Microservices Overview

Whitepaper

Why Move to a Microservices Architecture

Craig Pilkenton

Solving Technology. Together.

Table of Contents

1. Introduction **2**
2. About Hylaine **2**
3. What are Microservices **2**
4. Design Patterns **4**
4.1. API Gateway (Single entry point for all clients) 4
4.2. Event Bus (Pub/sub Mediator channel for asynchronous event-driven communication) 5
4.3. Service Mesh (Sidecar for interservice communication) 5
4.4. Monitoring / Logging 6
5. Development Stacks **7**
6. Key Benefits **8**
6.1. Increased Resilience 8
6.2. Improved Scalability 8
6.3. Technology-Agnostic Development Tools 9
6.4. Decreased Time-to-Market 9
6.5. Improved Return on Investment (ROI) 9
6.6. Continuous Delivery 9
7. Risks & Stumbling Blocks **9**
7.1. Business Risks 9
7.2. Technology Risks 10
7.2.1. DevOps Maturity 10
7.2.2. Design Challenges 10
7.2.3. Security and Complexity 10
7.2.4. Performance 11
8. Conclusion **11**
9. References **11**

Document Version	Date	Notes
1.0	07/01/2019	Craig Pilkenton and Ryan McElroy
1.1	09/18/2019	Updated Risks section and some tech

1. Introduction

Microservices is a modern architectural style and approach to software development where applications are built as a suite of modular services that are kept as small, scalable, and customer-focused endpoints with specific business goals. Being independently deployed and scaled, each service provides a firm boundary of what business case they are solving for. They are also designed to be independently scalable for reuse by many other applications.

The primary goal behind this type of development methodology is so that applications become easier to build and maintain when broken down into smaller, composable components. They can then work together in an application to become the sum of its constituent components. Responsible for highly defined and discrete tasks, these separate modules communicate with each other through universally accessible APIs, running independently of other services and consuming applications. This focus on decoupled services also allows for increased fault tolerance in our applications.

This whitepaper lays out the case for organizations to increase the maturity of their development methodologies from legacy *n*-tier design patterns, also known as monolithic systems, to the more modern microservice architectures. Moving to smaller, focused services within your development groups allows for increased scalability and is considered particularly ideal when supporting a range of platforms such as cloud, web, mobile, and Internet of Things (IoT).

2. About Hylaine

Hylaine is a technology-agnostic software consultancy firm based in Charlotte, NC. Hylaine's consultants are industry veterans who have delivered hundreds of projects across 4 core service lines: Application Development, Quality Assurance, Business Intelligence, and Process Consulting.

3. What are Microservices

A microservice architecture, or simply microservices, is a distinctive process of developing software endpoints that focuses on building single-function modules with defined interfaces and actions. This implementation pattern has grown popular in recent years as large enterprises such as Netflix, eBay, Amazon, Twitter, and PayPal have moved to being more Agile, as well as increasing DevOps practices and continuous testing.

The idea and naming of microservices came in the early 2000's from James Lewis and Martin Fowler, who suggested it as such.

> "*Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*"

While there is no formal definition of the term microservices or a standard model that represents every system based on this architectural style, we can expect most microservice systems to share similar characteristics.

1. Decentralized – In a microservice application, each service usually manages its own unique data store that includes internal data management

2. Non-Complex Routing – these services are smart endpoints that process info to apply logic and generate a response accordingly, with simple pipes through which the information flows

