Managing Microservices

CONTENTS

WRITTEN BY ARIFF KASSAM
CHIEF TECHNOLOGY OFFICER, NUODB

Overview

The promise of microservices is the ability to increase developer agility by breaking the application into smaller, more manageable components with clear interfaces. As a result, making changes to a microservice requires less coordination between components and less testing. These smaller components significantly improve the agility, scalability, and availability of applications, which offers significant benefits to developers as they seek to deploy new application features rapidly to meet customer demands.

While unquestionably adding to agility, microservices aren't for every application. When starting with microservices, it's important to remember that it's not something you can adopt in a vacuum. Microservices require significant development and delivery skills, including security built in and automated at every layer of development, a mature DevOps environment, and a high degree of standardization and automation using technology such as containers and Kubernetes for container orchestration.

While microservices can be used in any industry, right now it makes the most sense to adopt them in organizations that use Agile development methodologies and need to make changes to customer facing applications quickly. Highly regulated industries and those that use Waterfall development methodologies and have less frequent software releases, such as healthcare, government, oil and gas, and manufacturing, may not benefit from a move to microservices architectures. Other industries, such as financial services, already have an agile development environment and a customer base that demands innovation and rapid delivery and, therefore, benefit greatly from the adoption of microservices. Industries that require services to always be available, reliable, and responsively scalable based on real-time demand benefit from a move to microservices.

Microservices and FinTech: A Use Case

Financial organizations have been facing significant competition and disruption due to the introduction, and now expectation, of technology that allows end users to rapidly make mobile payments, transfer

money, make or request loans, conduct fundraising, and access wealth management tools. FinTech, or financial technology, encompasses any type of technology in financial services, from mobile payment apps to cryptocurrency.

FinTech originally referred to technology applied to financial institutions' back-end systems, but today includes many more consumer-focused applications, including smartphone apps to manage funds and make payments, trade stocks, exchange cryptocurrency, and make budgets. FinTech isn't just for consumers, however. It can also provide better financial services to businesses, such as expense tracking, accounting software, employee payment, and even sales tracking and invoicing.

Many banks and early FinTech companies were built on legacy systems, which they've relied on as they build new apps and interfaces, but the shift towards digital services necessitates greater flexibility and agility. Microservices are a perfect fit, as financial services organizations seek agility and scalability, but may require significant cultural and architectural shifts. A few important Agile and DevOps competencies for application delivery teams looking to make the transition to microservices include:

- Security embedded in DevOps processes.
- Continuous Integration and Continuous Delivery (CI/CD).
- Automation of Core Infrastructure and Releases.

Microservices Benefits and Requirements

PROGRAMMING LANGUAGE AGNOSTIC

Microservices do not force any specific programming model, style, or language due to their technology independence. As each microservice communicates with other services through standard channels like APIs, they don't rely on technology-related restrictions. This enables development teams to choose the programming language that works best for each microservice, as well as choosing a particular pattern or database based on what's best for their use case. Using microservices and containers, a single instance can host native code, .NET, Java, or any other programming language that is the best fit for the particular microservice.

REGULATIONS

New regulations are emerging to address the evolving digital payment economy, which aim to address questions about the use and ownership of data, operational resilience and business continuity, and increasing competition. In the European Union, the Payment Services Directive version 2, or PSD2, aims to bring banking into the open API economy, which will drive interoperability and integration through open standards. In order to meet these standards, financial services will benefit from the advantages microservices bring to APIs, integration, and open data requirements.

LOGGING AND MONITORING

Logging and monitoring go hand in hand with security, and are essential for successful microservices deployment. Since microservices architecture is highly distributed, it can be hard to determine where a failure occurred and what caused it without logging and monitoring tools. Due to the fact that microservices break an application into many smaller components, logging generates a lot of data, so it's helpful to choose tools that help you parse or visualize the results. Prometheus is an open source application used for event monitoring and learning.

