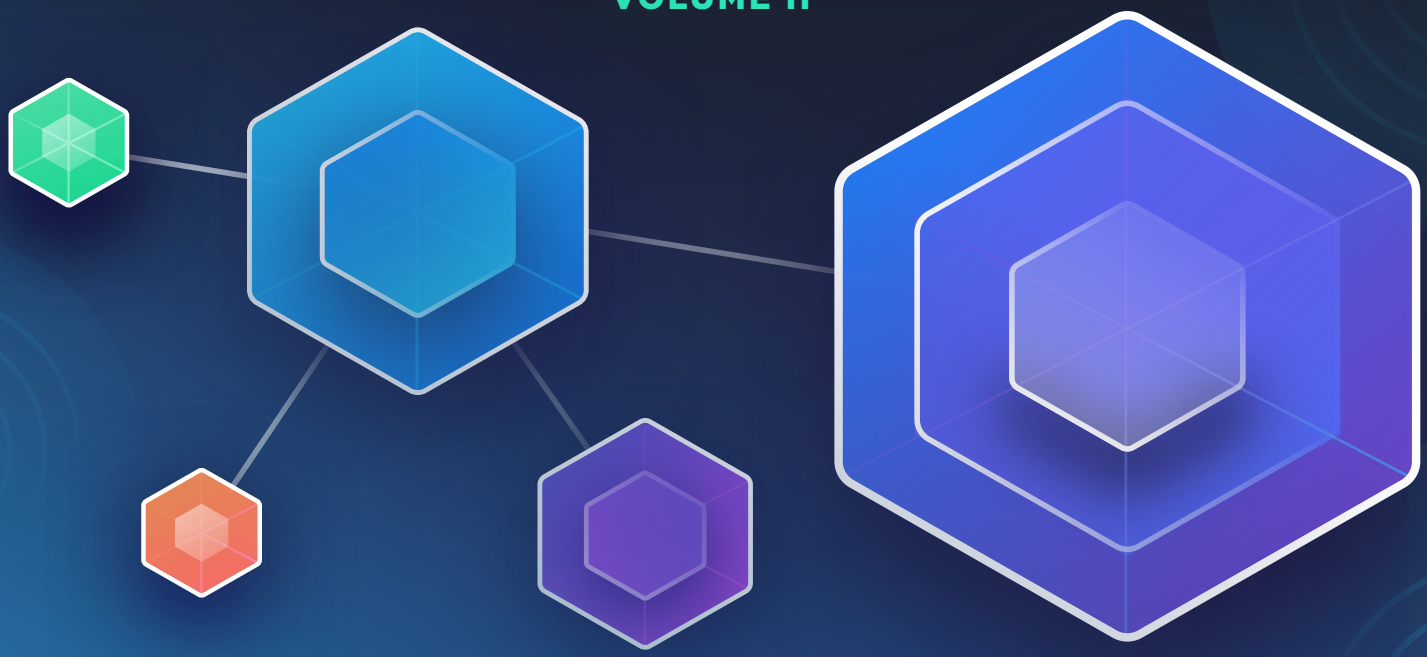
DZone
A DEVADA MEDIA PROPERTY

THE 2018 DZONE GUIDE TO

Microservices

SPEED, AGILITY, AND FLEXIBILITY

VOLUME II



BROUGHT TO YOU IN PARTNERSHIP WITH

ASPEN MESH

aiven

axway

Ballerina
Cloud Native Programming Language

CLOUD FOUNDRY

LIGHTSTEP

okta

TIBCO®

Dear Reader,

It seems like the major themes in software development over the past few years have been focused on making applications as highly scalable as possible. New advancements in technology such as cloud computing and the Internet of Things created the need, and so new tools and strategies such as serverless, functions, containers, and microservices are helping developers achieve those goals.

Microservices, in particular, help developers create scalable applications by breaking up components into different services that can be changed in isolation from the rest of the application. When something breaks, you'll know where to look, and you shouldn't have to worry about breaking the rest of the app as a result. They also have an added benefit of facilitating continuous delivery and continuous deployment practices due to the speed at which developers can make changes. New applications built with microservices can also allow developers to experiment with various languages, technologies, and tools without changing an entire application.

In this year's edition of DZone's *Guide to Microservices: Speed, Agility, and Flexibility*, we surveyed our users about how they were using microservices since our first guide on the subject last year. We've found that developers are still thrilled about microservices and feel that the hype surrounding them is still warranted. We've also seen that there is a steady adoption of microservices in development, though not everyone is playing with microservices in production yet.

You'll also find articles written by several experts across the world and the industry about utilizing microservices on the application frontend, Kafka and other messaging protocols, comparing microservices with functions-as-a-service (FaaS), moving from a monolith to microservices, and case studies from adopting microservices at Google.

Microservices may not be "new," but the challenge of adopting and maximizing their potential is still present. We hope the articles and research in this guide helps clear the air around the subject, and that they encourage you to explore new ways to build better applications.

Happy reading!

By Matt Werner

PUBLICAIONS COORDINATOR, DEVADA

Table of Contents

DZone is...

BUSINESS & PRODUCT
MATT TORMOLLEN, CEO
MATT SCHMIDT, PRESIDENT
JESSE DAVIS, EVP, TECHNOLOGY
KELLET ATKINSON, MEDIA PRODUCT MANAGER

SALES
CHRIS BRUMFIELD, SALES MANAGER
JIM DYER, SR. ACCOUNT EXECUTIVE
ANDREW BARKER, SR. ACCOUNT EXECUTIVE
BRETT SAYRE, ACCOUNT EXECUTIVE
ALEX CRAFTS, KEY ACCOUNT MANAGER
SEAN BUSWELL, SALES DEV. REP.
JORDAN SCALES, SALES DEV. REP.
DANIELA HERNANDEZ, SALES DEV. REP.

PRODUCTION
CHRIS SMITH, DIRECTOR OF PRODUCTION
ANDRE POWELL, SR. PRODUCTION COORD.
G. RYAN SPAIN, PRODUCTION COORD.
BILLY DAVIS, PRODUCTION COORD.
NAOMI KROMER, SR. CAMPAIGN SPECIALIST
JASON BUDDAY, CAMPAIGN SPECIALIST
MICHAELA LICARI, CAMPAIGN SPECIALIST

MARKETING
SUSAN WALL, CMO
AARON TULL, DIR. OF MARKETING
SARAH HUNTINGTON, DIR. OF MARKETING
WAYNETTE TUBBS, DIR. OF MARKETING COMM.
ASHLEY SLATE, SR. DESIGN SPECIALIST
KRISTEN PAGÀN, MARKETING SPECIALIST
COLIN BISH, MARKETING SPECIALIST
LINDSAY POPE, CUSTOMER MARKETING MAN.
SUHA SHIM, ACQUISITION MARKETING MAN.

EDITORIAL
SUSAN ARENDT, EDITOR-IN-CHIEF
MATT WERNER, PUBLICATIONS COORDINATOR
SARAH DAVIS, PUBLICATIONS ASSOCIATE
KARA PHELPS, CONTENT & COMMUNITY MANAGER
TOM SMITH, RESEARCH ANALYST
MIKE GATES, CONTENT TEAM LEAD
JORDAN BAKER, CONTENT COORDINATOR
ANNE MARIE GLEN, CONTENT COORDINATOR
ANDRE LEE-MOYE, CONTENT COORDINATOR
LAUREN FERRELL, CONTENT COORDINATOR
LINDSAY SMITH, CONTENT COORDINATOR

Executive Summary

BY KARA PHELPS - CONTENT AND COMMUNITY MANAGER, DEVADA

Early adopters like Netflix and Amazon first began experimenting with microservices nearly a decade ago. Since then, their popularity has grown exponentially. Rather than relying on monolithic applications that are difficult to maintain and scale, microservices break down large tasks into simple, independent processes that communicate with each other via APIs. Developers adopt microservices architectures to help solve common, complex issues like speed and scalability, while also supporting continuous testing and continuous delivery.

To see what's changed in the microservices space this year, we surveyed 548 tech professionals on how and why they use microservices, the challenges they face in developing microservices, and the frameworks and tools they find most helpful.

The Web Apps and the Enterprise Business Apps

DATA Among survey respondents who reported building web applications and services, 43% said they use microservices in development and production. 30% said they are considering them. Among survey respondents who are building enterprise apps, 36% said they use microservices in development and production. 28% said they are considering them.

IMPLICATIONS Most organizations are either considering implementing microservices, or already have. Microservices have a wide range of applications, addressing many of the more troublesome issues faced in modern IT.

RECOMMENDATIONS If you're planning to develop a new application, whether it's web-based software-as-a-service (SaaS) or designed for the enterprise, microservices may be the best way to future-proof it. Still, they aren't a one-size-fits-all solution — among survey respondents who said they aren't interested in microservices, 58% said it was due to a lack of applicable use cases. The rest attributed their disinterest to a lack of knowledge on the subject or a lack of training.

Microservices By Any Other Language

DATA Among survey respondents working at companies with Java ecosystems, 52% said using a microservices architecture has made their job easier, compared to 40% who work with JavaScript, 26% who work with Node.js, and 21% who work with Python. Among respondents developing web apps, 83% said Java is the programming language that best supports microservices development. Among those developing enterprise business apps, 85% agreed.

IMPLICATIONS Java is the most popular programming language ecosystem used at companies where survey respondents work, sitting at 85%. Likewise, it's also one of the most commonly used worldwide. Java is a mature language with a wide variety of resources — libraries are more readily available, and tools for microservice orchestration abound. In addition to developers' familiarity with Java, these factors often make the choice a straightforward one.

RECOMMENDATIONS The challenge of picking the right programming language for microservices isn't much different than it is for other service-oriented architectures. Talent and support are major considerations. If your organization is already committed to the Java ecosystem, it makes sense to continue using Java. Java 8 took some major steps forward in its support for microservices development, as well.

The Not-So-Rosy Side, and What to Do About It

DATA When asked what challenges they face when building apps with microservices, 37% of survey respondents building web apps and 39% of those building enterprise apps said that monitoring was the most significant issue. When asked what challenges they face when refactoring legacy apps to a microservices architecture, responses were slightly more diverse. For those developing web apps, the largest share went to "overcoming tight coupling" at 28%, closely followed by "finding where to break up monolithic components" at 27%. Those developing enterprise apps also named those two categories the most frequently, each at 31%.

IMPLICATIONS Monitoring microservices is a more complex task than monitoring monolithic applications due to multiple potential points of failure. Systems degrade, and any slowdowns in performance could impact your app's dependencies. When it comes to refactoring legacy apps, tasks like updating databases for loose coupling and deciding how to divide capabilities among microservices are crucial and resource-intensive.

RECOMMENDATIONS This all might sound a bit daunting, but don't worry! You can choose from a wide variety of tools and resources designed to help with these common problems. Application performance monitoring and structural code analysis can be automated. Check out the Solutions Directory in this Guide for a convenient place to start your research.

Key Research Findings

BY JORDAN BAKER - CONTENT COORDINATOR, DEVADA

DEMOGRAPHICS

For the 2018 DZone Guide to Microservices, we surveyed developers, architects, and technologists from across the software and IT industry. We received 682 responses with a 79% completion rate. Based on these numbers, we've calculated the margin of error for this survey to be 2%. Below is a brief look into the demographics of our survey takers.