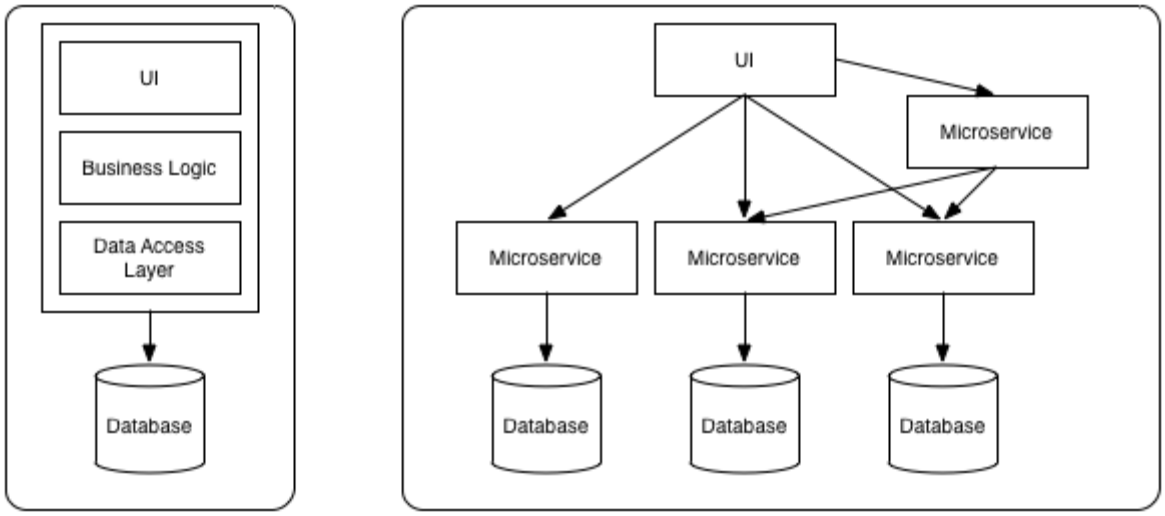
3. Multiple Components – this software can be broken down into multiple component services so that each can be deployed, updated, and then redeployed independently to minimize compromising integrity of any application

4. Failure Resistant or Loosely Coupled – microservices should be designed to cope with failure, as well as their consumers who should implement retry patterns with them

5. Progressive – this architecture is an evolutionary or growing design, and is ideal for systems where you may not fully anticipate the types of devices that may one day be accessing the application

6. Built for Business – these services are usually organized around business capabilities and priorities, where the implementation utilizes cross-functional teams to deploy

Microservice Architecture vs. Monolithic



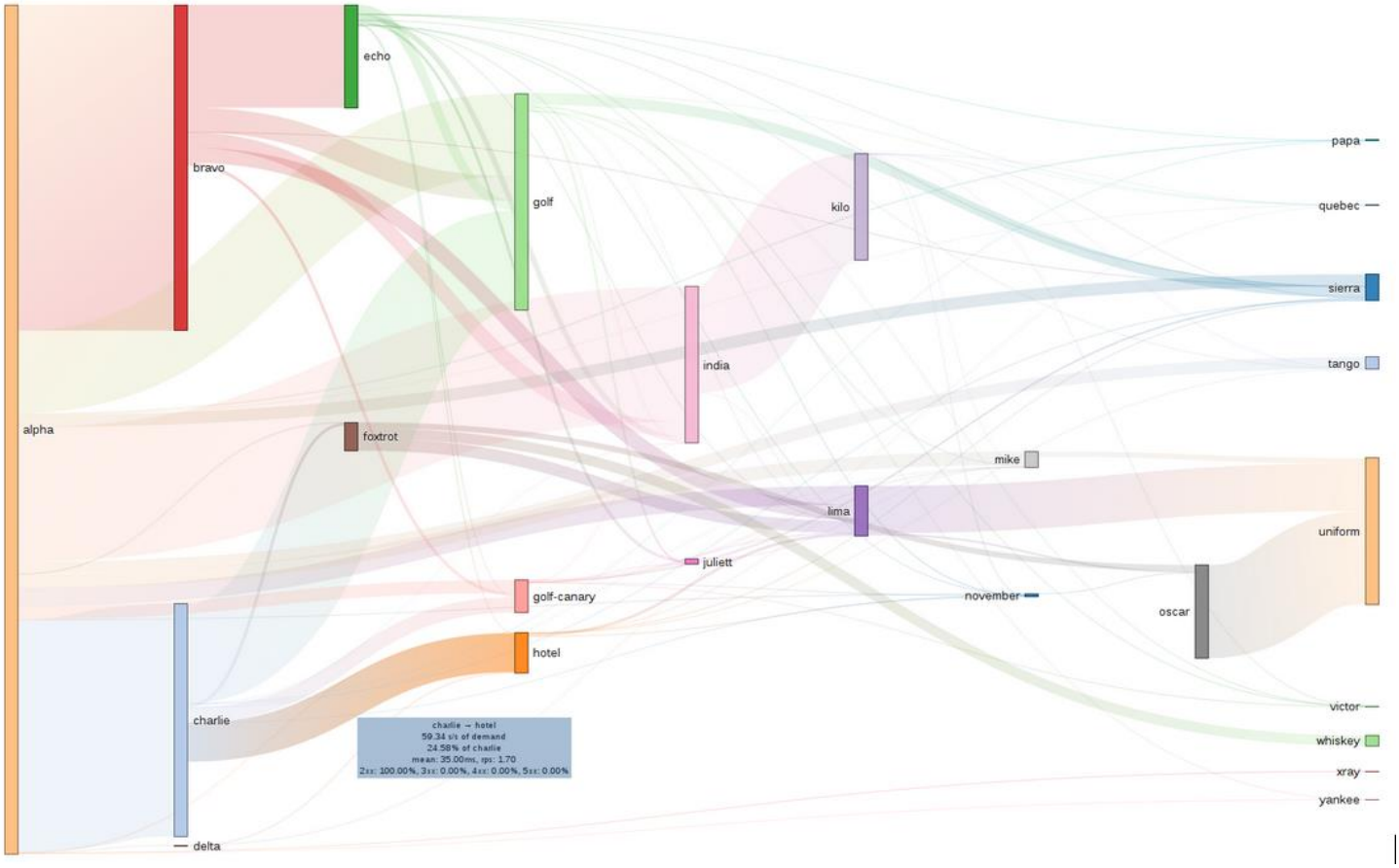
Monolithic Architecture Microservices Architecture

