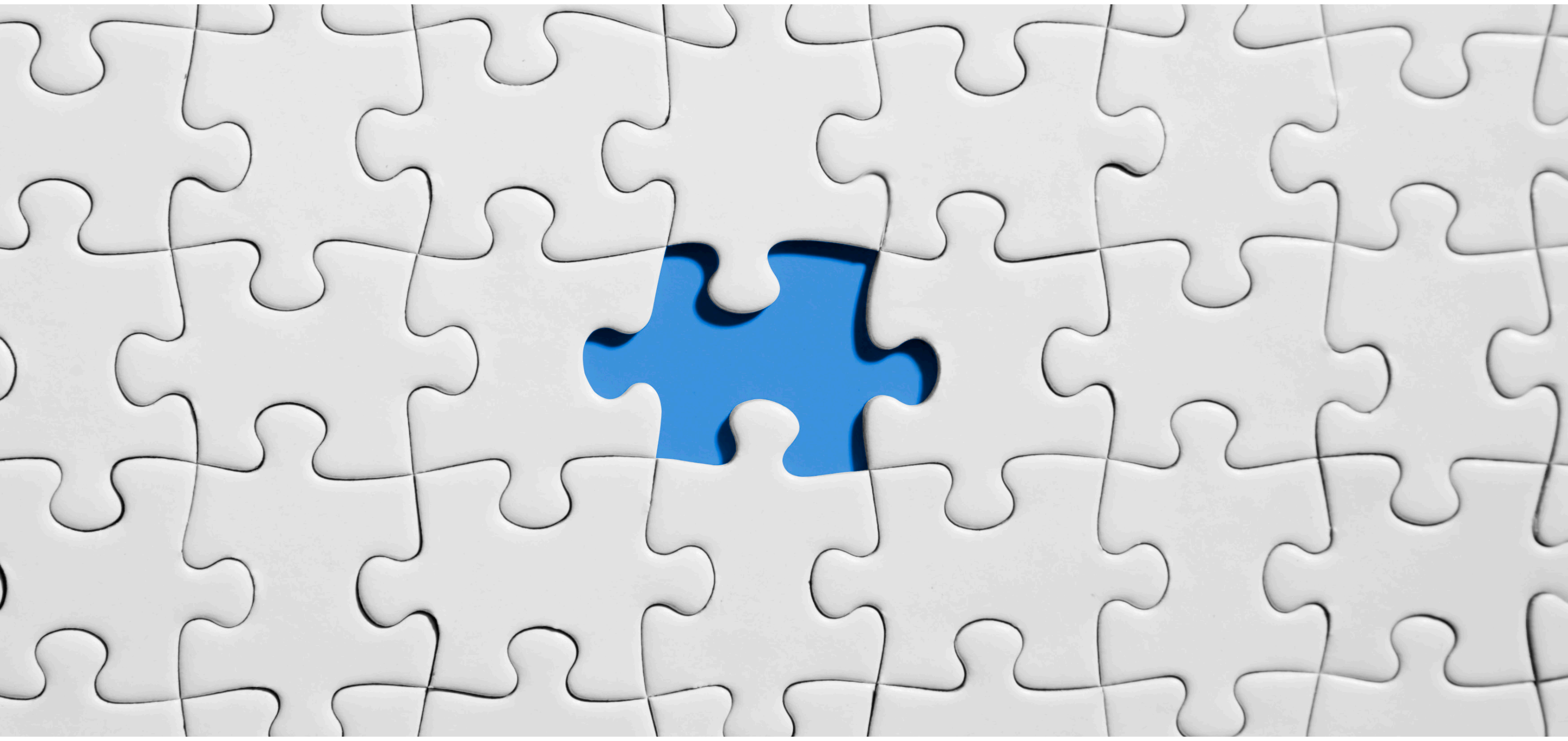
MICROSERVICES: INDEPENDENTLY DEPLOYABLE SERVICES



Swati Sinha
Inside Product Specialist
Dell Technologies
Swati.sinha@dell.com

Anuj Seth
Inside Product Specialist
Dell Technologies
Anuj.seth@dell.com

Anirudh Sandur
Inside Product Specialist
Dell Technologies
Anirudh.sandur@dell.com

The Dell Technologies Proven Professional Certification program validates a wide range of skills and competencies across multiple technologies and products.

From Associate, entry-level courses to Expert-level, experience-based exams, all professionals in or looking to begin a career in IT benefit from industry-leading training and certification paths from one of the world's most trusted technology partners.