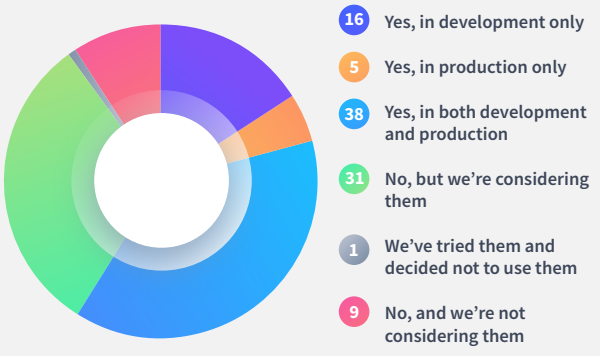
- 43% live in Europe, 23% in the USA, and 10% in South Central Asia.
- 43% work for companies headquartered in Europe and 32% work for US-based companies.
- Most survey takers work for enterprise-sized organizations.
  - 28% work for organizations sized 100-999 employees.
  - 18% work for organizations sized 1,000-9,999 employees.
  - 18% work for organizations sized 10,000+ employees.
- Respondents were split into three main job categories:
  - 37% work as developers/engineers.
  - 25% serve as developer team leads.
  - 20% work as architects.
- 82% work on developing web applications/services, 46% develop enterprise business applications, and 14% develop high-risk software.
- There were five main programming language ecosystems reported:
  - 85% Java
  - 70% JavaScript (client-side)
  - 38% Node.js
  - 34% Python
  - 33% C#
- The average respondent has 17 years of experience in the software industry.

THE WHO, WHEN, AND WHY OF MICROSERVICES

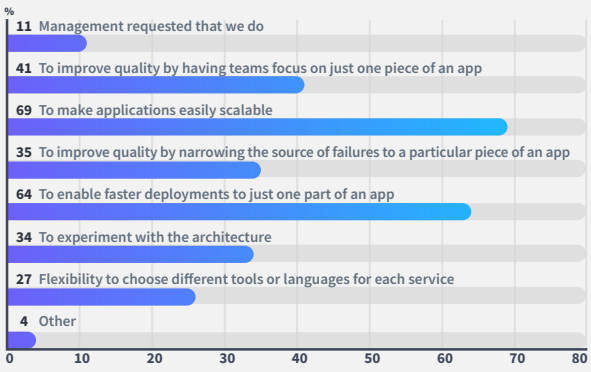
59% of respondents reported using microservices in some capacity, though where they implemented microservices in the SDLC proved somewhat variant. 38% use microservices in both development and production, while 16% use them in development only, and just 5% use microservices in production only. If we compare these numbers to the two most prominent types of developers in this survey (web app and enterprise business app), we see that using microservices in both development and production proved more popular among web developers. 43% of web app developers (versus 36% of enterprise business app developers) reported using microservices in both dev and prod.

Comparing the data regarding where in the SDLC respondents use microservices to the data regarding language ecosystems used by respondents, web development technologies come to the fore. 50% of respondents who work with the Node.js ecosystem reported using microservices in both development and production; this proved the highest percentage of any language ecosystem. Following Node.js, 47% of those who work with the Python ecosystem use microservices in both dev and prod, while 43% of client-side JavaScript developers and 42% of Java developers use microservices architecture in both environments.

ARE YOU CURRENTLY USING A MICROSERVICES ARCHITECTURE FOR ANY OF YOUR APPLICATIONS?



WHY ARE YOU USING MICROSERVICES?



Now that we know who is using microservices, and where in the SDLC microservices are prominent, let's examine why developers use this architectural pattern. At 68%, microservices' ability to make applications more easily scalable proved the most popular answer to this question among respondents. In a close second, 64% of survey takers told us that they use microservices to enable faster deployments to just one part of an application. These two uses of microservice architectures were by far the most popular, with a statistical differential of 23% separating enabling faster deployment from the third most popular use case, improving quality by having teams focus on just one piece of an app. The other notable uses, each chosen by about a third of respondents, were to improve quality by narrowing down the source of failures to a particular piece of an app (35%) and to experiment with the architecture (34%).

Interestingly, if we compare this data to the type of programming language ecosystems respondents use, we find that each language ecosystem lends itself to a different benefit of microservices. Among those working in the Java ecosystem, the most popular use case for microservices (chosen by 86%) was experimenting with architecture. For those working in the JavaScript ecosystem, improving by having teams focus on just one piece of an app (chosen by 72%) came out as the most popular option. For Node.js (45%) and Python (28%) ecosystem developers, making applications easily scalable won out.

Due to the benefits delineated above, 68% of respondents told us they feel microservices architecture has made their job easier, and 72% said they think the excitement around microservices is indeed warranted.

While a large majority of respondents enjoy working with microservices, we did have a small faction (99 respondents) who reported to be uninterested in microservices. The most notable reason being a lack of applicable use cases (58%). Other responses included a lack of knowledge on the subject (34%) and a lack of training (25%). All three of these critiques exhibit a rather interesting year-over-year pattern. While microservices detractors decreased from 125 respondents in 2017 to the 99 in 2018 noted earlier, the percentage of those who feel microservices has a lack of applicable use cases among this group went up by 19%. But also with the passing of a year, the percentage who claimed a lack of knowledge on the subject fell by 4%.

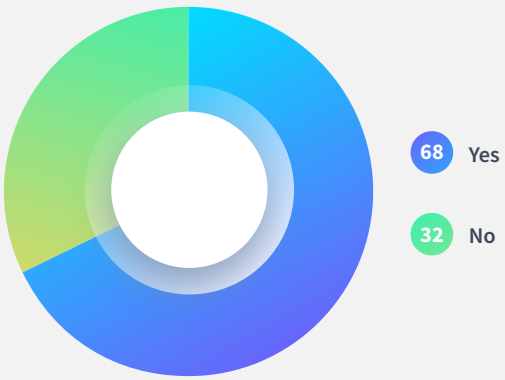
DEVELOPING MICROSERVICES

Among our general survey population, 70% (434 respondents) reported implementing DevOps processes such as continuous delivery. Of those 434 respondents, 39% told us they use microservice architecture in both development and production, 16% use microservices in development only, and another 6% use microservices in production only. Adding these all up, 61% of the 434 respondents who work with DevOps processes use microservices in some capacity when developing applications. The mean number of applications respondents are currently running with a microservices architecture came out to five; the highest number of apps reported was 50 and the lowest 0. And, of those respondents who have refactored legacy apps to take advantage of a microservice architecture, the mean number of refactored apps per respondent came out to two, with 15 being the highest number of applications reported.

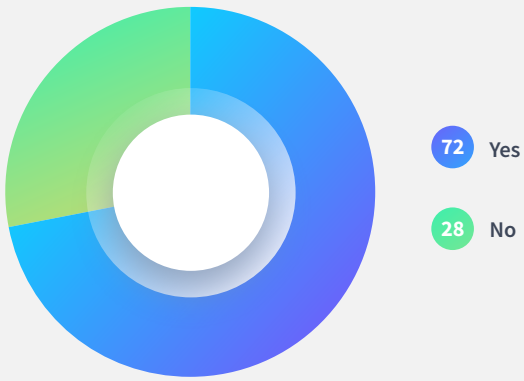
When asked for the language they feel best supports this type of microservices-based development, an overwhelming majority, coming in at 82% of respondents, said Java. The second most popular option was the JavaScript-based runtime environment, Node.js, with a 40% adoption rater among survey takers. And, with a 31% adoption rate, client-side JavaScript and Python also proved rather well-liked. If we compare the adoption rate of these languages to our data on the types of applications that respondents develop (and how respondents choose to secure their microservices), we get some intriguing results.

Among both web application (83%) and enterprise business application (85%) developers, Java proved the most popular language. Python, similarly, was statistically stable between these two user groups, with a 32% adoption rate among web app developers and a 31% use rate among enterprise business devs. When we get to the data on client-side JavaScript and Node.js, however, interesting fluctuations appear. While 45% of web app developers reported using Node.js, only 34% reported

HAS USING MICROSERVICES AS ARCHITECTURE MADE YOUR JOB EASIER?



DO YOU THINK THE EXCITEMENT AROUND MICROSERVICES IS WARRANTED?



using client-side JavaScript for their microservices; among enterprise business devs, 42% reported using Node.js, while only 28% claimed to use JavaScript on the client-side. This is interesting to note, as one of the main advantages of Node.js is the ability to code both server-side and client-side applications in the JavaScript language. And yet, we see far more respondents interested in using Node.js for their microservices-based development than client-side JavaScript.

When we asked respondents how they secure their microservices, 47% said JSON web tokens, 43% reported using OAuth 2, and 28% told us they implement user authentication. When we compare these numbers to the data on the top microservices-friendly languages, we find that JSON web tokens proved more popular among these respondents than among the general survey population. 55% of those who use Node.js for microservices development reported using JSON web tokens, 52% who use JavaScript use JSON web tokens, and 49% who use either Java or Python use JSON web tokens to secure their microservices. User authentication, too, proved more popular among users of these four languages than the general population, while OAuth 2's adoption rates witnessed far less fluctuation.

Despite the popularity of microservices, this architectural pattern comes with its own set of challenges. 58% of respondents reported that monitoring can present an issue when building apps with microservices. Fascinatingly, the second most oft reported challenge of building apps with microservices was changing culture to be open to microservices. 40% of respondents told us cultural shift presents an issue. Though other, more technical, challenges were reported — like changing API contracts (34%) and communicating between microservices (32%) — it appears that organizational structure has become a bigger roadblock to microservice adoption.

TOOLS FOR BUILDING MICROSERVICES

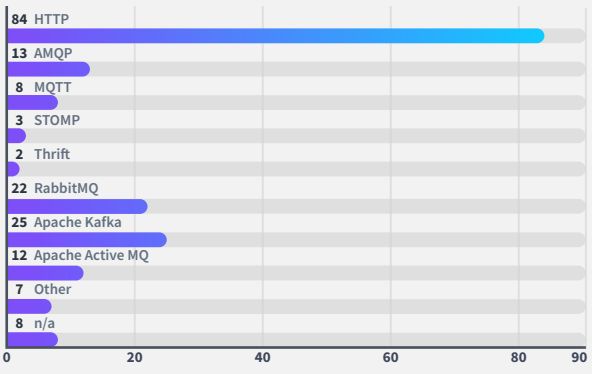
Given that 82% of respondents said they felt Java was one of the languages that best supports microservices, it comes as no surprise that Spring Boot (57%) and Java EE (22%) were the two most used frameworks/tools reported for building microservices. The platforms used to manage microservices once built, however, had a more even spread. 32% of those who microservices management platforms told us they use Istio, 18% use Kong, 17% use Conduit, and 13% used Linkerd.

Looking at the data for communication protocols, three main choices emerged: HTTP (84%); Apache Kafka (25%); RabbitMQ (22%). If we compare this data to the four largest programming language ecosystems delineated in the Demographics section (Java, JavaScript, Python, and Node.js), we find that HTTP remains the first choice for communication protocols among a large majority of respondents. Here's a breakdown of HTTP users per ecosystem: 84% Java; 86% Node.js.; 85% JavaScript; 84% Python. Apache Kafka proved more popular among Python ecosystem developers than anyone else, with a 33% adoption rate. Similarly, RabbitMQ saw a 30% adoption rate among Node.js ecosystem developers, an 8% increase over the general survey population.

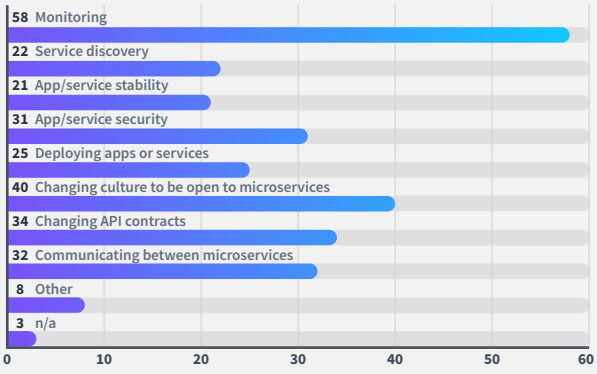
For the environments used to build, host, and deploy their microservices-based applications, over half of respondents (70%) use containers. Container technology factored in heavily in development, with 27% of respondents using containers only in the development stage and 35% using containers in both development and production. Among this group of survey takers doing microservice development in containerized environments, 51% use Kubernetes as their container orchestration tool. To delve deeper into the topics of containerization and container orchestration, head over to the *DZone Guide to Containers: Development and Management*.

