WHITE PAPER

THE TRANSFORMATIVE BENEFITS OF MICROSERVICE ARCHITECTURE

Application Architecture using a microservice model has been the foundation for large enterprise applications for more than a decade. This model has proven itself in the data center, and has now successfully been adapted to cloud applications.

What lessons have been gained about the benefits and caveats of microservices? How can they be incorporated into applications still tied to large, monolithic structures? This paper will present a high overview of what microservices do and how to create or transform applications to benefit from them.

Ramona Maxwell
Application Architect
May 2019

Magenic® Fast Forward

Introduction

Monolithic architectures are common in enterprise applications, but may suffer from some anachronistic limitations in terms of both scalability and modernization. Yet they are still a standard implementation in many organizations, even for new applications, as they have certain advantages in terms of supportability – particularly in organizations with internal data centers.

The problem to be solved

Monolithic applications' size and complexity generally preclude just spinning up another instance when an application needs to quickly scale. Additionally, either due to organic growth or poor design, complex interdependencies may develop. The potential for failure of one cog in the machinery taking down the whole ship thus makes rapid deployment of new features an unacceptable risk. Proper separation of concerns within the application may be lacking not only functionally, but also extend to the architectural design so that domain boundaries are muddled.

Microservices Enter the Picture

Service-oriented architecture, and its darling child microservices, began to establish a new normal for application architecture in response to the need to keep applications current and secure, as well as to provide for unexpected loads. Their transformative effect was partly cultural, as software development teams shifted to a new paradigm in order to achieve the continuous delivery model that underpins a successful microservice implementation. A microservice approach presented its own challenges, particularly in supporting additional request and response traffic and managing a more complex application structure. The term microservice is relative in size to the monolith as a whole, as a single microservice may be quite large in a typical enterprise application.

How microservices help

Ideally microservices will address some of the most limiting characteristics of a monolith, bringing substantial benefits that are worth the price tag microservices carry in terms of complexity. A failure of a dependency within a monolithic application may trigger the failure of the application as a whole, particularly if that application has grown massive through endless rounds of feature stapling. In contrast, a proper microservice should fail without the Jenga effect of collapsing the tower. Growth or spurts of heavy application usage can also be managed since the portions of the application that need to scale can do so independently.

Microservice Implementation Benefits

Applications that scale

Microservices are inherently scalable, in that multiple instances can be instantiated in response to demand. Additionally, microservices can be written to support multiple applications, thus creating a standardized API for both existing and new applications utilizing a service. Microservices can also provide for failover in case a particular endpoint becomes inaccessible, thereby allowing an application to remain healthy despite the demise of one of its components.

Application lifecycle

A healthy application lifecycle allows for continuous delivery of new features, keeping the application relevant in terms of its capabilities while preserving stability. An advantage of microservices in this regard is that they can be deployed independently of the application consuming them. In fact, whether a service can be upgraded or deployed without requiring changes to other system components is a decisive marker as to whether it is truly a microservice.

Microservices can be updated without stopping an entire application, and when a new version of a service is released it can also be distinctly evaluated. For instance, a service providing data access may be found to have an exceptionally slow query that must be re-worked to keep the application performant. Once the query is fixed and tested, it can then be deployed in a blue/green fashion to the application in order to verify that calls to the updated service are providing the hoped-for improvement.

Beyond upgrades, microservices allow wholesale conversions to new technologies that might benefit a specific component. As long as a service can return expected values, it is indifferent to the application itself how that is accomplished. Thus, the ability of a microservice architecture to integrate various LOB systems is a key value-add. It also allows a development team to avoid the temptation of recycling legacy code from the monolith, and to adopt a fresh approach geared toward modernization and upgraded performance.

Transformation considerations

A key tenant of microservice implementation is that it thrives within a DevOps culture. Small, agile teams are another benefit of the microservice approach. Development cycles that may have taken months become shorter and shorter, until continuous integration of new code becomes a reality.

Agile does not equal unstructured, as teams also need to have a big-picture view of the domain problem they are trying to solve. Carving a piece of functionality off of the monolith could create more complexity if the core monolithic application is left running an alternate to the new microservice, thus the aim should be to replace the monolithic service with the new microservice. A scattershot approach to implementing microservices will not return the benefit that can be gained by properly targeting the right service to transform. Architects and developers will also have to exercise caution so that the original problems of the monolith don't leak into the new microservice layer, either through dependencies on the monolith or coding practices that tie to specific infrastructure or technologies.

Monitoring microservices for security and health is also key to supporting a microservice dependent application. If a microservice goes down it needs to fail fast and gracefully. For instance, a microservice managing a list of recent purchases for a shopping application might temporarily be unable to load but that should not prevent completing a transaction to buy items already in the cart. Recovery should also be part of the design so that if a microservice goes down its replacement can step in without manual intervention.

