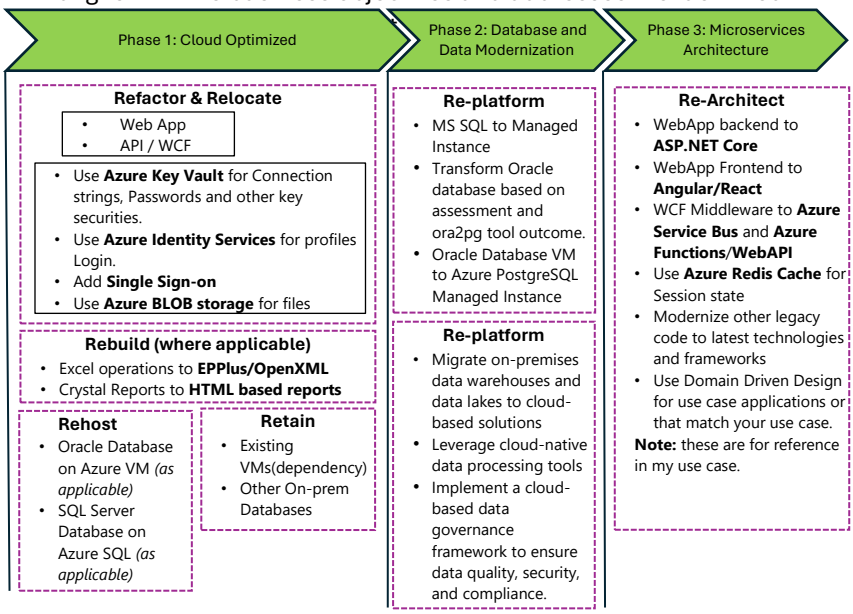
Before delving into an in-depth understanding of the application, it is important to consider a few key points that can help align business and technology objectives:

Business and Technology Alignment | Define Objectives | Inventory and Evaluation

2. Discovery and Assessment

Aligning with the overarching business objectives is crucial when evaluating medium-scale applications. To ensure the right outcome, a comprehensive 3-step assessment process is essential:

- Tool Discovery: Identify and evaluate the relevant tools, technologies, and platforms that can potentially address the application's requirements.
- Collaborative Workshop: Conduct a structured workshop to deep-dive into the application, covering the following key areas:
  - Application Overview and Dependencies: Understand the application's functionality, architecture, and integrations with other systems.
  - Code Review: Assess the quality, maintainability, and technical debt of the codebase.
  - Database and Infrastructure Analysis: Evaluate the database design, performance, and the underlying infrastructure.
- Solution Preparation: Based on the insights gathered from the assessment, prepare a comprehensive solution proposal that aligns with the business objectives and addresses the identified



3. Execution

Customers may struggle to prioritize which applications to modernize first. Start with a pilot project to showcase feasibility and build momentum. Ensure the modernization follows best practices, leveraging cloud-native and microservices architectures.

Expect higher initial costs for assessment, migration, and new development, but plan for long-term operational savings.

Carefully evaluate the technical debt and complexity of legacy applications versus the effort required to modernize. Balance the need to modernize critical systems with retaining certain legacy components where the effort outweighs the benefits.

A structured approach and clear communication can help navigate the application modernization journey effectively.

Application Example

For our example, we will consider one of the real estate developer community portals.

It's a central place for owners and renters to discover and use services offered by developers or particular community

- New Project Notification
- Maintenance
- Event Management
- Bill Payout
- Facility Reservation System
- Complains
- Feedback Mechanism
- Others

Discovery Tools

There are many discovery tools available in the market we have used CloudPilot for code review and Azure Migrate for Dependency and TCO Mapping.

CloudPilot®

Existing Application understanding

Application	Technology Stack

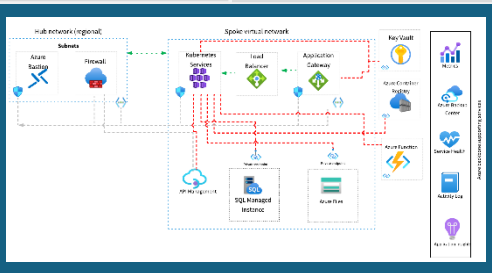
CP(Community Portal)
Community Portal is centralized portal for owners and renters to discover and use services offered by developers or community.
• New Project Notification
• Maintenance
• Event Management
• Bill Payout
• Facility Reservation System
• Others
 	
Url: https:/abcdxyz.com/Login.aspx
 	
Dependent CP Apps:
• Mobile Service
• Payment Gateway
• Offline data
• Sale Service data
 	