Even with all these statistics, developers still appear split on the usefulness of tools and frameworks for microservices development. 60% of respondents told us they feel that tools and frameworks have provided sufficient best practices for working with microservices, with 40% feeling there is still work to be done on this front.

WHAT PROTOCOLS DO YOU USE TO COMMUNICATE BETWEEN SERVICES?



WHAT CHALLENGES DO YOU FACE WHEN BUILDING APPS WITH MICROSERVICES?



How Kafka Solves Common Microservice Communication Issues

BY HANNU VALTONEN
VP PRODUCT, **AIVEN**

QUICK VIEW

01. There are multiple approaches to handling microservices communication.

02. HTTP REST APIs are often simple to start with but become complex as systems grow.

03. Having centralized message queues simplify certain aspects of microservices architecture, but have scaling limitations.

04. Apache Kafka decouples message senders from receivers and extends message queuing systems towards being able to handle streaming data.

Microservices have communicated with each other in different ways since their inception. Some have preferred to use HTTP REST APIs, but these come with their own queuing issues, while some have preferred older Message Queues, like RabbitMQ, which come with scaling and operational concerns.

Kafka-centric architectures aim to solve both problems.

In this article, I'll explain how Apache Kafka improves upon the historical HTTP REST API/message queuing architectures used in microservices and how it further extends their capabilities.

A TALE OF TWO CAMPS

The first camp in our story is one where communication is handled by calling other services directly, often over HTTP REST APIs or some other form of Remote Procedure Calls (RPC).

The second camp, while borrowing from the Service-Oriented Architecture (SOA) concept of an Enterprise Service Bus, uses an intermediary that's in charge of talking to the other services and operates as a message queue.

This role has often been accomplished by using a message broker like RabbitMQ. This way of communicating removes much of the communication burden from the individual services at the cost of an additional network hop.

MICROSERVICES USING HTTP REST APIS

HTTP REST APIs are a popular way of performing RPC between services. Its main benefits are simplified setup in the beginning and relative efficiency in sending messages.

However, this model requires its implementor to consider things like queuing and what to do if the amount of incoming requests exceeds the node's capacity. For example, if you assume that you have a long chain of services preceding the one exceeding its capacity, all of the preceding services in the chain will need to have the same sort of back pressure handling to cope with the problem.

Additionally, this model requires that all of the individual HTTP REST API services need to be made highly available. In a long processing pipeline made of microservices, none of the microservices can afford to lose all of their component parts and this only works as long as at least one process from any given group is still operating normally.

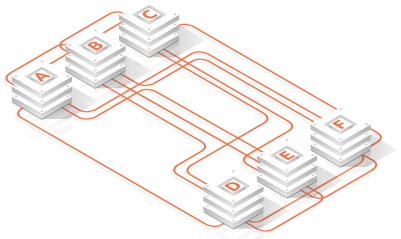
This often requires load balancers to be put in front of these microservices. Also, service discovery is often a must since the different microservices need to find out where to call in order to communicate with each other.

One advantage of this model is that it provides for potentially excellent latency, as there are few middle-men in a given request path and those components, like web servers and load balancers, are highly performant and thoroughly battle-tested.

General dependency handling between different RPC microservices often quickly becomes more complex and eventually starts to slow development efforts. New approaches like Envoy proxy, which provides a service mesh, have also been introduced to solve these issues.

Although these address many of the load balancing and service discovery problems with the model, they necessitate an increase in the system's overall complexity from just using simple, direct RPC calls.

Many companies start with only a few microservices talking to each other but eventually their systems grow and become more complex, creating a *spaghetti of connections* between each other.



MESSAGE QUEUES

Another way of building microservices communications revolves around the use of a message bus or message queuing systems. Old-fashioned Service-Oriented Architecture called these enterprise service buses (ESBs). Often, they've been message brokers like RabbitMQ or ActiveMQ.

Message brokers act as a centralized messaging service through which all of the microservices in question talk to each other, with the messaging service handling things like queuing and high availability to ensure reliable communication between services.

By supporting message queuing, messages can be received into a queue for later processing instead of dropping them when processing capacity is maxed out during peak demand.

However, many message brokers have demonstrated scalability limitations and caveats around how they handle message persistence and delivery in clustered environments.

Another large topic of conversation around message queues is how they behave in error cases, e.g. whether the message delivery is guaranteed to happen at least once, at most once, etc.

The semantics chosen depend on the message queue implementation, meaning you will have to familiarize yourself with its message delivery semantics.

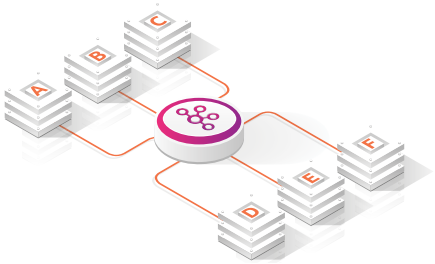
Additionally, adding a message queue to an architecture adds a new component to be operated and maintained, and network latency is also increased by having one additional network hop for sent messages, which incurs extra latency.

Security matters are slightly simplified in this model by the centralized nature of the Access Control Lists (ACLs) that can be used with message queuing systems, giving central control over who can read and write what messages.

Centralization also brings some other benefits in regard to security. For example, instead of having to allow all services to connect to each other, you can allow only connections to the message queue service instead and firewall the rest of the services away from each other, decreasing the attack surface.

ADVANTAGE OF A NEW KAFKA-CENTRIC AGE

Apache Kafka is an event streaming platform created and open-sourced by LinkedIn. What makes it radically different from older message queuing systems is its ability to totally decouple senders from receivers in the sense that the senders need not know who will be receiving the message.



In many other message broker systems, foreknowledge of who would be reading the message was needed; this hampered the adoption of new use cases in traditional queuing systems.

When using Apache Kafka, messages are written to a log-style stream called a topic and the senders writing to the topic are completely oblivious as to who or what will actually read the messages from there. Consequently, it is business as usual to come up with a new use case to process Kafka topic contents for a new purpose.

Kafka is completely agnostic to the payload of the sent messages, allowing messages to be serialized in arbitrary ways, though most people still use JSON, AVRO, or Protobufs as their serialization format.

You can also easily set ACLs to limit which producers and consumers can write to and read from which topics in the system, giving you centralized security control over all messaging.

It's common to see Kafka employed as a receiver of firehose-style data pipelines where the data volumes are potentially enormous. For example, Netflix reports that they process over two trillion messages a day using Kafka.

One of the important properties that consumers have is that when message load increases and the number of Kafka consumers changes due to faults or increased capacity, Kafka will automatically rebalance the processing load between the consumers. This moves the need to handle high availability explicitly from within the microservices to the Apache Kafka service itself.

The ability to handle streaming data extends Kafka's capabilities beyond operating as a messaging system to a streaming data platform.

To top it all off, Apache Kafka provides fairly low latency when using it as a microservices communication bus, even though it incurs an additional network hop for all requests.

This powerful combination of low latency, auto-scaling, centralized management, and proven high availability allows Apache Kafka to extend its reach beyond microservices communications to many streaming real-time analytic use cases that you've yet to imagine.

HANNU VALTONEN is VP Product at Aiven, a DBaaS company, and an open source expert. He has decades of experience in designing large-scale distributed systems and their assorted communications architectures.

Monolith to Microservices with the Strangler Pattern

BY SAMIR BEHARA
SOLUTION ARCHITECT, **EBSCO INDUSTRIES**

QUICK VIEW

01. Due to complexity, difficulty of scaling, and technical debt, maintaining monolithic legacy applications can be incredibly difficult.

02. When moving from a monolith to microservices, the Strangler Pattern allows you to replace application functionality step-by-step, rather than all at once.

03. Learn how to implement the Strangler Pattern in your refcatoring efforts using the three-step transform, co-exist, and eliminate pro

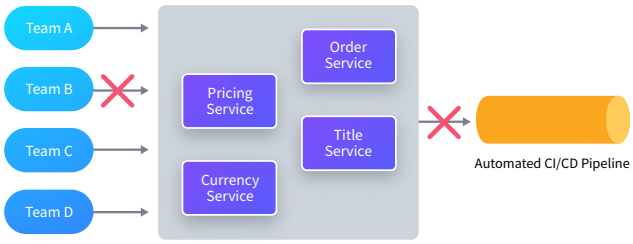
Working in a legacy codebase has its own challenges. With the advent of new technologies, organizations have found it difficult to continue delivering and staying competitive with their monolithic legacy applications. To keep up with the market demand, its necessary to continuously evolve your applications. Transforming your current legacy applications into a number of small microservices seems to be the best approach.

Maintaining a legacy application is cumbersome and often leads to additional work because of multiple reasons:

- Lack of unit tests
- Violation of the Single Responsibility Principle
- High complexity leads to more time spent in maintenance activities
- Inability to scale individual components to meet increased demand
- Tight coupling between components makes it difficult to deploy regularly
- Technical debt is accumulated over time and makes future development difficult

Consider a scenario where you have multiple teams working on a monolithic application. If Team B deploys bad code to a lower environment, it is going to block all the remaining teams from pushing any of their code changes until Team B pushes a code fix. Since you are dependent on a single pipeline to deploy changes from multiple teams, there is a high chance of one team blocking the other, causing delays in feature delivery to the business.

DEPLOYING MONOLITHIC APPLICATIONS



I have worked with clients on their mission-critical insurance and manufacturing applications operating on older technologies with manual deployments. Even though there were multiple teams working on making changes to cater to new business requirements, we were not able to deploy the monolithic application regularly. We used to have monthly releases since manual testing of the new changes was time-consuming and also critical in an integration/staging environment. Most of the time, even the monthly deployments could not be done because new defects were identified during the testing phase a couple of days before the production deployment.

Over the years, the code complexity for these applications has grown and components were tightly-coupled with each other, which makes it difficult to implement good automated testing around the codebase. Making a simple change in one class has the potential to break an existing functionality implemented in an interconnected module. Rather than relying on failing tests, we have to depend upon a few key people who have been working on these systems for an extended period of time. This creates dependencies and delays in getting changes reviewed.

Finally, when everyone realized that this was not a sustainable model to deliver business features, we decided to make architectural changes to decompose the monolithic application into smaller, loosely-coupled services. This would enable teams to deploy changes to production more frequently.

REFACTOR THE MONOLITH OR REBUILD IT FROM SCRATCH?

Rewriting a large monolithic application from scratch is a big effort and has a good amount of risk associated with it. One of the biggest challenges for us was having a good understanding of the legacy system. We did not want to carry forward the technical debt associated with the legacy system into our new modern system.

Also, if you go ahead with rewriting the monolith from scratch, you cannot start using the new system until it is complete. You are in a corridor of uncertainty until the new system is developed and functioning as expected. Depending upon the size and complexity of the application, this might take over a year in a lot of scenarios. During this period, since you are busy developing the new system, there are minimal enhancements or new features delivered on the current platform, so the business needs to wait to have any new feature developed and pushed out of the door.

The Strangler Pattern reduces the above risk. Instead of rewriting the entire application, you replace the functionality step by step. The business value of new functionality is provided much quicker. If you follow the Single Responsibility Principle and write loosely-coupled code, it makes future refactoring work very simple.

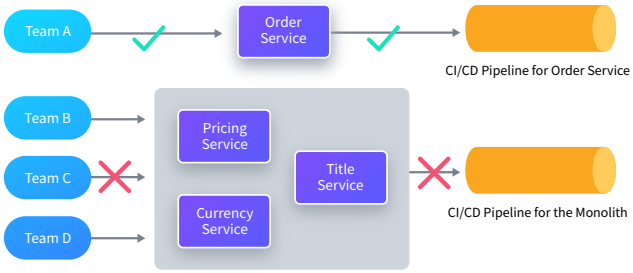
WHAT IS THE STRANGLER PATTERN?