Proven Professional certifications include:

- Cloud
- Converged/Hyperconverged Infrastructure
- Data Protection
- Data Science
- Networking
- Security
- Servers
- Storage
- Enterprise Architect

Courses are offered to meet different learning styles and schedules, including self-paced On Demand, remote-based Virtual Instructor-Led and in-person Classrooms.

Whether you are an experienced IT professional or just getting started, Dell Technologies Proven Professional certifications are designed to clearly signal proficiency to colleagues and employers.

Learn more at www.dell.com/certification

Table of Contents

Introduction to Microservices

Microservices is an application architecture that takes every application function and puts it in its own service that runs in a container; these containers communicate over API.

In detail, Microservices are an architectural approach to building applications. As an architectural framework, microservices are distributed and loosely coupled, so one team's changes won't break the entire app. The benefit to using microservices is that development teams can rapidly build new components of apps to meet changing business needs. Microservices architecture offer lightweight and services architecture that comes handy in modern technology world.

Abstract

This article provides a high-level overview of "Microservices", a highly scalable structural style for developing software applications today. The term "Microservice Architecture" has emerged over the past few years to describe a particular way of designing a software application as suites made up of independently deployable services.

This article will serve as a reference document for anyone seeking to understand Microservices and its architecture and also provides a starting point to appropriately look at best practices of Microservices, including:

- An introduction to Microservices.
- Evolution of Monolith to Microservice Architecture.
- How could we look at the right sizing of microservices?
- How to divide a monolith and how much to divide it up by. When is the monolith considered too small?
- How to take a requirement and design a microservices system from those requirements.

Also covered is how microservice architecture is optimized for development of cloud-native applications. Boundaries and advantages of microservices are also presented with the help of a use-case.

Evolution of Monolith to Microservice Architecture

Microservices stem from an evolutionary design background and see service decomposition as the future toward enabling application developers to control changes in their application without application downtime. Change control doesn't necessarily mean change reduction – with the right approach and decomposition of services you can make frequent, fast, and necessary changes to an application. We will discuss the monolith architecture and advantages that microservices provides.

Monolith Architecture

Monolith Architecture is a service-side system based on a single application. For example, in Java, an application will be put into a jar file and deployed as whole into a production environment.

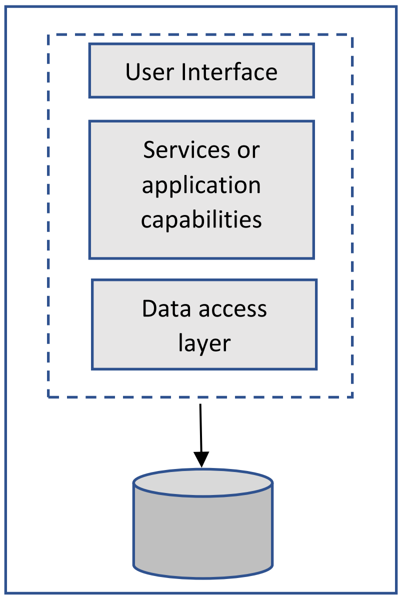


Figure 1: Monolithic Architecture

Microservice Architecture

A microservice is a core function of an application and runs independent of other services. However, a microservices architecture is about more than just the loose coupling of an app's core functions; it's about restructuring development teams and interservice communication in a way that prepares for unexpected failures, future scalability, and adding new capabilities.

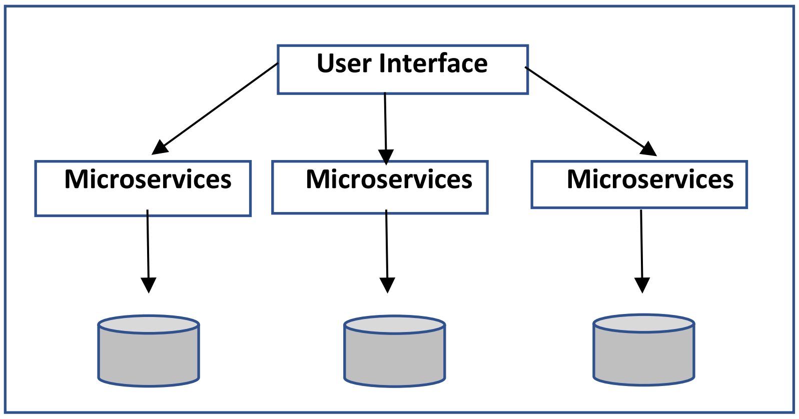


Figure 2: Microservices Architecture

Advantages of Microservices Over Monolith Architecture

Consider an online application that provides ticketing service for various events. Now let's compare the same application using the Monolith architecture and Microservices.