One of the most prominent examples in recent years of moving from monolithic systems to a microservices architecture is Netflix, as mentioned above. Their move was precipitated by several factors and outages that had arisen previously, but was primarily to increase availability, scale, and speed for their growing customers base.

Based on their self-documented journey, a roadmap was created of which sub systems to tackle first as a proof-of-concept, all the way to the end with everything touch point being a microservice. This allowed them to fail fast on non-visible sections and process learnings to mov, e faster as they progressed to more customer-facing services.

Beginning in 2009, they first began moving non-customer facing portions of the system to microservices such as movie encoding. Starting in mid-2010, more customer facing services of the website were migrated to their cloud instance such as account sign up, movie selections, TV selections, metadata, and device configuration. Finally, in late 2011 Netflix moved all websites and services up to the cloud separating their monolith architecture into hundreds of discrete and distinct microservices.

In the Feb 2015 post on the Netflix Tech Blog, A Microscope on Microservices, the authors discuss the handling of massive instance counts as well as providing quick, actionable insight against their large-scale, microservice-based architecture. The following graphic visualizes the relationship seen during that analysis of their microservices between system endpoints to see which are called and how much time is spent in each.



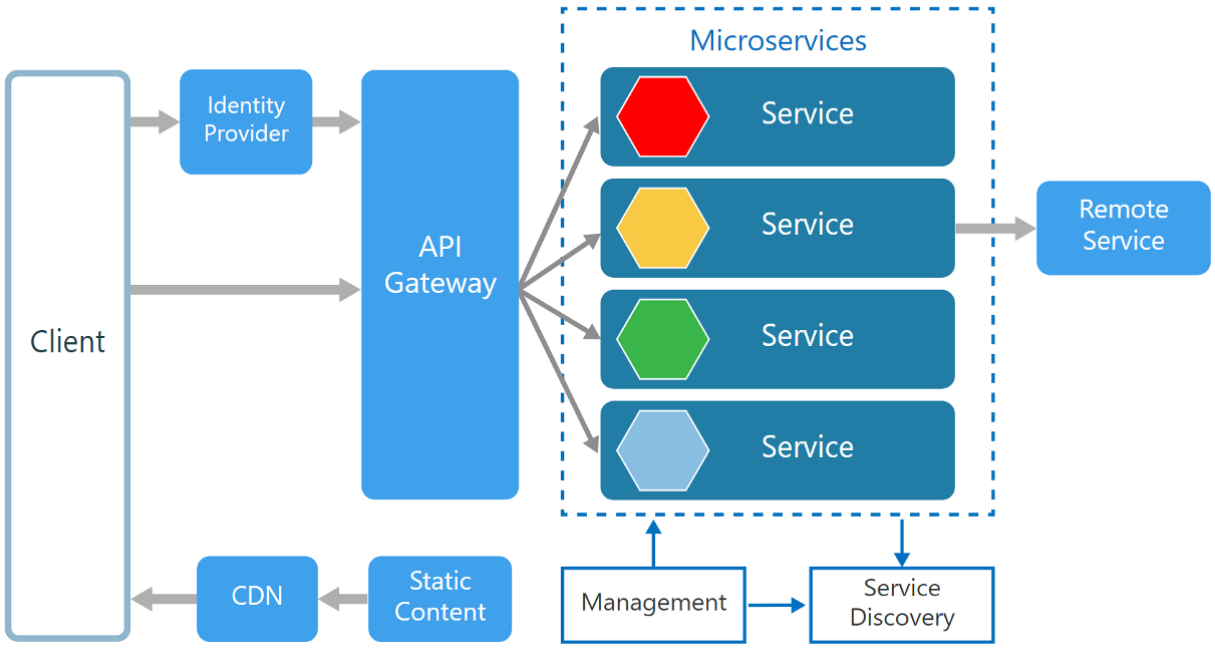
Analyzing their microservice usage and consumption allows them to monitor for bottlenecks and determine opportunities for splitting out to new smaller microservices. This analysis also allows them to plan for future growth and map API interactions as new features are needed.