The Strangler Pattern is a popular design pattern to incrementally transform your monolithic application into microservices by replacing a particular functionality with a new service. Once the new functionality is ready, the old component is strangled, the new service is put into use, and the old component is decommissioned altogether.

Any new development is done as part of the new service and not part of the Mmonolith. This allows you to achieve high-quality code for the greenfield development. You can follow Test-Driven Development for your business logic and integrate SonarQube with your deployment pipeline to prevent any technical debt from accruing.

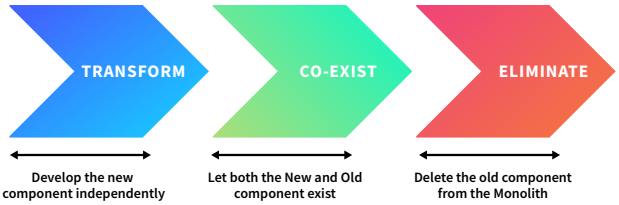
In the below diagram, the Order Service is eventually strangled from the monolith into an independently deployable service with its own CI/CD pipeline. Team A is now not dependent upon any issues with other teams.

STRANGLER PATTERNS IN ACTION



You need to have processes in place to streamline this transition from monolith to microservices. To implement the Strangler Pattern, you can follow three steps: Transform, Coexist, and Eliminate.

TRANSFORM AND ELIMINATE PATTERN



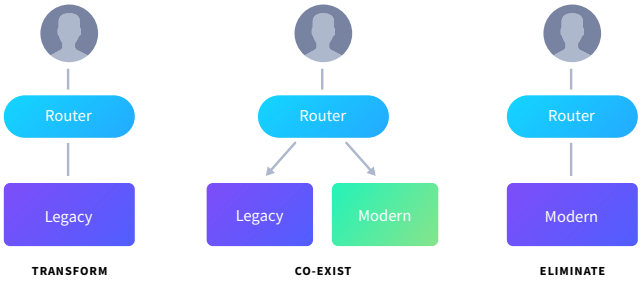
You can develop a new component, let both the new and the old component exist for a period of time, and finally terminate the old component.

Based on the diagram below, the steps can be summarized:

- Initially, all application traffic is routed to the legacy application.
- Once the new component is built, you can test your new functionality in parallel against the existing mono-lithic code.

- Both the monolith and the newly built component need to be functional for a period of time. Sometimes the transitional phase can last for an extended duration.
- When the new component has been incrementally developed and tested, you can get rid of the legacy monolithic application.

TRANSFORM AND ELIMINATE PATTERN



HOW DO YOU SELECT WHICH COMPONENTS TO STRANGLE/REFACTOR FIRST?

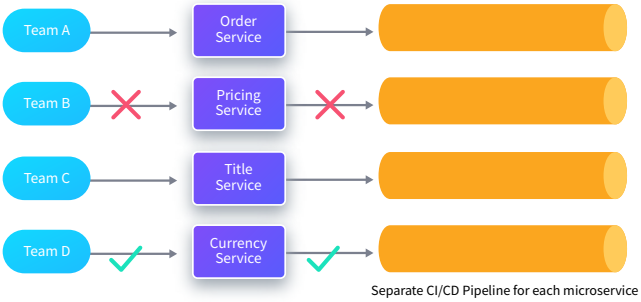
- If you are following the Strangler Pattern for the first time and are new to this design pattern, playing it safe and selecting a simple component is not a bad option. This will ensure that you gather practical knowledge and acclimatize yourself to the challenges and best practices before strangling a complex component.
- If there is a component which has good test coverage and less technical debt associated with it, starting with this component can give teams a lot of confidence during the migration process.
- If there are components which are better suited for the cloud and have scalability requirements, then start with one of those components.
- If there is a component which has frequent business requirements and hence needs to be deployed a lot more regularly, you can start with that component. This will ensure that you don't have to redeploy the entire monolithic application regularly. Breaking it into a separate process will allow you to independently scale and deploy the application.

The cloud migration journey is not an easy one and you will bump into multiple hurdles. The Strangler design pattern assists you in making this journey a bit painless and risk-free since you are dealing with small components at one time. It's not a big undertaking when you plan to do the migration in increments and small pieces.

Reducing the complexity of an application enables you to deliver business features faster. It also allows you to scale your application based on increasing load. Having an automated CI/CD pipeline makes it a lot easier to deploy the microservices and can make the transition from monolith to microservices much smoother.

When you are finally able to decompose your monolithic application into a number of microservices, the representation will look like the diagram below. Each microservice has its own data store and CI/CD pipeline.

MICROSERVICES DEPLOYMENTS



CONCLUSION

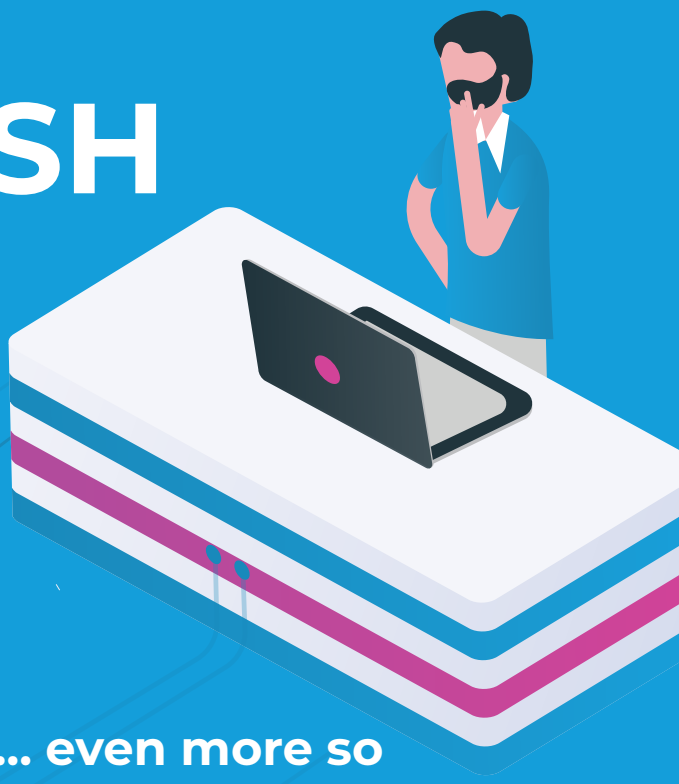
Transforming your existing legacy monolithic application into cloud-native microservices is a nice end goal to have, but the journey is challenging and needs to be well architected and planned. In this article, we discussed a design pattern called the "Strangler Pattern," which can assist you in this journey. Teams can continue delivering business value by doing all greenfield development as part of new services and incrementally migrate from monolith to microservices. However, keep in mind that strangling a monolith is not a quick process and may take some time.

Please let me know if you have any questions and I would be happy to discuss them.

SAMIR BEHARA is a Solution Architect with EBSCO Industries and builds cloud native solutions using cutting edge technologies. He is a Microsoft Data Platform MVP with over 12 years of IT experience. Samir is a frequent speaker at technical conferences and is the Co-Chapter Lead of the Steel City SQL Server UserGroup. He is the author of www.dotnetvibes.com.

ASPEN MESH

SERVICE MESH SIMPLIFIED



Microservices are complex
Managing them in the enterprise... even more so
We did the heavy lifting, so you don't have to

ENTERPRISE READY

Everything you expect from Istio plus additional policy, analytics, and RBAC capabilities

FULLY SUPPORTED

Our fanatical engineers help you overcome any challenges, ensuring success on your microservices journey

EMPOWER YOUR TEAMS

Thrill your developers with a powerful platform that provides the scalability and stability they need

For Kubernetes.
Built on Istio.
Any questions?
aspenmesh.io

TRY IT FOR FREE

The Importance of RBAC for Microservices in the Enterprise

Microservice architectures come with huge upsides. They enable enterprises to develop more quickly, lead to higher availability, and make it easier to scale. Microservices also present a unique set of challenges, of which one of the most pressing is managing large, distributed development teams contributing to a single application.

Large organizations usually consist of several independent development teams contributing code to a single project. This is advantageous as it allows them to act autonomously and decide what technologies, frameworks, and tools are best for their domain. The downside is that several different teams, composed of many developers, require widely varying levels of access. Access control that provides an appropriate separation of concerns is critical for large organizations. The ideal balance is providing the lowest level of access that enables developers to see and control all that they need to.

Role-based access control solves the challenges that come with needing to provide varying levels of access to distributed teams in the enterprise. Providing RBAC is a task made easy with a service mesh. Service mesh enables platform owners to ensure app developers can only write policy for their own apps so that they can move quickly without impacting other teams.

To address the pressing need for enterprises to more effectively manage microservices, continue speeding development, and protect network integrity, Aspen Mesh provides advanced service mesh policy and RBAC features. Try them out for free by signing up for Aspen Mesh beta access.

WRITTEN BY ZACH JORY
HEAD OF MARKETING, ASPEN MESH

PARTNER SPOTLIGHT

Aspen Mesh

ASPEN MESH

Aspen Mesh makes service mesh easy with an enterprise-ready distribution of Istio

PRODUCT
Service Mesh

OPEN SOURCE?
No

RELEASE SCHEDULE
Continuous

CASE STUDY

Aspen Mesh is working with a US-based bank as they're migrating to a microservice architecture so their development teams can move more quickly to create solutions for emerging customer needs.

Shortly after starting this this transition, this bank found that they didn't have the visibility they were used to with the tooling they had built up around their monolith. They struggled to identify, understand, and address performance issues. They weren't sure how to understand their security posture and identify vulnerabilities.

They started testing Istio, and found it powerful, but complicated.

They decided to work with Aspen Mesh to harness the power of Istio with the ease of a fully-supported enterprise distribution.

STRENGTHS

- Intuitive UI
- Advanced policy capabilities
- Analytics and alerting
- Multi-cluster/multi-cloud
- Fully supported

WEBSITE aspenmesh.io | **TWITTER** @AspenMesh | **BLOG** aspenmesh.io/blog

Lessons From the Birth of Microservices at Google

BY DANIEL "SPOONS" SPOONHOWER
CO-FOUNDER, LIGHTSTEP

QUICK VIEW

01. As hardware started to fail more and more, Google needed to design applications differently.

02. Projects like GFS, BigTable, and more were born from this need.

03. Review some of the lessons Google learned from these innovations.

04. Understand why you're adopting microservices, beware of giant dashboards, and be sure to monitor everything.

Our story begins in the early 2000s when Google's Search and Ads products were really taking off. At the time, the conventional wisdom suggested that building a reliable internet service meant buying reliable — and expensive — servers. To keep costs down, Google engineering teams switched to commodity hardware. While this approach saved on cost, these servers were (as you might expect) highly unreliable. As hardware failure became the common case rather than the exception, Google engineers needed to design accordingly.

Today, we might start by going to GitHub to look for code that others have developed to tackle similar problems, but in 2001, this simply wasn't an option. While there were strong communities around projects like Linux and the Apache Web Server, these projects were mostly focused on the software running on a single machine. As a result, Google had no choice but to build things from scratch.

What followed was an explosion of now-legendary infrastructure projects, including GFS, BigTable, and MapReduce. These systems, along with widely used products like Google Search and Ads, had several characteristics that we now associate with microservice architectures:

- Horizontal scale points that enabled the software to scale across thousands of physical machines
- Reusable code for RPC, service discovery, load balancing, distributed tracing, and authentication
- Frequent and independent releases

However, even if Google arrived at the results we are hoping to achieve with microservices, should we follow the same path?

THE LESSONS

LESSON 1: KNOW WHY (YOU'RE ADOPTING MICROSERVICES)

The most important reason to adopt microservices has nothing to do with how your software scales: microservices are useful because they address engineering management challenges. Specifically, by defining service boundaries between teams and letting them work independently, teams will need to communicate less and can move more quickly.