In the Monolithic world, architecture is something as represented in Figure 3:

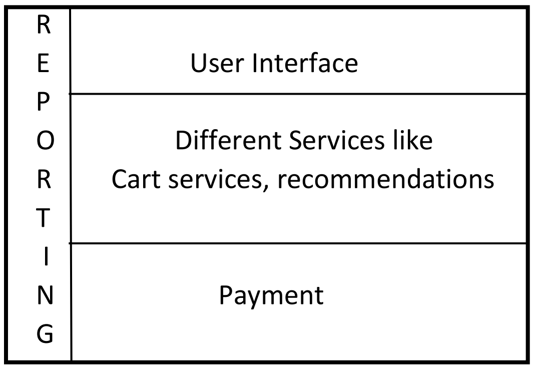


Figure 3: Monolithic Architecture of ticketing application

This architecture is highly dependent, having shared libraries within architecture. If we want to make any changes, we must understand which components rely on which shared library. Their language or framework is another drawback; if this application is written in Java and changes or additional features to the application are needed, they must be done only through Java. This limits and bounds the development team.

Adding features or capabilities to the application becomes much larger and very difficult for the DevOps team to understand and remember every little thing about the application changes. Consequently, as an application grows, the architecture becomes more complex.

Also, adding capabilities requires shutting down the application on Friday, deploy and stabilize the application over the weekend and make sure it is all set by Monday. This would be frustrating and painful for many teams. When there is a surge of user traffic, one more instance of an application must be deployed, which is time consuming as well.

Summarizing the disadvantages of using in Monolithic architecture:

- It is highly dependent.
- Language or framework used is constraint.
- Growth, deployment and scaling is difficult.

Now, let's compare this application function using Microservices Architecture.

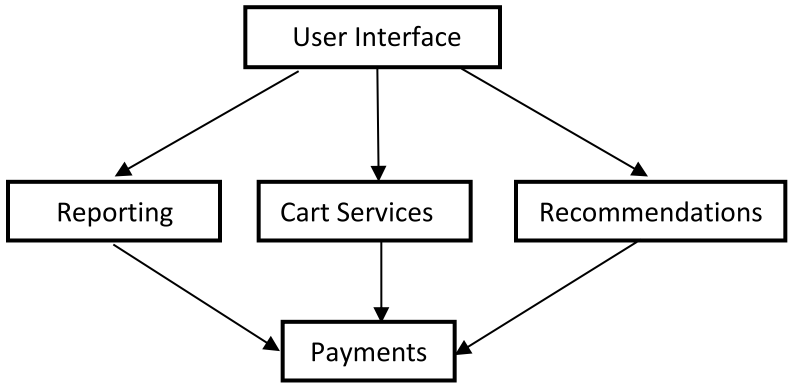


Figure 4: Microservice Architecture of ticketing application

Figure 4 shows how every app function is its own service having its own container and all the functions communicate via API. This enables different service teams to use the language or framework they are familiar with and need not rely on other teams as would be the case with Monolithic or traditional architecture. Also, there is no need to wait for the entire application to be ready to test. As and when each service is completely developed it is tested and placed in DevOps pipeline. This saves a lot of time of different teams and need to be dependent.

Some advantages of Microservices over Monolithic architecture are:

- Independent.
- Language/Framework is not a constraint.
- Less risk when changing or adding features.
- Independent scaling.

Rightsizing of Microservice

Given that microservices are supposed to be "micro", there's a lot of discussion about the right size. A typical answer to this question is: A microservice should do just one thing. We don't really think that's an answer, as "one thing" leaves a lot of room for interpretation. Most people suggest that each individual microservice should be as small as a single function, but we strongly disagree with this for almost every situation. Consider, for example, a function that computes something based on three input values and returns a result. Is that a good candidate for a microservice, i.e. should it be a separately deployable application of its own?