CONTINUOUS INTEGRATION (CI)/CONTINUOUS DELIVERY (CD)

There are many continuous integration build systems available that provide access to pipeline builds, such as TeamCity, Bamboo, and Jenkins. Microservices scale extremely well, accommodating increasing numbers of users and transactions and delivering new functionality rapidly. Responding in real time to demand isn't an option without an effective implementation of a CI/CD solution.

INTEGRATION

Microservices make it easier to manage and secure the API layer through isolation, scalability, and resilience. The APIs enable easy communication with both internal and external services. Well-defined APIs enable you to prevent dependencies between microservices while still enabling the exchange of data.

SECURITY

When deploying microservices, it's important to build security into your development processes. By embedding security processes and automating them, it's possible to bolster security in your applications. The platform used to deploy microservices should provide developers with options for identity and access control and authorization (such as OAuth), certificate management capabilities, automated security updates, periodic automated vulnerability and security scanning, and control over the container images available for use to limit security risks.

DATA PERSISTENCE

A microservices architecture isolates each microservice from the others, so you have a choice when addressing data storage and data persistence. This is not the place to permit open access to a single monolithic database, which traditional monolithic development environments frequently allow. If your microservices can communicate with one another through the database, you're likely to see unexpected coupling. Different microservices each own the data related to the business functionality supported by that service, and they may require different options. Some will best use a NoSQL storage system, while others require a SQL-based relational database. Regardless, each microservice should use unique credentials and access should be limited to that microservice's data. Some examples of databases that pair well with microservices are:

- MongoDB, an open source NoSQL database.
- NuoDB Community Edition (CE), a free distributed SQL database.

CONTAINERS

Containers are the most common deployment mechanism for microservices, in part because they deliver many of the same benefits that microservices offer, such as platform independence, scalability, isolation,

resource efficiency, and speed. Because containers can easily be scaled out and back, it's relatively simple to spin up a number of instances of a specific version of your service for development, test, staging, and production environments.

DISTRIBUTED SYSTEMS

Microservices architecture enables the concept of decentralized data management, supporting distributed deployment. Microservices deployed in containers allow you to reprovision any container individually, or replace, deprecate, or add new microservices when there's a new feature to release, a vulnerability to patch, a bug to resolve, or any other change that you want to roll out rapidly. They're also simple to scale independently and responsively, so you can scale out based on demand for a particular service without scaling the entire application.

LOAD BALANCING AND RESILIENCY

In a microservices environment, load balancing functionality is typically moved into the software layer, performing the load balancing logic at the distributed edge. Microservices also provide resiliency by handling errors through retries, queuing, deadlines, and default and caching behaviors. When done correctly, microservices architecture can help you deliver self-healing applications that operate even when there are partial outages, automatically deploying new containers that allow your application to recover quickly and seamlessly.

MANAGING SERVICES

When first starting with microservices, your microservices architecture is likely to be small, and managing them seems simple. However, as the number of microservices grows, you need to start thinking about the macro-architecture of your microservices environment. There are many solutions that offer infrastructure, often referred to as service meshes. Service meshes provide a **control plane**, which sets the policy that will be enacted by the data plane through configuration files, API calls, and user interfaces, and a **data plane**, which translates, forwards, and observes every network packet that flows to and from a service instance.

- Linkerd is an open source service mesh that acts as a proxy between services and provides load balancing, circuit breaking, service discovery, dynamic request routing, HTTP proxy integration, retries and deadlines, TLS, transparent proxying, distributed tracing, and instrumentation. Protocol support includes HTTP/1.x, HTTP/2, gRPC, and anything TCP-based.
- Istio is another widely used open source service mesh and provides automatic load balancing, fault injecting, traffic shaping, timeouts, circuit breaking, mirroring, and access controls for HTTP, gRPC, WebSocket, TCP traffic, automatic metrics, logs, and traces, and infrastructure level run-time routing of messages.