The architecture developed for Google Search and Ads, while superficially similar to microservices, was really there to support the scale needed to index and search the web while serving billions of requests every second. This scale came at a cost in terms of features, and there were hundreds of other smaller teams at Google who needed tools that were easy to use, and who didn't care about planetary scale or tiny gains in efficiency.

As you are designing your infrastructure, consider the scale of what you need to build. Does it need to support billions of requests per second... or "just" millions?

LESSON 2: SERVERLESS STILL RUNS ON SERVERS

Many organizations are considering skipping microservices and moving straight to a "serverless" architecture. Of course, serverless still run on servers, and that has important consequences for performance. As Google built out its infrastructure, measuring performance was critical. Among the many things that Google engineers measured and used are two that are still relevant to serverless:

- A main memory reference is about 100 nanoseconds.
- A roundtrip RPC (even within a single datacenter) is about 500,000 nanoseconds.

This means that a function call within a process is roughly 5,000 times faster than a remote procedure call to another process.

Stateless services can be a great opportunity to use serverless, but even stateless functions need access to data as part their implementation. If some of these data must be fetched from other services (which almost

goes without saying for something that's stateless!), that performance difference can quickly add up. The solution to this problem is (of course) a cache, but caching isn't compatible with a serverless model.

Serverless has its place, especially for offline processing where latency is less of a concern, but often a small amount of context (in the form of a cache) can make all the difference. "Skipping" right to a serverless-based architecture might leave you in a situation where you need to step back to a more performant services-based one.

LESSON 3: WHAT INDEPENDENCE SHOULD MEAN

Above, I argued that organizations should adopt microservices to enable teams to work more independently. But how much independence should we give teams? Can they be *too* independent? The answer is definitely yes, they can.

In adopting microservices, many organizations allow each team to make every decision independently. However, the right way to adopt microservices is to establish organization-wide solutions for common needs like authentication, load balancing, CI/CD, and monitoring. Failing to do so will result in lots of redundant work, as each team evaluates or even builds tools to solve these problems. Worse, without standards, it will become impossible to measure or reason about the performance or security of the application as a whole.

Google got this one right. They had a small number of approved languages and a single RPC framework, and all of these supported standard solutions in the areas listed above. As a result, product teams got the benefits of shared infrastructure, and infrastructure teams could quickly roll out new tools that would benefit the entire organization.

LESSON 4: BEWARE OF GIANT DASHBOARDS

Time-series data is great for determining when there is a regression but it's nearly impossible to use it to determine which service was the cause of the problem. It's tempting to build dashboards with dozens or even hundreds of graphs, one for each possible root cause. You aren't going to be able to enumerate all possible root causes, however — let alone build graphs to measure them.

The best practice is to choose fewer than a dozen metrics that will give you the best signal that something has gone wrong. Even products as large as Google Search were able to identify a small number of signals that really mattered to their users. In this case, perhaps unlike others, what worked for Google Search can work for you.

Ultimately, observability boils down to two activities: measuring these critical signals and then refining the search space of possible root causes. I use the word "refining" because root cause analysis is usually an iterative process: establishing a theory for what went wrong, then digging in more to validate or refute that theory. With microservices, the search space can be huge: it must encompass not only any changes in the behavior of individual services, but also the connections between them. To reduce the size of the search space effectively, you'll need tools to help test these theories, including tracing, as described next.

LESSON 5: YOU CAN'T TRACE EVERYTHING...OR CAN YOU?

Like anyone who has tried to understand a large distributed system, Google engineers needed tools that would cut across service boundaries and give them a view of performance centered around end-to-end transactions. They built Dapper, Google's distributed tracing system, to measure the time it takes to handle a single transaction and to break down and assign that time to each of the services that played a role in that transaction. Using Dapper, engineers can quickly rule out which services aren't relevant and focus on those that are.

Tracing, like logging, grows in cost as the volume of data increases. When this volume increases because of an increased number of business transactions, that's fine — as presumably you can justify additional costs as your business grows. The volume of tracing data also increases combinatorically with the number of microservices, however, so new microservices mean your margins will take a hit.

Google's solution to this problem was simple: 99.99% of traces were discarded before they were recorded in long-term durable storage. While this may seem like a lot to throw out, at the scale of services like Google Search, even as few as 0.01% of requests can provide ample data for understanding latency. To keep things simple, the choice about which traces to keep was made at the very beginning of the request. Subsequent work, like Zipkin, copied this technique.

Unfortunately, this approach misses many rare yet interesting transactions. And for most organizations, it's not necessary to take such a simplistic view of trace selection. They can use additional signals that are available until after the request has completed as part of selection. This means we can put more — and more meaningful — traces in front of developers without increasing costs. To do so will require rethinking the data collection and analysis architecture, but the payoff, in terms of reducing mean-time-to-resolution and improved performance, is well worth the investment.

SUMMARY

While Google built systems that share many characteristics with microservices as they exist today (as well as a powerful infrastructure that has since been replicated by a number of open source projects), not every design choice that Google engineers made should be duplicated. It's critical to understand what led Google engineers down these paths — and what's changed in the last 15 years — so you can make the best choices for your own organizations.

DANIEL "SPOONS" SPOONHOWER is a co-founder at LightStep, where he's building performance management tools for modern software systems. Previously, Spoons spent almost six years at Google where he worked on developer tools as part of both Google's internal infrastructure and Cloud Platform teams. He has published papers on the performance of parallel programs, garbage collection, and real-time programming. He has a PhD in programming languages from Carnegie Mellon University but still hasn't found one he loves.


LIKE
BACON
AND
EGGS...
your apps and Aiven were just meant to be together.
Once you build your apps on top of Aiven, you can focus on developing them while we manage their data infrastructure.
By choosing our platform, you can be confident that your infrastructure will be more integrable, flexible, and secure. Even better, it'll be open-source!
With us, you'll have access to:
• 7 managed open-source data systems
• One-click deployment and integrations
• Hosting across 6 public clouds in over 70 regions
• Single tenant VMs with encryption at transit and rest
Your apps and Aiven: the perfect match.
aiven
visit aiven.io today

State for Microservices

Microservices architectures offer a wide array of benefits over traditional monolithic architectures. The decoupling of individual components allows for independent and much faster development cycles and evolution of system components.

While microservices architectures make application development easier and faster, they also introduce new requirements for handling, communicating, and persisting data which traditional monolithic shared databases do not usually play well with. Instead, microservices should own their own data and use a common messaging platform for interservice communications.

Apache Kafka is an excellent choice for an interservice messaging platform, and there's a variety of excellent open source database engines supporting different use cases for your applications. However, although microservice platforms excel in running stateless systems, they often struggle in handling data persistency efficiently and scalably.

Specialized cloud services, such as Aiven data cloud, are often a better choice for handling stateful data for microservices.

Thus, integrating and managing those database engines in the microservice ecosystem remains a tough problem and could feel like a step back in flexibility. Specialized cloud services, such as our Aiven data cloud, are often a better choice for managing databases and other stateful systems for microservices workloads.

Aiven makes the best open source data infrastructure systems like Kafka, Cassandra, and PostgreSQL available as fully-managed services in all public clouds and integrates into existing development workflows, with open APIs and tools like Terraform to make developers' lives easier to allow you to focus on applications, not infrastructure.

WRITTEN BY OSKARI SAARENMAA
CEO, AIVEN

PARTNER SPOTLIGHT

Aiven Kafka

Aiven Kafka provides high-availability and low-latency at scale for your app's most critical workflows.

PRODUCT
Managed distributed cloud data systems

OPEN SOURCE?
Yes

RELEASE SCHEDULE
Continuous

CASE STUDY

Comcast Xfinity Home is Comcast Corporation's (Nasdaq: CMCSA) home security and automation product providing 24/7 monitoring and smarter home security features, as well as management of third-party IoT devices.

With millions of customers using millions of networked devices across many regions, they needed a scalable, durable solution that could maintain sub 50ms end-to-end latency. Apache Kafka's profile made sense, but it's notoriously difficult to manage.

After evaluating vendors according to Comcast's strict requirements, they chose Aiven Kafka because of Aiven's price, support, and performance. Aiven Kafka now forms the backbone for the critical workflows within their asynchronous architecture design driving Xfinity Home.

STRENGTHS

- **Fully automated:** The Aiven platforms handles operations such as balancing, failed-node replacement, and security patches.
- **Service integrations:** Aiven Kafka can be integrated with other Aiven and external services.
- **Management dashboard:** Manage your Kafka topics and ACLs from our UI.
- **Terraform support:** Include your Aiven infrastructure in your Terraform tooling.
- **Secure:** Single tenant VMs with encryption at transit and rest.

CUSTOMERS

• Comcast • Toyota • Atlassian • OVO Energy • Sphero

WEBSITE aiven.io | **TWITTER** @Aiven_io | **BLOG** aiven.io/blog

Introduction to Microservices Messaging Protocols

QUICK VIEW

01. A microservice-based architecture thrives on the autonomy of the microservices and their instant availability to the consumer.

02. Since a microservices-based application could be running across a large variety of servers, hosts, and processes, we see communication tilted towards HTTP (hypertext transfer protocol), TCP (transmission control protocol), and AMQP (advanced message queuing protocol).

03. Replication, propagation, or duplication will help sidestep the synchronicity issue.

BY SARAH ROMAN
CONTENT MANAGER, **ROBUSTWEALTH**

When companies have pulled together an application based on a variety of services, you can expect that they're running the microservice architectural structure. Used primarily for implementation, microservices provide patterns, protocols, and deployment of complex applications. Foundationally, this architectural style subverts many of the problems associated with monolithic scaling, speed, language barriers, and organization.

While there's large-scale adoption of microservices technology due to these reasons, we should home in on two parts of the microservices architecture that is often a stumbling block for developers: communication and messaging.

WHAT'S SO DIFFERENT ABOUT COMMUNICATION IN A MICROSERVICE ARCHITECTURE?

When transitioning from a monolithic application structure to a multi-independent structure, you'll find immediately that calling on other components isn't aligned to a single-process system. Within the single-process system, you would typically call on components using various language methods or via Dependency Injection, like we see with Angular. Since a microservices-based application could be running across a large variety of servers, hosts, and processes, we see communication tilted towards HTTP (HyperText Transfer Protocol), TCP (Transmission Control Protocol), and AMQP (Advanced Message Queuing Protocol). All of these protocols are built out for IPC, or inter-process communication, since they're managing shared data.

So, how does the microservices architecture tackle communication across a distributed, independent set of processes? A few intersecting ways:

- Synchronous protocol
- Asynchronous protocol
- Single receiver
- Multiple receivers

Since services, hosts, and clients communicate differently, microservices-based messaging or communication is built on the intersections of protocols and receivers.

SYNCHRONOUS PROTOCOL

You'll find yourself engaging in a synchronous protocol process every day, as it's built into chat functions, HTTP, instant messaging, and "live" capabilities. It's a data transfer that occurs at regular intervals and is usually dependent on the microprocessor clock, as there needs to be a clock signal between the sender and receiver. You can also understand this as a protocol that requires a primary/replica configuration, as one device has control over another to ensure synchronicity.

ASYNCHRONOUS PROTOCOL

The opposite of synchronous, asynchronous protocol works outside of the constrains of a clock signal and occurs at any time and at irregular intervals. As mentioned above, the synchronous protocol is microprocessor dependent, meaning that these protocols are often used for the processes occurring inside a computer. With asynchronous processes, the lack of this dependence makes them widely-used for

cloud environments and operating systems. There is no need to wait for a response from receiver after sending information.

SINGLE RECEIVER

Implicit in the name, a single receiver setup denotes that each request must be processed by one receiver. So, in the case of data transmission, requests need to be staggered in order to be received, as they cannot be received at the same time.

MULTIPLE RECEIVERS

Multiple receivers can process multiple requests, as each request can be processed by zero to multiple receivers. For example, you'll usually see multiple receivers used in a Saga pattern, as it needs to be asynchronous and simultaneously promote data consistency.