We believe it's easier to approach this from the opposite direction. For example, a web-based email system. Let's not overcomplicate things. Assume it's traditional and offers the minimal features you'd expect, such as logging in and out, maintaining some user settings, creating messages (from scratch or by replying to or forwarding an existing one), deleting messages, viewing your inbox, moving messages into folders (that you can create, view and delete), maintaining an address book, search for messages, etc. At one extreme, we could absolutely build this as a single application and ensure it's built not as a single package but using a reasonable internal modularization strategy. We could decide to write its core as a set of collaborating classes, maybe adhering to the domain-driven design (DDD) approach which would classify the classes according to the role they play. Then we'd add the dependencies to the outside world, such as the UI, the data storage, external systems (such as maybe external mail handlers, LDAP (Lightweight Directory Access Protocol)directories, etc.), possibly using a layered or hexagonal architecture.

The team(s) working on this application would need to synchronize very tightly, as the whole thing is released at once. It will also be scaled in an all-or-nothing fashion, and will be down or up and running completely, not partially. That may be perfectly fine! But let's assume (again) you deem it's not and want to cut it apart into separate parts that have their own lifecycle and are isolated from each other.

How would you go about decomposing this into separate applications or services? First, the login/logout stuff (the auth system) is a good candidate, as is the user profile. They could go into one service, but if we consider that the auth system has to maintain passwords (or rather, password hashes), it makes sense in my view to treat it differently from the rest. The emails and folders themselves seem quite cohesive to me, though: You could separate them, but we probably wouldn't. If there are multiple ways to connect to the outside world, i.e. via the Web interface, POP3, IMAP, and SMTP, I can imagine each of those being their own service. Maybe we would factor out the storage of messages into its own service, one that doesn't know the difference between a document and an email. We think the address book, including its data storage, its UI and its API seems like a natural candidate to be separated from the rest.

But in all, we would probably end up with a dozen, maybe twenty or thirty services (or self-contained systems, as I prefer to call them). More importantly, we think that for any given interaction triggered by some outside event – e.g. a user clicking a button after entering data into a form – I'd end up touching maybe 3-5 of them.

In other words, it's not a goal to make your services as small as possible. Doing so would mean you view the separation into individual, stand-alone services as your only structuring mechanism, while it should be only one of many.

Designing microservice system from given requirements

We need to understand the fundamental components of a microservice if we want the resulting artifact to operate properly and not end up looking like the same old monolithic application with a new paint job.

Here are five elements that your microservice will need before it can take its place in a distributed application architecture.

Properly scoped functionality

The biggest design issue with monolithic application architectures is that there's so much code in them that implements widely differing functionality. To make any change to a monolithic app, you must coordinate across different groups to ensure that everyone's code continues operating properly. As a result, developers often spend more time on integration and testing than on delivering new application capability.

For this reason, the first element of a microservice is to define what it should do. What is the breadth of functionality it should implement? On their initial foray into microservices, many people are concerned that they'll over-partition their functionality and end up with too many tiny microservices. In my experience, over-partitioning is rarely the issue; it's more common to stuff too much into each service.

One way to define the proper scope is to partition the services along logical functionality lines. For example, a tax lookup function in your monolithic app that many other functions call is a candidate to be broken out into its own service.

Another scoping approach is to mirror the development organization's structure. Each application subgroup (e.g. the authentication group responsible for user identity and authorization) takes responsibility for creating one or more microservices for the functionality that falls into its area.

Presenting an API

Once you break up a single application into multiple cooperating services, how should the services talk to one another? Typically, this is done with Representational State Transfer (REST) web services API calls, although you can use other transport mechanisms as well.

Presenting an API to calling services in some way represents the old challenge of integration. For an overall application to run properly, each of the individual services must be able to reliably send and receive data, and testing that APIs operate properly is necessary to ensure that everything hangs together.

The foundation of an API is exposing the service at a known location with a format that, when called by a client service, can respond with the appropriate functionality and/or response data. Recognize, though, that as individual services mature, they may add new functionality that requires a richer API. This, in turn, implies that the new API must be exposed alongside the old one. Absent this, every API change cascades into a requirement that all callers update their code and retest, which results in the same problem that monolithic applications pose.

Traffic management

In the real world of operations applications, a service may run slowly, and calls to it to take a long time. Or a service can be overwhelmed with calls and lack the processing power needed to respond quickly enough. Even worse, a service might simply stop running due to a software or hardware crash. And sometimes a client is issuing too many calls for the lower-level service to respond quickly enough.

Addressing this too-heavy traffic situation requires management. There must be a way for calling and called services to communicate status and coordinate traffic loads.