Disaster Recovery (DR)
• UAE North for Primary Site
• UAE Central for DR Site
• Database used for DR replication
	• **.NET Framework 4.7.x**
	• **ASP.NET Web Forms**
	• **Telerik UI for ASP.NET AJAX**
	  • Telerik.Web.Design
	  • Telerik.Web.Gui
	  • Telerik.Web.UI
	  • Telerik.Web.UI.Skins
	• **Reporting**
	  • Crystal Reports
	  • Report Definition Language (RDL) for reports
	  • Microsoft.ReportViewer.WebForms
	• **Windows Communication Foundation (WCF)**
	• **Export/Import Formats**
	  • Excel
	  • CSV
	  • ***Diagrams listed under Integrations***
	• **Excel**
	  • EPPlus
	  • ACE OLEDb Provider
	  • JET OLEDB Provider
	  • Office Interop for Excel
	  • NPOI for Excel
	• **Logging**
	  • Elmah
	  • Itextsharp
	• **Data Layer**
	  • ADO.NET
	  • Microsoft Enterprise Library
	  • Microsoft.Practices
	  • Microsoft.Practices.Unity
	  • Microsoft.Practices.EnterpriseLibrary
	  • mssql.DataAccess

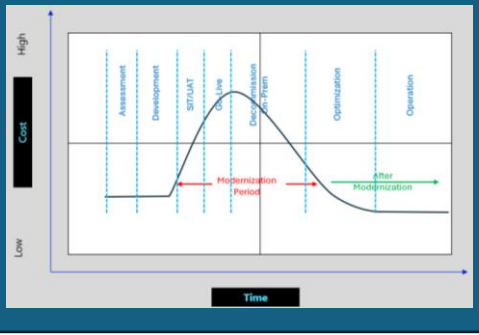
Server	RPO	RTO
Application	12 Hours	3 Hours
Database	1 Hours	3 Hours

Database

Database assessment with DMA(MS SQL) for migrating the database to Native Managed Instance.

Mandatory Changes	
Cloud Readiness Change	**Summary**
SessionState Section *(Mandatory)* *In Proc, Session mode will not support when you scale up your Application.*	1- Use OutProc Session where Session State is stored In the StateServer and SqlServer modes. ***Recommend.*** 2- Azure Redis Cache provides a session state provider to store your session state in a cache rather than in-memory or in a SQL Server database.
Email *(Mandatory)* *Mailing service is being used by your application.*	SendGrid is a cloud-based email service that provides reliable transnational email delivery, scalability, and real-time analytics along with flexible APIs that make custom integration easy. ***Recommend.***
Office Interop *(Mandatory)* *office.interop is not supported on Azure App Service.*	Use EPPlus (https://www.epplussoftware.com/), which is very simple and supports both .xls and .xlsx. ***Recommend.***





Mapping Cloud Native Application to Azure Services (Rearchitecting-Microservices)

The CommunityPortal app was built from the ground up to thrive in the cloud on Microsoft Azure. It uses the modern .NET framework and can run on either Linux or Windows containers, giving you flexibility in deployment. The app itself is broken down into smaller, independent pieces called microservices. Each microservice does its own specific job and can talk to different data storage options, like SQL Server and Redis.

Here's a breakdown of the app's building blocks:

- **Client Apps:** These are the user interfaces, like the mobile app, web app, and single-page application (SPA) that you interact with. They talk to the backend using HTTP requests.
- **API Gateways:** These act as middlemen between the client apps and the microservices. They follow a pattern called "backends for frontends" (BFF), which means it was tailor the data and functionality for each specific client type.
- **Microservices:** These are the independent workhorses of the app. Each one handles a specific piece of business logic and can store its data wherever it needs to, using the choices mentioned earlier.
- **Event Bus:** This is like a central messaging system that allows the microservices to talk to each other.
- **Architectural Patterns:** The microservices use different design patterns to tackle different tasks. CRUD stands for Create, Read, Update, Delete, which is a foundational approach for data management. DDD (Domain-Driven Design) helps structure the code based on the real-world domain of the application. CQRS (Command Query Responsibility Segregation) separates how data is written (commands) from how it's read (queries), which can improve performance and scalability.

By using a cloud-native microservices architecture, CommunityPortal can take full advantage of what Azure offers. This includes things like containers, Kubernetes for managing those containers, and a variety of data storage options. This makes the app scalable, flexible, and able to leverage the power of the cloud.

Container Orchestration and Clustering

The application's services, ranging from ASP.NET Core MVC apps to individual Event Management and Facility Reservation System microservices, can be hosted and managed in Azure Kubernetes Service (AKS). The application can run locally on Docker and Kubernetes, and the same containers can be deployed to staging and production environments hosted in AKS. This deployment process can be automated, as we'll explore in the next section.

AKS offers management services for individual clusters of containers. The application deploys separate containers for each microservice in the AKS cluster. This approach enables each service to scale independently based on its resource requirements. Furthermore, each microservice was deployed independently, with deployments ideally incurring zero system downtime.

API Gateway

In the CommunityPortal application, we have several front-end clients and different back-end services. Each client doesn't just connect to one specific service; there's a mix-and-match scenario. This can make it tricky to write software for the clients to talk to the services securely. Each client would have to figure out this puzzle on its own, leading to a lot of repeated work and many places where updates would be needed as the services change or new rules are put in place.