4. Design Patterns

While there are several design patterns that can be implemented around a microservice architecture, including variations and mixing, there are several common patterns that provide the most success when implemented.

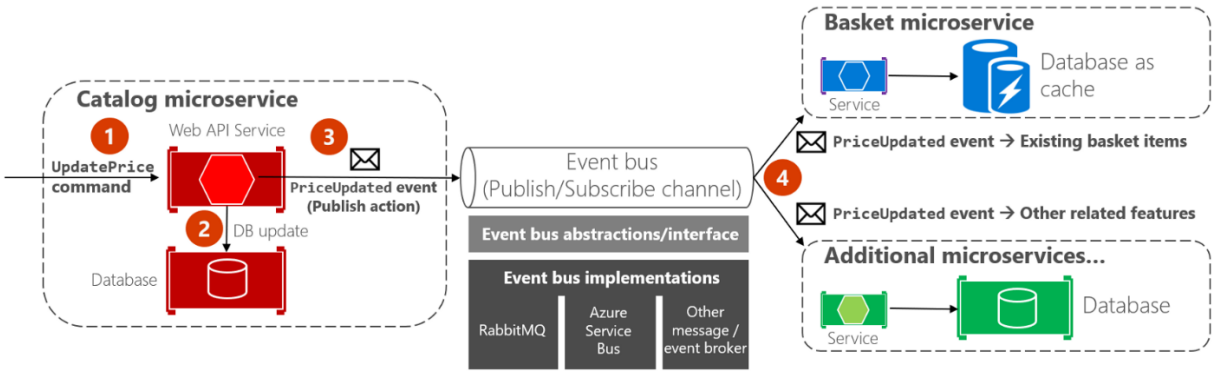
4.1. API Gateway (Single entry point for all clients)

The API Gateway pattern is the most common pattern implemented, being an intermediary with minimal routing capabilities and acting as a 'dumb pipe' with no business logic inside. Using this pattern allows you to simply consume a managed API over REST/HTTP. In conjunction with API Gateway, there are other types of microservices integration patterns such as Point-to-point style that invokes services directly from a client-side app, as well as Message Broker style that implements asynchronous messaging. A core tenant of the API Gateway pattern is that it should always be a highly available and performant component, since it is the entry point to the entire system.