How to Manage Microservices

With the rise of popularity of microservice architectures, containers are now the best deployment mechanism for microservices. As noted above, containers provide many benefits, including platform independence, resource efficiency, speed, and isolation. They also enable flexibility and scalability for organizations deploying applications, because containers can easily be scaled out and back individually based on demand for those services.

As containers have grown in popularity, managing them at scale is the next real challenge, which created a need for container orchestration. In response to this need, Google released Kubernetes as an open source platform for automating deployment, scaling, networking, managing, and maintaining availability for container-based applications. Using microservices, containers, and container orchestration tools together can simplify running applications in the cloud, which greatly improves business agility.

Using microservices, containers, and container orchestration, developers now have on-demand access to IT resources and the architectural paradigms that help them speed up the process of both development and moving code from the dev environment to production. This significantly improves your ability to transform slowly maturing applications into adaptable containerized microservices. This is particularly true when building applications that are stateless.

Building and deploying stateless applications in containers is relatively easy. Every time you start a stateless application it has the same information that it does every time you start it, which makes them easy to scale horizontally to accommodate increased user demands (add more instances) and protect against failures (start new instances).

Many applications, however, require persistent state. That means that these applications require the ability to store data to protect against failures so that the application will not lose any data. Traditionally, stateful applications have been much harder to fit into the world of containers. While databases are the standard for managing state for applications, traditional databases have a number of issues when managed by a container orchestration solution, typically Kubernetes.

EXTERNAL PROCESS LIFECYCLE MANAGEMENT

Kubernetes automatically distributes running containers across the cluster, which is one of the advantages it provides. If a machine fails, any containers running on that system are automatically restarted on other nodes in the cluster. Kubernetes also automatically rebalances container distribution periodically; it's fairly common for containers to be stopped and then restarted on different nodes. Kubernetes controls the lifecycle management of container processes.

It's important to note that this lifecycle management is simple for stateless containers. Stateless containers can be started and stopped at any time, and stateless containers can be run on any node in the cluster. As long as you have at least one instance of the container running at any time, the service that application provides is always available.

Stateful containers aren't as flexible, partly because the state information needs to be accessible on any node to which the container can be

moved. Kubernetes recently added container-native storage solutions to allow the state to be accessed in this way. Issues related to container lifecycle management remain, such as: to migrate a container from one node to the other, Kubernetes shuts down the current container and starts a new container. It's possible that the two container instances (new and old) briefly run concurrently during this time, which means that applications could connect to either instance.

Traditional databases can't handle this scenario, because there can only be a single "active" instance of the database at any given time. All data written to the database instance being shut down is lost while the new container becomes the active one. To work in containers and the Kubernetes orchestration environment, databases must be capable of handling multiple processes running at the same time ***without*** any data loss.

SCALE OUT

By design, Kubernetes addresses performance issues by deploying more containers, thus enabling horizontal scale out. This horizontal scale out is simple for stateless applications. However, because traditional databases only support a single "active" process, they require scale up, not scale out. Scale up doesn't translate well to the Kubernetes environment. Kubernetes and containers are built to scale out based on demand, using as many or as few processes as necessary to handle throughput. Traditional relational databases can't spawn new Kubernetes pods; that would require a more expensive machine or necessitate that you shard your database. To work well in Kubernetes, you need a database that can scale out for both reads and writes on demand, not one limited by a single server.

CONSISTENCY

In scale out architectures, there are multiple instances of the container. For stateful containers, it's important that clients are able to connect to any instance of the container and receive a consistent view of the data. Different scale out databases have different consistency models. It's important for application developers to understand the consistency model supported by the database they are using.

For some applications, eventual consistency works. Some databases are able to achieve high availability in distributed environments using eventual consistency. With this consistency model, the application must be able to handle consistency conflicts. This may require significant application changes to support that ability. Many applications being migrated to the cloud and containers require a stricter consistency model, particularly applications handling business-critical data.