Azure API Management (APIM) is a tool that helped us publish our APIs (the interfaces that allow our front-end and back-end to communicate) in a way that's organized and easy to manage. APIM has three main parts: the API Gateway, the admin portal (the Azure portal), and the developer portal.

The API Gateway wass like a receptionist for our APIs. It takes requests from the front-end clients and sends them to the right places in our back-end services. It can also do things like check if the request is allowed and change the request to fit what the back-end expects, all without needing to change the code. The Azure portal is where we set up our APIs and decide how they should be used. We can also see reports and set rules, like how many requests someone can make or how the API should change requests before sending them to the back-end. The developer portal is where developers go to learn about our APIs, test them out, and see how they're using them. Developers can also manage their own accounts, like getting a special code to use our APIs.

With APIM, we can group our services in different ways, so each front-end client gets the right set of back-end services. APIM works well for complex setups. For simpler needs, we can use a simpler tool called API Gateway Ocelot. In our CommunityPortal app, we use Ocelot because it's easy to use and can be part of the same setup as the rest of our app. If we're using AKS, another option is to use the Azure Gateway Ingress Controller. This lets us connect our AKS setup to an Azure Application Gateway, which helps balance the traffic going to our AKS services.

Data

In the CommunityPortal application, we use different storage methods for our various back-end services. Some microservices rely on SQL Server databases, while others use Redis or MongoDB for their data storage. Azure provides support for each of these storage formats.

For SQL Server databases, Azure offers a range of products, from single databases to highly scalable SQL Database elastic pools. Each microservice was set up to communicate with its own SQL Server database, making it easy to manage and scale databases according to each service's needs.

The Facility Reservation System microservice, for example, uses a Redis cache to store the user's current facility blocking and payment between requests. During development, this cache can be set up in a container, and in production, we can use Azure Cache for Redis, which is a fully managed service offering high performance and reliability without the need to manage Redis instances or containers ourselves.

Event Bus

The CommunityPortal application uses events to share updates between different services. To achieve this, it employs various implementations. For local operations, RabbitMQ is utilized. However, when hosted in Azure, the application switches to Azure Service Bus for messaging.

Azure Service Bus is a fully managed integration message broker that enables applications and services to communicate reliably and asynchronously. It supports individual queues and topics, which are used for publisher-subscriber scenarios. In the CommunityPortal application, topics in Azure Service Bus are utilized to distribute messages from one microservice to any other microservice that needs to respond to a specific message.

Azure Functions and Logic Apps (Serverless)

In the CommunityPortal sample, there's a feature for tracking online marketing campaigns. For this purpose, an Azure Function is used to track the details of a marketing campaign based on its ID. Instead of building a full microservice, a single Azure Function is considered simpler and sufficient for this task.

Azure Functions offer a straightforward build and deployment process, especially when set up to run in Kubernetes. Deployment of the function is automated using Azure Resource Manager (ARM) templates and the Azure CLI. Since this campaign service is customer-facing and performs a single operation, it is an ideal use case for Azure Functions. Configuring the function is minimal and involves setting up a database connection string and image base URI settings. Azure Functions can be configured easily using the Azure portal.

Azure Key Vault

Azure Key Vault is a managed service designed for securely storing and accessing secrets. Secrets can be anything that requires tight access control, such as API keys, passwords, or certificates, and are organized into logical groups called vaults.

Using Key Vault significantly reduces the risk of accidental leakage of secrets. By utilizing Key Vault, application developers no longer needed to store sensitive information within their application's codebase. For instance, instead of embedding a database connection string directly into the application's code, it was securely stored in Key Vault. Applications can then access this information securely using URIs, which allows them to retrieve specific versions of a secret without the need for custom code to protect this information.

Access to Key Vault is controlled through proper caller authentication and authorization. In a cloud-native microservices architecture, each microservice typically uses a ClientId/ClientSecret combination for authentication. It's crucial to store these credentials outside of source control and set them in the application's environment. For accessing Key Vault directly from Azure Kubernetes Service (AKS), Key Vault FlexVolume can be used.

Reference

Mark Russinovich. Microservices: An application revolution powered by the cloud
https://azure.microsoft.com/blog/microservices-an-application-revolution-powered-by-thecloud/
• Martin Fowler. Microservices
https://www.martinfowler.com/articles/microservices.html
• Martin Fowler. Microservice Prerequisites
https://martinfowler.com/bliki/MicroservicePrerequisites.html
• Jimmy Nilsson. Chunk Cloud Computing
https://www.infoq.com/articles/CCC-Jimmy-Nilsson
• Cesar de la Torre. Containerized Docker Application Lifecycle with Microsoft Platform and Tools (downloadable e-book)
https://aka.ms/dockerlifecycleebook
The Twelve-Factor App. XI. Logs: Treat logs as event streams
https://12factor.net/logs
• Microsoft Diagnostic EventFlow Library GitHub repo.
https://github.com/Azure/diagnostics-eventflow
• What is Azure Diagnostics
https://learn.microsoft.com/azure/azure-diagnostics