WHAT INTERSECTION IS POPULAR FOR MICROSERVICE-BASED ARCHITECTURE?

As you can see, the architecture relies on combinations of protocols and receivers due to the wide variety of services that need to be consolidated and managed. The most commonly-used combination tends to be single-receiver communication with synchronous protocol, like HTTP or HTTPS, as it's calling on a regular web service. You can imagine its frequency when we think about how Docker enabled the use of containers that can easily run web applications.

WHAT'S THE OVERALL IMPORTANCE OF THIS?

While these discussion points could be somewhat topical for you, maintaining and integrating microservices is really the crux of the matter when it comes to communication. Developers want to be able to maintain the independence of each microservice while seamlessly and asynchronously integrating them into an application.

Synchronous communication among microservices in an application is often viewed as a complication, headache, and, at worst, a death knell. This is due to the fact that a microservice-based architecture thrives on the autonomy of the microservices and their instant availability to the consumer. So, even if another microservice is crippled within the application, there isn't a ripple effect throughout all of the microservices due to synchronous dependencies.

From anecdotal research, developers often are troubled when building out architecture due to data dependencies. Sometimes, you'll find that one microservice needs the data from another microservice within the application. If you were to use synchronous communication to request the data, there's no backup in place if that requested microservice were to suddenly stop working. Instead of just one microservice in an "out-of-order" state, there would now be two (or more).

HOW DO I AVOID SYNCHRONOUS DEPENDENCIES?

Replication and propagation will help sidestep the synchronicity issue. With replication, you can store data in more than one site (like a server). This greatly improves the availability of data and reduces inconsistency. With propagation, you can push and pull the data from server to client, understandably helpful for local access scenarios.

If replication and propagation aren't current routes to be taken, you could also duplicate data across microservices. This works hand-in-hand with domain-model boundaries for each microservice. When working with domain-model boundaries, you're looking to focus in on the scope of business capabilities and create understandable and meaningful separations between services. If you achieve meaningful separation, this will also help boost scalability and domain-driven choices for data and iteration.

WHAT ARE THE POPULAR COMMUNICATION STYLES?

So, after the larger considerations of microservice to microservice communications, what do most people do?

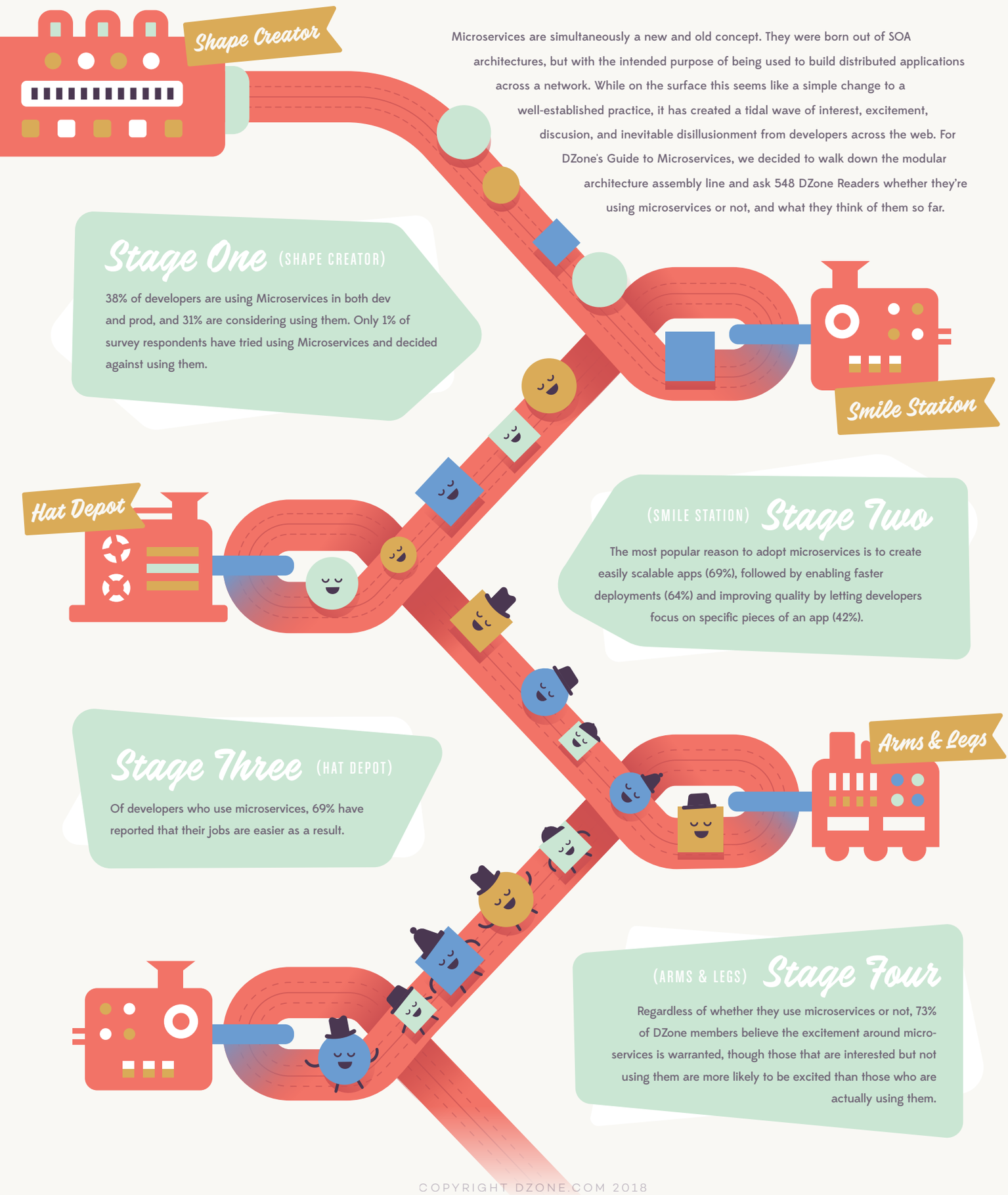
- Use HTTP/REST (synchronous) communication for services outside of the internal microservices structure. The HTTP/REST communication is ideal for external requests, as it's built to easily handle items like real-time interactions from a client.
- Use AMQP (asynchronous) for communication methods within the internal microservices structure.

Ultimately, the microservices architecture follows logical conclusions for the application of messaging and communication protocols within an application. When adopting microservices, there are forward-thinking pieces that you have to make sure you address as you develop and build a multi-faceted application: scalability, infrastructure, and transitions. It's prudent to lay a strong foundation of communication throughout the services, as it helps it grow and affects other parts of the application down the line.

As you broaden your knowledge on microservices and potentially head into development and implementation, you'll want to explore more around creating consistency within the application, exploring the case studies around adopting microservices, and the microservices' impact on the DevOps lifecycle. The microservices-based architecture is extremely agile and powerful, especially when approached after developing the foundational aspects behind communication, understanding the scope of each service, and the front-end needs for clients and consumers.

SARAH ROMAN is currently the Content Manager for the FinTech company RobustWealth. She has a technical background in microcomputers and microcontrollers and has worked on a variety of writing projects that cover how-tos and implementations. She primarily writes articles geared towards technology information and education. Prior to her roles in technical writing and content management, she was a high school English teacher.

Big Things COME IN MICROSERVICES



Microservices are simultaneously a new and old concept. They were born out of SOA architectures, but with the intended purpose of being used to build distributed applications across a network. While on the surface this seems like a simple change to a well-established practice, it has created a tidal wave of interest, excitement, discusion, and inevitable disillusionment from developers across the web. For DZone's Guide to Microservices, we decided to walk down the modular architecture assembly line and ask 548 DZone Readers whether they're using microservices or not, and what they think of them so far.

Stage One (SHAPE CREATOR)

38% of developers are using Microservices in both dev and prod, and 31% are considering using them. Only 1% of survey respondents have tried using Microservices and decided against using them.

(SMILE STATION) Stage Two

The most popular reason to adopt microservices is to create easily scalable apps (69%), followed by enabling faster deployments (64%) and improving quality by letting developers focus on specific pieces of an app (42%).

Stage Three (HAT DEPOT)

Of developers who use microservices, 69% have reported that their jobs are easier as a result.

(ARMS & LEGS) Stage Four

Regardless of whether they use microservices or not, 73% of DZone members believe the excitement around microservices is warranted, though those that are interested but not using them are more likely to be excited than those who are actually using them.

axway IMAGINE SUMMIT 2019
Accelerating the future. Together.

Coming March 4-6
Rosen Centre | Orlando, FL

Modernize the monolith

Break up applications into microservices to innovate faster

- Quickly build microservices and APIs that drive business value
- Mobilize orchestrations as independently scalable microservices
- Target new channels and devices with easy and secure access to enterprise resources

AMPLIFY™ API Builder
TRY IT FOR FREE

axway.com/api-builder-free-trial

Modernizing the Monolith: Accelerate Your Microservices Projects With Axway

Modernizing IT is about breaking down the monoliths of the past so the business can innovate faster and stay competitive and relevant in the future. Implementing microservices efficiently is essential to getting it done.

To build microservices that do one thing and do it well, you'll need to focus on making sure they are independently scalable, quickly deployable, and reliably consistent. Here are three key factors to keep in mind.

1. **Orchestration and mediation deliver scalability and high availability**
   In a microservice-based approach, each microservice owns its model and data so it will be autonomous from a development and deployment point of view. These kinds of systems are complex to scale out and manage, so you absolutely need a solution that lets you quickly orchestrate and mediate APIs if you want to have a production-ready application to target new channels or devices.

2. **Service mesh enables safe and reliable fast inter-service communication**
   To simplify service-to-service communication, use a configurable service mesh for service discovery, load balancing, encryption, authentication and authorization, support for the circuit breaker pattern, and other capabilities.

   By creating a dynamic infrastructure layer for communication at varying degrees of granularity, you can unlock the data trapped in your organization and mesh it up to deliver new business value.

3. **An API-first strategy offers speed and flexibility**
   Don't be afraid to start designing an API before you start implementing the microservice that will consume it. This approach accelerates projects by letting you validate your API design before investing too much in writing the actual microservice.

WRITTEN BY SHRAVANTHI REDDY
DIRECTOR, API AND APP DEVELOPMENT SOLUTIONS, AXWAY

FaaS vs. Microservices

QUICK VIEW

01. Just because FaaS is the latest trend doesn't mean you necessarily need to skip over microservices.

02. There is no "one size fits all" strategy when adopting microservices.

03. You need a pragmatic lens through which to judge and apply microservices and FaaS to your existing technology stack.

BY CHRISTIAN POSTA
CHIEF ARCHITECT, **RED HAT**

In your journey to cloud-native nirvana, you may be adopting microservices architectures for your next-generation applications. You may have had some initial successes with DevOps and automating your builds to help forge a path forward for your microservices journey. After a year or so, you may also be trying to convince the rest of the organization to go down this path as well. Unfortunately, whether anticipated or not, you'll run into issues. Microservices is a complex architectural pattern and distributed systems themselves are difficult to get right. While you march down the path of working through and solving some of these challenges, you will certainly have naysayers and folks who wish to go in a different direction. With how fast technology moves, and with the new shiny options for building services and applications, can you be *sure* your effort to make microservices a successful endeavor for your organization is not in vain?

The simple answer is yes.

These days, "serverless" and "functions as a service" (FaaS) have found themselves at the early side of the hype cycle. Some folks have gone so far to say that serverless and functions are the next evolution of microservices, and that you should just skip the whole microservices architecture trend and go right to serverless. Just as you'd expect, this is a bit hyperbolic; by which you shouldn't be surprised. There's a lot of exciting, new technology available to us as architects and developers to improve our ability to achieve our business outcomes. What we need is a pragmatic lens through which to judge and apply these new technologies. Although as technology practitioners it's our responsibility to keep up with the latest technology, it's equally our responsibility to know when to apply it in the context of our existing technology and IT departments. Let's take a look at a model for understanding how microservices architectures and serverless along with Functions as a Service fit into our toolbox.