From the perspective of the calling service, it should always track its calls and be prepared to terminate them if the response takes too long. From the perspective of the called service, the API design should include the ability to send a response that indicates overload. This response, typically referred to as backpressure, signals that the calling service should reduce or redirect its load.

Data offloading

The vagaries and erratic traffic of microservice applications mean that individual services come and go. Adding to the constant service instance churn is the reality that the underlying infrastructure also is unreliable. Virtual machines crash, fail to respond or go into high-load status while not performing any useful work (thereby requiring hard termination). Nevertheless, while individual services instances are transient, the overall service must be available and continue operating so that users will keep obtaining results from the application.

This need for continuous operation is quite different from traditional applications, which often stop operating if the underlying infrastructure fails.

To ensure that users can continue to perform useful work despite failure of one instance from which their sessions are being served, you can migrate user-specific data off of service instances and into a shared, redundant storage system that's accessible from all service instances. Thus, you can ensure that no instance crash stops user interactions.

Monitoring

Decomposition of a monolithic application, along with insertion of offloaded data layer and caching to increase performance, inevitably means a more complex application topology — a lot more complex.

For this reason, traditional monitoring tools and approaches cannot deal with the scale and dynamic environments associated with microservices. The monitoring system for a microservices-based application must allow for ongoing resource change, be able to capture monitoring data in a central location, and display information that reflects the frequently changing nature of microservices applications.

But more is necessary to deliver useful metrics for microservices applications. As an end-user action triggers application work, API calls and service work cascade down the application topology, and a single action may result in tens, or hundreds, of monitorable events. Trying to manually correlate errors across a service cascade is nearly impossible, so use a monitoring system that can discover and display events based on a common timeline to support root cause analysis.

Most microservices monitoring systems place a monitoring agent on each service instance, where it can track specific instance data. These monitoring systems can also capture application-created log information. All this data migrates to a centralized database, where the system does cross-correlation, allowing monitoring alerts or humans to track important event data.

Cloud-Native Approach with Microservices

In February 2020, the Cloud Microservices Market Research Report predicted global microservice architecture market size would increase at a CAGR of 21.37% from 2019 to 2026 and reach $3.1 billion by 2026.

Cloud-native applications are the future; there are no two ways about it.

With newer research and development taking place daily, here are some characteristics of what a modern cloud-native application should consist of:

It Should Be 'Dockerized'

Docker is an open-source software platform used to create, manage, and deploy application containers in an operating system.

Allowing a higher degree of portability, Docker has become extremely popular due to its efficient application development, low use of system resources, and quicker deployment time compared to virtual machines.

Utilize A ‘Designed for Failure’ Approach

System failures are inevitable – despite many attempts to perfect the hardware and server availability, they still occur. This approach advocates for developers to design applications that support swift and rapid recovery.

Simply put, this leads to quick deployment across redundant cloud components with minimal common points of failure, designing every component to be partition-tolerant and distribution of components across availability zones, amongst others.

Autoscaling

As the name implies, this refers to the intuitive scaling of the computational resources in a server farm based on the load. It enables companies to incur reduced cloud computing charges, allows for efficient energy use, and provides greater uptime and availability.

Infrastructure as Code (IaC)

IaC leads to efficient management and provision of data through machine-readable files, as opposed to physical or interactive configuration tools. As organizations face scaling problems, this can provide cost-effective, quick and error-free infrastructure modeling with the help of code.

Conclusion

With modern applications becoming larger and more intricate, the software development industry has realized the shortcomings of the conventional monolith design. As major organizations continue to switch to a microservices architecture style, many more will also do so in pursuit of a better data management approach.

While the monolith approach does provide value in certain cases, it’s become clear that microservices architecture enables organizations to become more agile and responsive to consumer demands – two key aspects for success in the modern economy.

References

https://www.innoq.com/blog/st/2014/11/how-small-should-your-microservice-be/

https://www.informit.com/articles/article.aspx?p=2738465

https://tech.transferwise.com/good-size-for-a-microservice/

https://cloud.google.com/files/Cloud-native-approach-with-microservices.pdf

https://www.instanttechnews.com/technology-news/2020/02/16/cloud-microservices-market-2020-trends-market-share-industry-size-opportunities-analysis-and-forecast-by-2026/

https://martinfowler.com/articles/break-monolith-into-microservices.html

https://techbeacon.com/app-dev-testing/creating-microservice-design-first-code-later