What is a service, and when is a service a microservice?

Good question. When using a service, we leverage a remote method or behavior versus simply extracting or publishing information to a remote system. Moreover, we typically abstract this remote service into another application known as a composite application, which is usually made up of more than one service.

A good example of a service is a risk analysis process, which runs within an enterprise to calculate the risk of a financial transaction. This remote application service is of little use by itself, but when abstracted into a larger application—for example, a trading system—that remote application service has additional value.

Note that we leverage the behavior of this remote service more than the information it produces or consumes. If you’re a programmer, you can view application services as subroutines or methods—something you invoke to make something happen.

The basic notion of service-oriented architecture (SOA), and SOA using cloud computing, is to leverage these remote services using some controlled infrastructure that allows applications to invoke remote application services as if they were local to the application. The result (or goal) is a composite application made up of many local and remote application services. Since they’re location and platform independent, they can reside on premises or within one of many cloud computing providers.

Furthermore, once services are identified and exposed, or developed from scratch, we might have services that can be placed on and span both on-premises and cloud-enabled platforms. So, those are services in general.1

Enter Microservices

Microservices is an architecture as well as a mechanism, and is often confused with traditional SOA-type services. Indeed, there’s a great deal of overlap. It’s an architectural pattern in which complex applications are composed of small, independent processes that communicate with each other using language-agnostic APIs.

This is service-oriented computing, at its essence, decomposing the application down to the functional primitive, and building it as sets of services that can be leveraged by other applications, or the application itself. This is also the foundation of reuse, and these services are systemic to the use of containers as well as non-container-based applications. (See https://blog.akana.com/the-venn-of-microservices.)

The benefits of this approach include efficiencies through reuse of microservices. As we rebuild applications for the cloud, we modify them to expose services that are accessible by other applications. More importantly, we can consume services from the rebuilt application so we don’t have to build functionality from scratch.
For instance, some programs have built-in systems such as credit validations, mapping, and address validation services that must be maintained. This can cost upward of hundreds of thousands of dollars per year. The service-based approach lets us reach out and consume remote services that provide this functionality and more, so you can get out of the business of maintaining services that can be found in other places. It also lets us expose services for use within the enterprise by other applications, or even sell services to other enterprises over the open Internet.

Enter Containers
The use of containers to wrap or containerize existing applications comes with a few advantages, including the ability to reduce complexity by leveraging container abstractions. The containers remove dependencies on the underlying infrastructure services, which reduces the complexity of dealing with those platforms. This means we can abstract the access to resources, such as storage, from the application itself. This makes the application portable, but also speeds the refactoring of the applications, since the containers handle much of the access to native cloud resources.

Containers also offer the ability to leverage automation to maximize their portability, and, with portability, their value. Through the use of automation, we’re scripting a feature we could also do manually, such as migrating containers from one cloud to another. However, this option’s use cases have proved limited. Indeed, most new applications are built to take advantage of containers, but existing applications are often difficult to containerize. The objective of leveraging containers seems to be that of a distributed architectural value versus portability, as originally thought. However, portability is always a byproduct of leveraging containers.

Also consider the ability to provide better security and governance services by placing those services around rather than within containers. In many instances, security and governance services are platform specific, not application specific. For example, traditional on-premises applications tend not to have security and governance functions innate to the application. The ability to place security and governance services outside of the application domain provides better portability and less complexity when refactoring. The ability to leverage microservices in this context provides the same advantages as well, no matter if you’re using containers or not.

Containers can provide better distributed computing capabilities as well. A traditional application can be divided into many different domains, all residing within containers. These containers can be run on any number of different cloud platforms, including those that provide the highest cost and performance efficiencies. So, applications can be distributed and optimized according to their utilization of the platform from within the container.

For example, one could place an I/O-intensive portion of the application on a bare-metal cloud that provides the best performance, place a compute-intensive portion of the application on a public cloud that can provide the proper scaling and load balancing, and perhaps even place a portion of the application on traditional hardware and software. All of these elements work together to form the application, and the application is separated into components that can be optimized.

Refactoring Traditional Applications for Containers and Microservices
The process of containerizing an application and service enabling it at the same time is more art than science at this point. However, certain success patterns are beginning to emerge as enterprises begin to migrate traditional applications to the cloud using containers and service orientation as the architecture.

Pattern one decides quickly how the application is to be broken into components that will be run inside of containers in a distributed environment. This means breaking the application down to its functional primitives, and building it back up as component pieces to minimize the amount of code that needs to be changed.