ROI from a microservice approach
People who can administer enterprise-scale legacy applications are rare. Application defects are sometimes simply tolerated, either because no one knows exactly how they could be fixed — or the effort to do so exceeds available development resources. Acquiring the skills of an individual or team who can troubleshoot a single microservice is eminently more possible. Additionally, if an entire application goes down the monetary cost of an outage can be severe. So, although conversion may be an expensive investment, the returns may be significant.

The ability of a microservice architecture to immediately scale core business applications is an enabler of organizational growth. Acquisitions, new lines of business or geographical expansion can all place sudden pressure on an application to be more than it was built to be. When companies like Etsy, Lyft, Amazon or Netflix tout the benefits of microservice architecture as part of their path to operational maturity it's hard to argue with that kind of success.

The additional complexity of managing and integrating microservices must be factored into any analysis of benefits and costs. A successful microservice implementation requires deep domain knowledge to set the service boundaries, capable teams and a switch in thinking to allow those teams to execute independently while trusting that – like the data they're entrusted with – they'll reach eventual consistency. For a monolithic application that meets its objective without failure or scalability issues, deconstruction into microservices is not a goal in and of itself.

Microservice Architecture in the Cloud

Applications that are cloud-native use microservices as a given, since paradigms for data center applications cannot successfully be adapted to a cloud environment. This does not obviate the need to design a microservice architecture.

Backing Services

Every piece of functionality a cloud-native application relies upon, from authorization and authentication to databases, logging and more, is a service to be called based on configuration information citing its location (regardless of whether the backing service is local or cloud-based). Assuring the application is designed to call a service effectively over the network in scenarios where traditional co-location strategies no longer apply is important, as will be selecting the right backing service for a particular purpose when the choices are very broad.

Processes

A microservice running in the cloud will not be dependent on a specific process being kept alive in order for it to operate, unlike an on-premise monolith that may crash if a process dependency is not met. A new web or worker process is simply instantiated to replace the one that fell down and the application continues to run. In other words, “give me one of those” not “give me that one.”

Containers

Building microservices in containers is not only a great strategy for the cloud, it brings some cloud-like advantages to on-premise data centers. Using Kubernetes for example, allows multiple containers to live within a single IP addressable pod. Containerized services inside the pod can communicate via localhost, thus eliminating much network traffic drag.

FAAS

Utility computing via Functions as a Service means that portions of an application may not even need a microservice of its own; the application can simply process its query or calculation and consume the answer. The challenge to this approach is that a function will not preserve the result it returns while the requestor performs operations on it; that result must immediately be stored somewhere else – an operation that could take time. Until network traffic clears and the result is deposited in storage, the next process in the application which depends on reading the function result is delayed – a delay that could be magnified by the slow storage available in the cloud or the network traffic in the tenancy.

In an on-prem system you could co-locate data with the application servers processing requests for it. In the cloud there are no guarantees as to geography for network communications or longevity of caches. Much like container management issues giving rise to technologies like Kubernetes these latency problems have inspired workarounds, such as Microsoft's Orleans framework for task management. Eventually it is hoped that cloud providers will also make improvements to their implementation that facilitate adjacent data processing.

Remaining stateless
Since microservices running in the cloud are designed to be stateless (storing state information in the database, on the client or anywhere but in the service itself) they can be self-healing. If a cloud-based microservice doesn't register a heartbeat with a monitoring service, and new instance of the service can be provisioned immediately to restore the health of the application. This provides application durability through various anomaly conditions, without the need to perform a root cause analysis unless there are repeated failures,

Use Cases

Amazon
Amazon suffers from a little debate in whether they're actually building microservices, or what they refer to as (and may have coined the phrase for) Service Oriented Architecture. What Amazon refers to as a service might be analogous to a pod in Kubernetes – it keeps one group of interdependent operations together. Within that service may be a database, a web interface or any other required component (i.e. microservices) but that service will perform one job well – whether that is tracking the number of items in your cart or the stock level on a particular item. Services at Amazon are insular, meaning applications may not talk to any internal function that is not exposed through the services interface.

Regardless of how one might nitpick over Amazon's terminology, its growth has been such that as of early 2018 they were the third largest company in the world and had built enough infrastructure to support not only its own sales enterprise, but that of many other companies. Amazon was inspired to make change happen, not by a Netflix style delivery failure, but simply because it couldn't grow its existing monolith any further.

Amazon also has a noteworthy approach to support. Development teams are bound to a service they own and they are responsible for its operation as well. Since a single team supports what it builds, it becomes much easier to isolate performance bottlenecks.