4.2. Event Bus (Pub/sub Mediator channel for asynchronous event-driven communication)

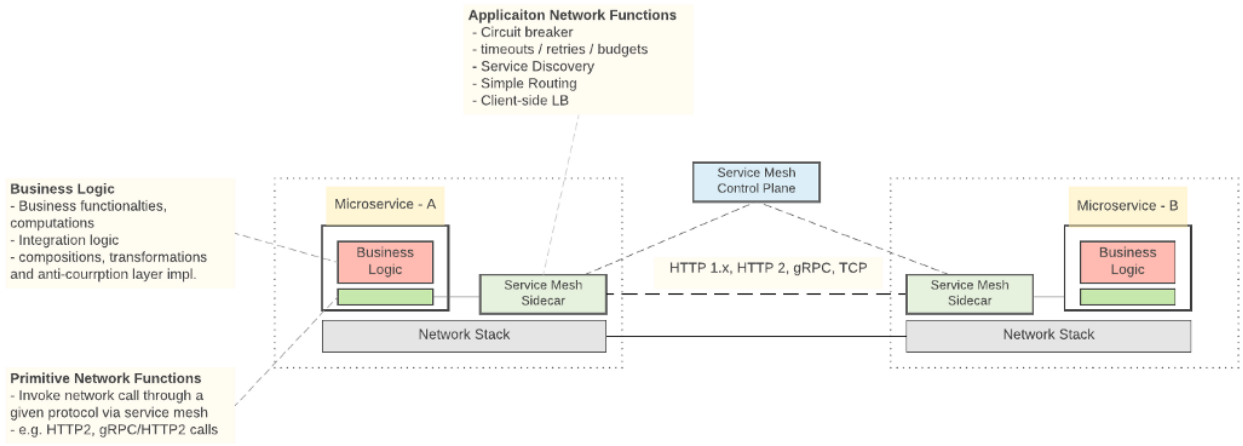
An Event Bus pattern implementation can be used for different parts of an application to communicate with each other irrespective of sequence of messages so as to be asynchronous, and to be language agnostic. Most event bus systems support publish/subscribe, distributed, point to point, and request-response messaging. Some Event Bus pr0ducts even allow the client side to communicate with corresponding server nodes using the same event bus.



4.3. Service Mesh (Sidecar for interservice communication)

A Service Mesh implements the Sidecar pattern by providing helper infrastructure for interservice communication. It usually includes features such as fault tolerance along with load balancing, service discovery, routing, observability, security, access control, communication protocol support, among many others. In common practice, an instance is deployed alongside each service usually in the same container. They are only allowed to communicate through simple network functions of the service. A control plane of a Service Mesh is separately organized to provide essential capabilities such as service discovery, access control, and observability. The most important aspect of the Service

Mesh style is that allows developers to decouple network communication functions from microservice code so that services focus only on the business capabilities developed.



4.4. Monitoring / Logging

The last design pattern for your organizations microservices architecture is how to monitor them for performance and issue resolution. It is critical to see exactly what is going on in an application at any given time which is even more important in a highly distributed topology. Depending on choices made in the Development Stack section, there may be native toolsets that can be directly leveraged for reporting, or an external solution may need implementation to support the deployed architecture.

This a critical piece of the control systems of your microservices and as the number of endpoints increases or become more complex, the harder it is to understand performance and troubleshoot problems. The following principles of monitoring microservices can help ensure that issues can be addressed quickly or before your end users notice their impact.

A. Monitor and log against all endpoints beyond binary up/down checks, such as returned content and latency over time
B. Map monitoring to the business capabilities deployed
C. If using Container development:
    a. Monitor the containers and what is inside of them
    b. Alert on service performance, not container performance

If these principles are leveraged, along with your organization's specific ones, it will allow your teams to establish more effective monitoring as microservices are deployed. These principles allow your organization to address both the technological changes associated with a microservices architecture, in addition to the organizational changes related to them.

5. Development Stacks

When teams are ready to begin building microservice applications, there are many available frameworks, languages, and cloud providers to decide upon at the enterprise level. A primary decision driver will be determining the programming language that the team will be developing in such as C# (.NET Core), Java, and/or Node.js which are some of the most prominent and popular languages today. While each microservice should rely on language-agnostic interchange approaches, standardizing on a primary development language within the team will make it easier for all team members to support and enhance different microservices.

Another decision driver is around the technology stack your microservices will be hosted on, either internally or in a cloud provider, and what underlying framework and development stack can be utilized there.

Once an organization has settled on infrastructure decisions for solutions, the next phase is to determine the development stack to build your microservices in, driven from the organization's programming language of choice.

One critical decision a team must make in this vein is whether to deploy microservices as containers, serverless, or from a hosted framework. Each stack has its own set of pros and cons that need to be considered when deciding on which stack fits the organization best.

Containers – lightweight alternatives to virtual machines where the services and operating system are layered into a contained object that is deployed to infrastructure using orchestration. May be part of a framework decision

- Pros:
    - Small instances of server resources,
    - Operating system and packages easier to maintain.
    - Easier to scale horizontally or vertically during high demand compared to traditional hosting models.
- Cons:
    - Usage not ubiquitous yet in corporate IT,
    - Requires specialized training and knowledge
    - Requires Orchestration solutions to manage

Serverless – microservices that are deployed in a vendor's infrastructure and only run when they are called by consuming applications, essentially renting resources only when they are needed

- Pros:
    - Good for building reliable and scalable infrastructure to serve geographically dispersed users.
    - For medium to low or highly variable workloads, *significant* cost savings over other hosting models
    - Low-effort deployment and development models that continues to mature
- Cons:
    - May have limited visibility or control of framework versions.
    - Serverless technology has "hard" limitations in regard to service states and available resources.