Pattern two builds data access as a service for the application’s use, and has all data calls go through those data services. This will decouple the data from the application components (containers) and let you change the data without breaking the application. Moreover, you’re putting data complexity into its
own domain, which will be another container that’s accessed using data-oriented microservices.

Pattern three splurges on testing. Although many will point to the stability of containers as a way around black-box and white-box testing, the application now exists in a new architecture with new service dependencies. There could be a lot of debugging that has to occur up front, before deployment.

There are other sides to this as well. Lori MacVittie, one of my advisory board members, noted in an email that containers and microservices seem to mix many tangentially related topics together, but microservices have nothing to do with containers other than they happened to appear at the same time.

The focus here should be on the technologies working together, not on each technology as a standalone. The concept is to understand that there are some additive advantages of leveraging both containers and microservices, which are indeed independent. That said, if the notion is that microservices are indeed services in the traditional sense, I have no retort for that.

**Operational Considerations**

Cloud-enabled traditional applications must be managed differently in production than they were prior to migration. This phase is known as cloud operations, or the operation of the application containers in the cloud.

When applications are put into production, those charged with cloud operations should take advantage of the container architecture. Manage them as distributed components that function together to form the applications, and are also separately scaled. For instance, the container that manages the user interface can be replicated across servers as the demand for that container goes up when users log on in the morning. This provides a handy way for cloud operations to build autoscaling features around the application, to expand and de-expand the use of cloud resources as needs change.

Most enterprises believe the cloud will become the new home for applications. However, not all applications are fit for the cloud—at least, not yet. Care must be taken to select the right applications to make the move.

The use of containers and microservices makes things easier. This approach forces the application developer charged with refactoring the application to think about how to best redesign the applications to become containerized and service oriented. In essence, you’re taking a monolithic application and turning it into something that’s more complex and distributed. However, it should also be more productive, agile, and cost effective. That’s the real objective here.

**Microservice Coupling**

As we create microservices to serve new or existing applications, we need to understand the benefits of being loosely coupled. Loose coupling, as related to microservices, has a few basic patterns.

In the *location independence* pattern, it doesn’t matter where the microservice exists; the other components that need to leverage the service can discover it within a directory and leverage it through the late binding process. This comes in handy when you’re leveraging microservices that are consistently changing physical and logical locations, especially services outside of your organization that you might not own, such as cloud-delivered resources. Your risk calculation service might exist on premises on Monday and within the cloud on Tuesday, and it should make no difference to you.

Dynamic discovery is key to this concept, meaning that calling components can locate microservice information as needed, and without
having to bind tightly to the service. Typically these services are private, shared, or public services as they exist within the directory.

In the communications independence pattern, all components can talk to each other, no matter how they communicate at the interface or protocol levels. Thus, we leverage enabling standards, such as microservices, to mediate the protocol and interface difference.

The security independence pattern is based on the concept of mediating the difference between security models in and between components. This is a bit difficult to pull off, but necessary to any service-based architecture. To enable this pattern, you have to leverage a federated security system that can create trust between components, no matter what security model is local to the components. This has been the primary force behind the number of federated security standards that have emerged in support of a loosely coupled model and Web services.

In the instance independence patterns, the architecture should support component-to-component communications using both a synchronous and an asynchronous model, and not require that the other component be in any particular state before receiving the request or message. Thus, if done right, all of the services should be able to service any requesting component asynchronously, and retain and manage state no matter what the sequencing is.

The need for loosely coupled architecture within your cloud computing solution is really not the question. If you leverage cloud computing correctly, you should have a loosely coupled architecture, except in some rare circumstances. Analysis and planning are also part of the mix, as are understanding your requirements and how each component of your architecture should leverage the other components of your architecture. Leverage the coupling model that works best for your requirements.

Although this seems like a lot of work, it’s really only a quick survey and explanation of the present services. Moreover, whereas I recommend basic concepts and basic approaches, the needs of your IT environment will be unique and might require slightly different approaches. As long as the objective of having a complete understanding of the microservices within the problem domain are achieved, how you do that project is up to you. Another benefit of understanding the domain at a services level is that you can easily leverage this work in other directions, perhaps to support the core enterprise architecture or build and/or refine your architecture.

Acknowledgments
Portions of this article were adapted from an article I wrote for TechBeacon.2

References

David S. Linticum is senior vice president of Cloud Technology Partners. He’s also Gigaom’s research analyst and frequently writes for InfoWorld on deep technology subjects. His research interests include complex distributed systems, including cloud computing, data integration, service-oriented architecture, Internet of Things, and big data systems. Contact him at david@davidlinticum.com.