First let's understand why we would use a microservices architecture. The main reason you'd opt to use a microservices architecture is to improve the speed at which you're able to make changes to your application when your application's monolithic nature *has become the source of bottlenecks* and impedance for change. All of the other benefits of microservices are derived from that basic premise. Of course, the reason why we'd want to go faster with our changes to our applications is to quickly get new features and functionality out to our customer to *test* whether we can achieve the expected positive outcomes as a result of these changes. If we make changes to our software in an effort to improve our business value and it does not pan out, then we need to quickly know that and move on to try something new. Microservices is an optimization of our application architecture to allow us to move faster and get those changes out quicker.

Most organizations will find that some percentage of their custom-built applications will benefit from an iterative progression to a microservices architecture. These applications will benefit from this change, and as architects and developers we should not get discouraged with the hurdles we'll see. The important thing here is to identify and measure improvement indicators, like how many changes we can make to the software per day, how safely we're able to make these changes, and so on.

On the other hand, not all applications require a highly complex, decomposed set of services to move faster. On the contrary, when

you're first experimenting with a minimum viable product and you're testing whether your idea has *any* market value, you may opt for a different architecture more amenable to this part of the lifecycle of an application. There's a good chance that for an MVP test, you strike out and there is very little, if any, market value. In that case, you would just throw that MVP application away. You will probably be iterating it very quickly and eliciting feedback from potential users. In this case, the business value is not understood, the domain that you implement in code will be constantly changing (if not disappearing altogether), and you'll gain insight into your code as you go. In this environment, you'll be changing APIs, boundaries, components, etc. that make up your system. Prematurely optimizing all of these components into distributed services with API contracts et. al, will actually *slow* you down. You'll be constantly changing your APIs and components *together* and coordinating with all of your small "two-pizza" teams, which smells of a distributed monolith. You might as well just use a monolith for that and you'll be able to go faster and get farther.

At this point, we see that microservices can be appropriate for some percentage of the application portfolio, while monolith applications make sense for some other percentage. There is no single "one size fits all" strategy. Another angle to consider when building either microservices or monoliths (and it follows whether you're optimizing an existing architecture or building to test ideas) is whether the functionality you need to build already exists either as third-party services or as existing services you own within your enterprise. The idea that we can fully leverage existing services to build our applications without having to procure hardware, install and patch operating systems, and optimize our capacity for the highest expected throughput for the life of the software is what the cloud and its services are really all about. Cloud providers and their partners regularly offer databases, message queues, caches, CDN, etc. and even higher-order functionality like language translation, mapping/geo-spatial coordinate mapping, weather, etc. as on-demand, pay-as-you-go services that can be combined in interesting ways to build applications. When we leverage existing higher-order services without worrying about how to install, provision, and plan for capacity, we are moving towards a "serverless" architecture. Serverless architectures strive to re-use existing *services* without worrying about what it takes to run the service.

Functions as a Service is related to serverless because it allows us to use a compute model (scoped down to that of a single application function) that helps stitch together the various services we may consume to build an application. In this compute model, functions are spun up on-demand and you're billed only for the time a function was running. This fits well with being billed on-demand and pay-as-you-go with the various other services you may consume. Now you can build applications that scale without having to worry about solving all of the difficult technology problems that need to be solved in order to make this happen. It's someone else's (the service provider or cloud provider's) problem. If you're able to outsource a lot of those difficult infrastructure challenges to someone else, focus squarely on the business logic that's differentiating to your business, pay as you go and on demand, you're able to more quickly experiment, try out ideas, and deliver business value for your customers.

Outsourcing this kind of functionality may not always be possible, however. When we decide to leverage cloud services, we give up control of uptime/SLA, feature road-map, bug fixing, regulatory compliance, et. al. to someone else. This may be an appropriate tradeoff for a part of the portfolio or for parts of the organization. Others may not be comfortable doing this.

Serverless doesn't have to be a full "public cloud or nothing" proposition, however. If we look within a single organization, parts of it may be "serverless" to other parts. For example, the "buying" side of a retail operation may provide services to other parts of the organization or even third-party vendors/partners that help others build analytics, recommendations, or other applications using the "buyer" services. With well-defined APIs, a workflow for subscribing and paying for the APIs, you could use microservices/monoliths on your own infrastructure (or cloud) and provide serverless capabilities. In many ways, this is just an evolution of Service-Oriented Architecture (SOA). The big difference when you're doing it yourself (or one part of the organization providing services to another) is that the organization as a whole is not serverless; someone still has to set up, manage, patch the servers and all of the supporting infrastructure.

Ultimately, business decisions, business goals, maturity and capability of an IT organization, and any existing legacy constraints come in to play when we decide what application architecture or technologies we can leverage. If you're going down the path of microservices for the right reasons, don't get distracted by other shiny things. Conversely, you will need to continually invest into the latest technology and skills and know when to use them. Monolith, microservices, and serverless all have their place.

CHRISTIAN POSTA is a Chief Architect of cloud applications at Red Hat and well known in the community for being an author (Istio in Action, Manning, Microservices for Java Developers, O'Reilly 2016), frequent blogger, speaker, open-source enthusiast and committer on various open-source projects including Istio, Apache ActiveMQ, Fabric8, et.al. Christian has spent time at web-scale companies and now helps companies create and deploy large-scale, resilient, distributed architectures - many of what we now call Microservices. He enjoys mentoring, training and leading teams to be successful with distributed systems concepts, microservices, devops, and cloud-native application design.

Ballerina

Cloud Native Programming Language

Ballerina makes it easy to write cloud native applications while maintaining reliability, scalability, observability, and security.

ballerina.io

Eliminating Monolithic ESB and Simplifying Microservices Integration

A microservices-based application typically comprises multiple business capability-oriented services that communicate over the network using diverse communication protocols and patterns. Integrating these services is one of the most challenging tasks in realizing microservices architecture.

Rather than using a conventional, centralized ESB-based approach, we need to build smart-endpoints and dumb pipes to decentralize and disperse the integrations across all the services. When creating service integrations or compositions, we must use different integration styles such as synchronous request-response style messaging (REST, GraphQL, gRPC) and event-driven asynchronous messaging (Kafka, AMQP, NATS). Using a hybrid of synchronous and asynchronous messaging styles is more pragmatic in most use cases. Also, as part of microservices development, we need to create data compositions of multiple business entities that are represented using diverse network-aware data types, such as JSON, XML, ProtoBuf.

These network communications between services have to be implemented in a resilient manner using inter-service communication resiliency patterns such as timeouts, retries, bulk-head, and circuit breakers. Besides the business logic, there are cross-cutting concerns such as security, observability (logging, tracing, metrics, services dependency visualization) that are essential for running microservices in production.

Therefore, to implement microservices, we need a microservices development technology that offers native support for network interactions, network resiliency patterns, network-aware data types, and integration styles, along with support for cross-cutting concerns such a security and observability.

WRITTEN BY KASUN INDRASIRI
DIRECTOR - INTEGRATION ARCHITECTURE, WSO2

PARTNER SPOTLIGHT

Ballerina

Ballerina makes it easy to write microservices that integrate APIs.

Ballerina
Cloud Native Programming Language

PRODUCT	OPEN SOURCE?	NEW RELEASE
Programming language	Yes	Continuously

PRODUCT OVERVIEW

Ballerina is a full-fledged, open source programming language that incorporates fundamental concepts of distributed system integration. The language offers a type-safe, concurrent environment to implement and integrate microservices. By offering first-class support for network programming, it aims to overcome limitations in conventional centralized and configuration-driven approaches to integration, such as those with ESBs, and the complexity of programming languages, with frameworks such as Spring and Node.js, which offer agility but require writing large amounts of complex and boilerplate code to integrate endpoints. Ballerina makes writing and integrating microservices agile and intuitive so that you can focus more on the business capabilities rather than spending time on plumbing your services together. Find out more: ballerina.io/learn.

LANGUAGE DESIGN PRINCIPLES

- Sequence diagrammatic: Every Ballerina program can be displayed as a sequence diagram of its flow.
- Concurrency: The concurrency model is parallel-first since interactions with remote parties always involve multiple workers.
- Type system: Ballerina has a structural type system with primitive, record, object, tuple, and union types.
- DevOps ready: The language includes a unit test framework, build system, documentation generation, dependency management, and versioning, and a way to share modules of reusable code as part of its core distribution.
- Secure by default: Ballerina enforces multiple security checks and secure defaults to prevent critical security vulnerabilities.
- Network-aware: Services, endpoints, and networking concepts are first-class constructs, which are represented graphically as well as textually.

WEBSITE ballerina.io | **TWITTER** @ballerinalang | **BLOG** blog.ballerina.io

Angular Libraries and Microservices

QUICK VIEW

01. Microservices are often considered for use on the backend, but there are applications on the front-end as well.

02. Walk through the implementation of a screen or page that includes data from multiple services using Angular CLI.

03. Review the basics of the client-side UI implementations and the chassis patterns.

BY ANTONIO GONCALVES
SENIOR SOFTWARE ARCHITECT,

We live in a Microservices world, and this world is here to stay. Back-end developers need to dive into Domain Driven Design, write stateless, resilient, highly available services, keep data synchronized through Change Data Capture, handle network failure, deal with authentication, authorization, JWT... and expose a beautiful Hypermedia API so the front-end developers can add a great user interface to it.

Good! But what about the front-end?

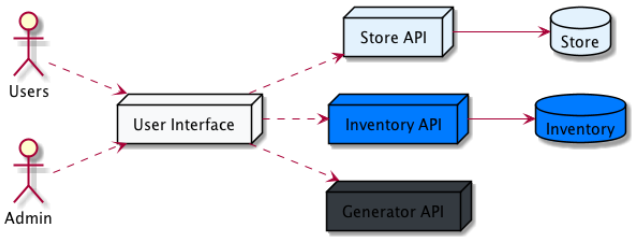
Let's say we have several microservices, several teams, several APIs and, therefore, several user interfaces. How can you create an integrated and consistent user interface so your users don't actually see that there are as many user interfaces as there are Microservices? **In this post, I'm implementing the Client-side UI composition design pattern using Angular Libraries.**

A MICROSERVICES ARCHITECTURE

Let's say we have a CD BookStore application that allows customers to buy CDs and Books online, as well as administrators to check the inventory. We also have some complex ISBN number generator to deal with. If we model this application into Microservices, we might end up splitting it into 3 separate microservices:

- **Store:** Deals with displaying information on CDs and books, allows users to buy them.
- **Inventory:** Allows administrators to check the availability of an item in the warehouse and buy new items if needed.
- **Generator:** Generates ISBN numbers each time a new book is created.

We end up with two roles (user and admin) **interacting with these 3 APIs through a single user interface**:



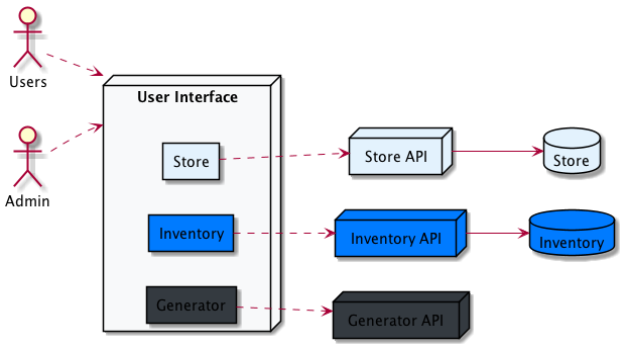
Client-side UI composition

In a perfect world, these three microservices are independent, developed by three different teams and, therefore, have three different user interfaces, each released at its own pace. On one hand we have three different UIs, and on the other, we want our users to navigate into **what they see as a single and integrated application**. There are several ways of doing it, and the one way I'm describing here is called the Client-side UI composition design pattern:

Problem: How to implement a UI screen or page that displays data from multiple services?