Lyft

Although the concept of ride-sharing is modern, the application stack Lyft started out with was not. The company's rapid growth was constrained by its dependence on an Apache/PHP monolithic application, which limited the amount of connections that could be made to the application at one time. Attempts to remediate by stapling on additional tools were increasing complexity and reducing the ability to trace the inevitable errors that sprouted as the stack grew. Further parts of the application were already migrating to the cloud, increasing the app surface considerably.

In order to take on the challenge of moving a business-critical monolithic application to a new service-oriented architecture Lyft created an open source HTTP proxy layer, Envoy, which is now in use at a laundry list of enterprises from Microsoft and Google to Airbnb and eBay. This service proxy layer handles every bit of the application's internal communication allowing it to be agnostic as to which application language a service is written in, what type of database is being connected to, and so on. This in turn facilitated transformation of services that could be peeled off the monolith and implemented as a microservice (a process that is still ongoing at Lyft). Another benefit was the standardization of monitoring data emitted by the application. The rideshare giant's 24B valuation at its IPO in March of 2019 was facilitated in part by the massive effort Lyft has made to move to microservices.

Netflix

Netflix, from a large-scale outage of its on-premise systems in 2008 to present, has moved its video streaming operations entirely onto Amazon Web Services. To illustrate how this move expanded the capabilities of its service, in just one year in 2016 it expanded into 130 new countries. It is noteworthy that Netflix took a cloud-native approach to migration, not recreating existing data structures in the cloud. They also adopted a microservices approach to application development and provisioning of internal systems. This Matryoshka doll approach to wrapping every layer of functionality in another that automates it is something we see flowering industrywide. For instance, Kubernetes automating Docker, and then in turn, Kubernetes being streamlined by Pivotal Container Services.

Planning for Microservices

Architectural considerations

A microservice transformation project is also an opportunity to review whether the health of the existing application aligns to principles of Domain Driven Design vs. simply being divided into sections reflecting its technology layers (such as networking, data storage and interface).

Domain Driven Design refers to a particular business unit's use case as its Bounded Context, i.e. a marketing department may utilize a subset of application features that track customer contact, while a manufacturing arm is managing its sales. Even though these two departments' needs appear to intersect, providing for them should not overlap within the application so that a query within marketing could not fail due to a temporary failure of the sales module.

Prior to implementing the first microservice its design implications should be well understood, and a complete architectural assessment should be performed before undertaking an application's overall transformation to a microservice model.

Timing

When should consideration be given to implementing a microservice architecture? Ideally an organization will consider moving toward a microservice architecture before application failures due to embedded components make it mandatory. Similarly, a greenfield implementation may be ideal but few organizations have that scenario when contemplating microservices. Still, there may be low-hanging fruit that is a good target for initial optimization, after which the gains of optimization can continue to be repaid into the existing technical debt.

Even when a full microservice implementation is not undertaken, aspects of a microservice approach can still address monolithic obstacles. For instance, a key characteristic of many monolithic applications is that the data pipeline terminates at a single endpoint. The craft site, Etsy, surmounted significant performance concerns by implementing microservice-like meta endpoints to backing services in order to facilitate concurrent requests to those services.

Summary

Is monolith a bad word?

A monolithic architecture will always retain certain advantages over a distributed microservice pattern, including simplified deployment and testing. The monolith of today might have been streamlined and performant at its creation, but as demands and complexity increased it became bulky and brittle. There is a standing debate among software architects as to whether every application should start the application lifecycle as a monolith, and only be carved into microservices when it becomes unwieldy. This approach is problematic, in that a service-oriented architecture may be much more difficult to extract when the single structure begins to lock in functional interdependencies.

Do Microservices Win?

Will an application built around microservices always be superior to its monolithic alternative? No. Do they provide key advantages that are worth the effort to transform existing applications in a majority of enterprise applications? Together with accompanying transformation to a productive DevOps culture, absolutely yes.

To achieve the stated performance goals microservices must be assessed not only individually, but in terms of their interactions with other application services. This particular challenge of communication between microservices has led to creative abstractions of the entire network service layer, including connectivity and security, via solutions like service mesh. Other aspects of microservice management, such as assuring that microservices deployed as containers are grouped with related services, or that they are deployed to the right type of environment led to container orchestration tools – most notably Kubernetes. Even after the above stated challenges are met, the design of a large application based on microservices will have to implement a well-designed API layer in order to produce an application that is demonstrably superior by measures beyond resilience.

Looking Ahead – With Magenic

In any journey, picking the right traveling companion is key. To learn more about microservice architecture and how it can accelerate the agility of your company, **contact Magenic today**.

About Magenic

Magenic is the digital technology consulting company built for speed. We have the right digital strategies, the right process, and the right people to get our clients' digital products to market faster.

Visit us at **magenic.com** or call us at **877.277.1044** to learn more or to engage Magenic today.

Magenic® // Fast Forward