DevOps + Microservices

DevOps is an evolving philosophy, and its goals are to tightly link the development of

software and its delivery to IT Operations, thus improving the quality of the software systems as a whole. Much like microservices, a DevOps approach accomplishes this by segmenting the system into manageable components, which are owned by teams that can resolve issues that prevent the system from operating properly.

Most DevOps advocates consider CI and CD defining attributes of DevOps. Continuous Integration allows developers to integrate changes into the source code mainline as soon as they're completed, which is easier when creating microservices because there's less testing needed when each component is built to operate independently. Likewise, Continuous Delivery allows microservices to be updated as needed.

Automating the process using CI/CD tools is also essential for successful adoption of DevSecOps, which is when security is automated and integrated within DevOps. Including (and automating) security tools into the DevOps process is essential, because there simply isn't time in a mature microservices environment for security to be an afterthought. To build an environment in which microservices and security co-exist, you must develop both a plan and a framework for development, governance, and management of microservices.

OPERATORS

Kubernetes Operators help encode the human operational logic normally required to manage services running of a Kubernetes-native application and aim to make day-to-day operations easier. Operators on application container platforms, such as Red Hat OpenShift and Rancher, can help end users experience the next level of benefits from a Kubernetes-native infrastructure, with services designed to work across any cloud where Kubernetes runs. As microservices are typically delivered via containers, Operators are an important part of the deployment process for deploying stateful applications in Kubernetes.

Kubernetes Operators and operator catalogs, such as the new OpenShift OperatorHub and OperatorHub.io, take complicated technical solutions and make deploying them simple. When Operators were first made public in a 2016 CoreOS blog post, the goal of Operators was to make the software itself include operational knowledge that previously resided outside of the Kubernetes cluster. Operators simplify that process by implementing and automating the most common Day-1 and Day-2 activities in a piece of software running inside the Kubernetes cluster.

Operators make the process of modernizing existing applications and building new applications a lot easier. While Kubernetes has made it pretty easy to manage and scale web apps, mobile backends, and API services, until recently it's been more difficult to manage stateful applications such as databases, caches, and monitoring systems. The new application domain knowledge contained in Operators makes it possible to scale, upgrade, and configure these types of applications in Kubernetes in multiple pods across the cluster.

Using the Operator Lifecycle Manager (OLM), users can subscribe to an Operator --- including individual channels, such as stable vs. beta releases, so subscribers are continuously updated to the latest version and its new capabilities.

Conclusion

Cloud-native applications built on cloud-native infrastructure make it possible to increase the velocity of software delivery, enable developers to be more agile, and allow greater application scalability. By developing and delivering applications using microservices, containers, and Kubernetes, technology innovators can deliver the agility, scalability, and availability of applications that modern businesses and consumers demand.

In this Refcard, we've reviewed how microservices require the right infrastructure and technical skills within your organization. While microservices provide many advantages, there's a lot to consider when deploying your microservices-based infrastructure. From the considerable benefits of providing a programming language agnostic framework to the ability to respond quickly to changing regulations to the tools and skills essential for effective logging, monitoring, and integration, microservices enable a new degree of flexibility that many developers and engineers will be quick to embrace.

Using microservices deployed in containers, application container platforms, Kubernetes, and Operators, you bring many powerful tools together. These solutions enable applications to scale well to accommodate increasing numbers of users and transactions and deliver new functionality rapidly, which is essential for distributed systems. As you build new applications and redesign legacy applications, microservices will serve you well, provided you select solutions that portable, flexible, and scalable.

Written by **Ariff Kassam,** Chief Technology Officer, NuoDB

Ariff is responsible for defining and driving NuoDB's product strategy. Kassam brings 20 years of database and infrastructure experience to NuoDB to help the company achieve its vision of a distributed database that can manage an organization's most valuable data while exploiting the emerging benefits of modern infrastructures such as cloud and containers.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300