Hosted Framework – a set of libraries that run on a self-managed server for building a distributed system of microservices

- Pros:
    - Good for those who have high capacities and resources to build entire infrastructure
    - With high quality resource planning and utilization, most cost-effective model
    - Easiest model to use for teams accustomed to monolithic architecture
- Cons:
    - Teams must patch and maintain their infrastructure in many instances
    - Difficult to scale horizontally or vertically during high demand

Once a team has determined its primary drivers, the final stage is to begin using a development stack that supports those choices. While a microservices architecture doesn't require specific toolsets for successful implementation, choosing the right one will empower your developers and ensure a shorter time-to-market for your endpoints.

If the organization is focusing on open source frameworks running on internal or cloud infrastructure and includes using Java for the development language, then looking at Vert.x, or SpringBoot/Cloud, or Dropwizard may handle many of the features needed for proper microservices.

If focusing on serverless, internal, or cloud infrastructure capabilities that include the .NET Core or Node.js languages, then Microsoft's Service Fabric framework, or Docker containers can support this type of development.

If the organization would like to take advantage of Platform-as-a-Service options (PaaS), such as Microsoft Azure or Amazon Web Services (AWS), the solutions needs may be met this way if the frameworks features are enough.

6. Key Benefits

No matter what development technology or cloud provider used to support these types of development, microservices offers increased modularity that make applications easier to develop, test, and deploy as well as enhance and maintain. It can empower companies to leave behind large, slow-moving teams with complex deployments and shift to more agile development delivering on their own cadences.

The following subsections will cover many of the core benefits that can be realized from this modern approach and how they benefit organizations.

6.1. Increased Resilience

With the use of microservice patterns, entire applications are decentralized and decoupled into independent services that act as separate entities. This separation ensures that failures in one service or function causes minimal impact to the rest of the application, allowing the consuming application to continue or fail gracefully as they are just decoupled data sources.

6.2. Improved Scalability

As each service is a separate component that makes up the whole, a single function or service can be scaled up or out without having to scale the consuming application. This not only allows for independent interactions, but these microservices can be reused by other consumers that may have differing consumption rates.

6.3. Technology-Agnostic Development Tools

Creating microservices allows us to have flexibility in using the right tools and programming languages for the right task, simply adhering to common data conventions and API inputs / outputs. Each service may use its own development language, supporting data sources, or other microservices while still being able to communicate easily with other endpoints in your application. This approach allows multiple development teams to work in their style and language without forcing a one-size-fits-all mindset.

6.4. Decreased Time-to-Market

Microservices are intentionally designed as targeted and focused endpoints of functionality, implemented in a loosely coupled manner within their consuming applications. This allows for development teams to implement smaller increments of functionality that are independently testable and deployable while allowing for applications and functionality to move to market quicker.

6.5. Improved Return on Investment (ROI)

With a microservices architecture, we can optimize resources and independent development teams by allowing them to deploy more quickly and pivot functionality direction more easily when needed. Development of new features can be reduced as teams can iterate more quickly over a service's functionality sets . The amplified effectiveness of microservices architectures not only can reduce infrastructure costs, it can also minimize downtime when combined with resilient deployment strategies.

6.6. Continuous Delivery

Moving to a microservices architecture pattern allows for cross-functional teams to handle the entire lifecycle of a service, reducing time-to-market of feature sets versus monolithic architectures in which all features were deployed together. This move allows for improved delivery as teams work more closely together and must be fully invested in all the pieces working together. With this incremental development approach, all microservices code and underlying infrastructure is continuously developed, tested, and deployed.

While Continuous Delivery and Microservices are technically separate concepts, they are extremely synergistic and intertwined.

7. Risks & Stumbling Blocks

Continuing to utilize aging or monolith architectures can make growth beyond a certain scale difficult or impossible. In turn, this impacts business and user experience. Businesses that have embraced microservice architectures can realize significant benefits. While microservices and their benefits are promising, not every business can capitalize on the architecture immediately due to organizational or technology impediments.

The following subsections cover several most pressing concerns when moving to this type of modern architecture. While not insurmountable, organizations must focus time and effort eliminating roadblocks and constraints in the development and support lifecycles.

7.1. Business Risks

Moving to microservices architectures can be quite powerful but is also challenging for large enterprises to adopt. This solution is not just a technology implementation, but rather an entirely new working model that can impact

virtually all aspects of business operations. Typically, if a large enterprise is going to implement microservices well, it will need to first transform the heart of its organizational culture to smaller, more agile teams, which can be at odds in the standard organizational inertia seen within larger enterprises.

7.2. Technology Risks

Utilizing incremental development and continuous delivery methods, microservices will keep an organizations infrastructure teams constantly learning and changing as they adapt to developers needs for resources in a timely manner.

7.2.1. DevOps Maturity

Business also need to incorporate agile devops practices as everyone can be responsible for service provisioning, and failures. To ensure an organization knows how services are performing over time and immediately react when something does go wrong, robust monitoring with alerts needs to be in place to effectively monitor and manage the entire infrastructure. In order to support true microservices, an organization's DevOps practice must:

- Have well defined code integration and automated testing practices
- Fully automate deployment to production and lower environments
- Be mature enough to rapidly develop new build, test, and deployment products for solutions

7.2.2. Design Challenges

Modifying existing software or creating new solutions in a Microservice Architecture can present unique difficulties. The most critical part of the design process is finding correct logical boundaries for separation of concerns. If two business concepts are separated into different microservices but cannot exist without one another, a solution will introduce additional complexity and performance hits for no appreciable gains.

Key questions to ask when attempting to split functionality or concepts into microservices include:

1. Is this concept able to exist independently and stand on its own? Would most of the potential changes or evolutions in this proposed microservice be accomplishable independently?
2. Is the functionality able to be shared and utilized by other services/microservices?
3. Is the concept large and important enough to justify the overhead of a Microservice implementation and infrastructure (Repositories, Pipelines, Lifecycle Management, Computing Resources)?
4. Can the solution or dependencies absorb the performance implications of microservice extraction?

7.2.3. Security and Complexity

The implementation of distributed systems can be complex, sometimes directly correlated to the number of services involved. Microservices require excellent documentation of endpoints and failed connectivity / retry patterns are inherently built into every design. Designers, implementers, and maintainers of systems must be able to understand a solution that may be made up of multiple microservices.

With inter-service or cross-domain communication that may come from consuming other microservices, the attack surface of an application or solution increases. This may require enhanced security measures to secure data and

communication. Organizations should ensure that security approaches have a default implementation and configuration when building multiple microservices at a time. Large Cloud Service Providers such as AWS and Azure allow the templating of security features and resources to make sure that applications are secure from the start.

7.2.4. Performance

When a solution utilizes microservices, one oft-overlooked implication is performance. When services do not reside in the same process, they must communicate through other means. Transmitting an HTTP message, for instance, can take multiple orders of magnitude longer than communicating between logical layers of the same application. Other communication patterns and techniques are more performant. Examples include Service Remoting in Microsoft's Service Fabric technology. However, these can dilute or eliminate a one of the key benefits of Microservices: decoupling logically separate concerns.

In addition, seemingly small decisions such as chosen interchange formats can have significant implications. For instance, a detailed XML message can be much larger than an identical message in JSON.

8. Conclusion

While a microservices architecture may not be the solution for every problem or application that is needed, the information laid out above shows that it should always be a consideration when developing new solutions for your internal or external customers. The ability for components to be more fault-tolerant, elastic, and serve broader audiences cannot be ignored.is a significant potential benefit that cannot be ignored. There are many case studies from major Internet companies such as Netflix, PayPal, and Amazon that showcase the architecture and methodology used to implement microservices to solve targeted issues.

9. References and Additional Reading

Microsoft – design patterns for microservices

- https://docs.microsoft.com/en-us/azure/architecture/microservices/design/patterns

Microsoft – Application Architecture with microservices

- https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

MicroServices.io – Microservice Architecture Patterns

- https://microservices.io/patterns/microservices.html

SmartBear Software – Why You Can't Talk About Microservices Without Mentioning Netflix

- https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/

Netflix Tech Blog – A Microscope on Microservices

- https://medium.com/netflix-techblog/a-microscope-on-microservices-923b906103f4

MartinFowler.com – Microservices Resource Guide

https://www.martinfowler.com/microservices/?source=post_page----------