Solution: Each team develops a client-side UI component, such an Angular component, that implements the region of the page/screen for their service. A UI team is responsible for implementing the page skeletons that build pages/screens by composing multiple, service-specific UI components.

The idea is to aggregate, on the client side, our three user interfaces into a single one and end up with something that looks like this:



Aggregating several UIs into a single one works better if they use compatible technologies, of course. In this example, I am using Angular and only Angular to create the 3 user-interfaces and the page skeleton. Of course, Web Components would be a perfect fit for this use case, but that's not the purpose of this article.

ANGULAR LIBRARIES

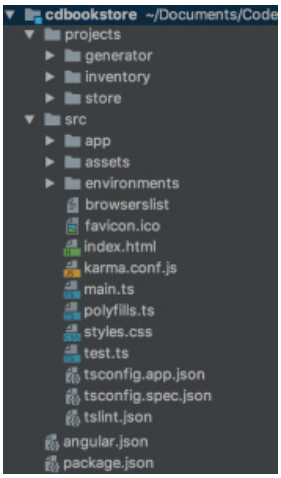
One novelty with Angular CLI 6 is the ability to easily create libraries. Coming back to our architecture, we could say the page skeleton is the Angular application (the CDBookStore application), and then, each microservice user interface is a library (Store, Inventory, Generator).

In terms of Angular CLI commands, this is what we have:

```
# Main app called CDBookStore
$ ng new cdbookstore --directory cdbookstore

/
 -routing
true

# The 3 libraries for our 3 microservices UI
$ ng generate library store --prefix str
$ ng generate library inventory --prefix inv
$ ng generate library generator --prefix gen
```

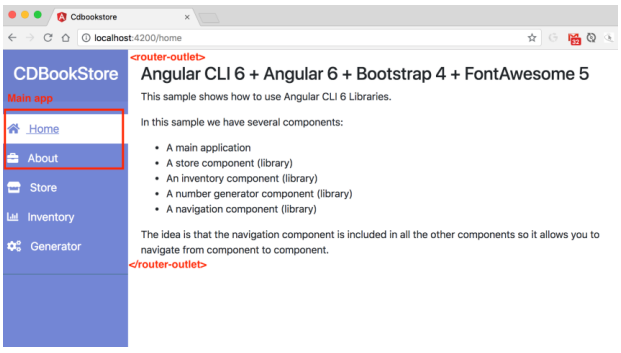


Once you've executed these commands, you will get the following directory structure:

- `src/app/` is the good old Angular structure. In this directory, we will have the page skeleton of the CDBookStore application. As you will see, this skeleton page is only a sidebar to navigate from one Microservice UI to another one.
- `projects/` is the directory where all the libraries end up
- `projects/generator`: the UI for the Generator microservice
- `projects/inventory`: the UI for the Inventory microservice
- `projects/store`: the UI for the Store microservice

THE SKELETON PAGE

The skeleton page is there to aggregate all the other components. Basically, it's only a sidebar menu (allowing us to navigate from one Microservice UI to another one) and a `<router-outlet>`. It is where you could have login/logout and other user preferences. It looks like this:



In terms of code, despite using Bootstrap, there is nothing special. What's interesting is the way the routes are defined. The AppRoutingModule only defines the routes of the main application (here, the Home and the About menus).

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'home', component: HomeComponent},
  {path: 'about', component: AboutComponent},
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

In the HTML side of the side menu bar, we can find the navigation directive and the router:

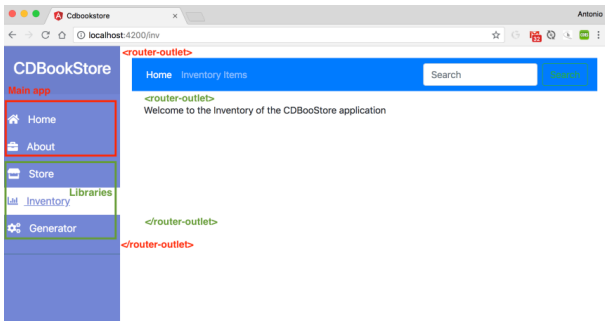
```
<ul class="list-unstyled components">
  <li>
      <a [routerLink]="['/home']">
        <i class="fas fa-home"></i>
        Home
      </a>
      <a [routerLink]="['/str']">
        <i class="fas fa-store"></i>
        Store
      </a>
      <a [routerLink]="['/inv']">
        <i class="fas fa-chart-bar"></i>
        Inventory
      </a>
      <a [routerLink]="['/gen']">
        <i class="fas fa-cogs"></i>
        Generator
      </a>
  </li>
</ul>
<!-- Page Content -->
<div id="content">
  <router-outlet></router-outlet>
</div>
```

As you can see in the code above, the routerLink directive is used to navigate to the main app components as well as library components. The trick here is that `/home` is defined in the routing module, but not `/str`, `/inv` or `/gen`. These routes are defined in the libraries themselves.

Disclaimer: I am a terrible web designer/developer. If someone reading this post knows about jQuery and Angular and want to give me a hand, I would love to have the sidebar to be collapsible. I even created an issue for you!

THE LIBRARIES

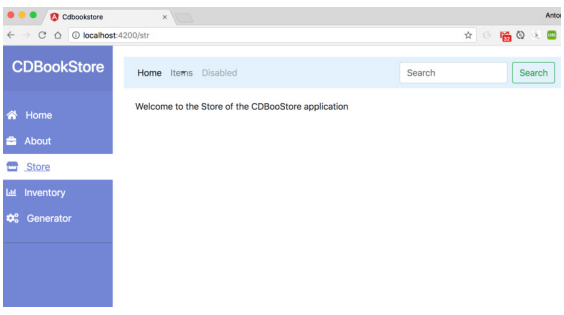
Now when we click on Store, Inventory, or Generator, we want the router to navigate to the library's components:



Notice the parent/child relationship between routers. In the image above, the red router-outlet is the parent and belongs to the main app. The green router-outlet is the child and belongs to the library. Notice that in the store-routing.module.ts we use str has the main path, and all the other paths are children (using the children keyword):

```
const routes: Routes = [
  {
    path: 'str', component: StoreComponent,
children:
[
      {path: '', component: HomeComponent},
      {path: 'home', component: HomeComponent},
      {path: 'book-list', component:
BookListComponent},
      {path: 'book-detail', component:
BookDetailComponent},
    ]
  },
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class StoreRoutingModule {
}
```

The beauty of having a parent/child router relationship is that you can benefit from "feature" navigation. This means that if you navigate to `/home` or `/about` you stay in the main application, but if you navigate to `/str`, `/str/home`, `/str/book-list` or `/str/book-detail`, you navigate to the Store library. You can even do lazy loading on a per feature based (only load the Store library if needed).



The Store library has a navigation bar at the top, therefore, it needs the routerLink directive. Notice that we only need to use the child path `[routerLink]="['book-list']"` without having to specify /str (no need to `[routerLink]="['/str/book-list']"`):

```
<nav>
 <div class="collapse navbar-collapse">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item dropdown">
      <a class="nav-link dropdown-toggle">Items</a>
        <div class="dropdown-menu" aria-labelledby=
"navbarDropdownMenuLink">
         <a class="dropdown-item"
[routerLink]="['book-list']">Books</a>
         <a class="dropdown-item"
[routerLink]="['cd-list']">CDs</a>
         <a class="dropdown-item"
[routerLink]="['dvd-list']">DVDs</a>
        </div>
      </li>
    </ul>
  </div>
</nav>
<!-- Page Content -->
<div id="content">
  <router-outlet></router-outlet>
</div>
```

PROS AND CONS

The Client-side UI composition design pattern has some pros and cons. If you already have existing user-interfaces per microservices, and if each one is using totally different technologies (Angular vs React vs Vue.JS vs ...) or different framework versions (Bootstrap 2 vs Bootstrap 4), then aggregating them might be tricky and you might have to rewrite bits and pieces to have them compatible.

But this can actually be beneficial. If you don't have extended teams with hundreds of developers, you might end up being the one moving from one team to another one and will be happy to find (more or less) the same technologies. This is defined in the Chassis pattern. And for your end users, the application will have the same look and feel (on mobile phones, laptops, desktops, tablets), which gives a sense of integration.

With the way Angular Libraries are structured, having a mono repo is much easier. Of course, each library can have their own life cycle and be independent, but at the end of the day, it makes it easier to have them on the same repo. I don't know if you love MonoRepos or not, but some do.

CONCLUSION

We've been architecting and developing Microservices for several decades now (yes, it used to be called distributed systems, or distributed services), so we roughly know how to make them work on the back-end. Now, the challenge is to be able to have a user interface which communicates with several Microservices, developed by different teams, and at the same time feels consistent and integrated for the end-user. You might not always have this need (see how Amazon has totally different UIs for its services), but if you do, then the Client-side UI composition design pattern is a good alternative.

If you want to give it a try, download the code, install and run it (there is only the front-end, no back-end APIs). And don't forget to give me some feedback. I would be interested to know what you do to aggregate several user-interfaces into "a single one."

REFERENCES

- Building and publishing Angular libraries using Angular CLI
- How to build a library for Angular apps?
- Building an Angular Library with the Angular CLI (version 6)
- Building an Angular Library?
- Angular Routing
- Client-side UI composition pattern
- NG CLI
- Bootstrap
- Bootstrap sidebar
- Why Google Stores Billions of Lines of Code in a Single Repository
- You too can love the MonoRepo

ANTONIO GONCALVES is a senior software architect living in Paris. Initially focused on Java development since the late 1990s, his career has taken him to different countries and companies where he works now as a Java EE consultant in software architecture. He is particularly fond of open source and is a member of the OOSGTP (Open Source Get Together Paris). He is also the co-creator of the Paris Java User Group and talks on Les Cast Codeurs podcast. He has also written numerous technical papers and articles for IT Web sites (DevX, JaxEnter, DZone) and IT magazines (Programmez, Linux Magazine).

CLOUD FOUNDRY

TECHNOLOGIES

THE FREEDOM OF OPEN SOURCE

FLEXIBILITY 

INTEROPERABILITY 

PRODUCTIVITY 

SCALABILITY 

SEE WHY ORGANIZATIONS CHOOSE CLOUD FOUNDRY

http://www.cloudfoundry.org/why-cloud-foundry

Cloud Foundry - Making Microservices Easy

Microservices add a layer of complexity to application development — we can agree on that. But they also enable choice, flexibility and independence far more than developing within a monolithic architecture. Microservices enable small teams to work nimbly on a particular function apart from the larger team. You can iterate, release and manage that function independently, giving you concentrated control.

If you use microservices to help your teams move faster and release code more frequently, the path from writing code to production should be as quick as possible. Platforms play a big part in this, coupled with a Continuous Integration tool, which takes responsibility for testing the code, approving it for release, compiling and publishing it.

Having a platform like Cloud Foundry when you go the micro-services route helps offset operational complexity. Both the Cloud Foundry Application Runtime and the Cloud Foundry Container Runtime manage the health of microservices. What does this mean? Your platform will automatically scale up your app when there's demand, watch for unhealthy instances, describe services talking to each other, accept code being offered by Continuous Integration pipelines, and more.

You don't need to use a microservices architecture to get value out of Cloud Foundry — you can certainly run monolithic applications inside it too, and see the same benefits. But if you choose a microservices architecture, a platform like Cloud Foundry removes the operational complexity and manages your application lifecycle to allow you and your teams to focus on your business — and ship great software.

WRITTEN BY CHIP CHILDERS
CTO, CLOUD FOUNDRY FOUNDATION

PARTNER SPOTLIGHT

Cloud Foundry Technologies

CLOUD FOUNDRY

Cloud Foundry makes it faster and easier to build, test, deploy and scale applications on any cloud, developer framework, and application service.

PRODUCT
Cloud Application Platform

OPEN SOURCE?
Yes

RELEASE SCHEDULE
Continuous

CASE STUDY

Zipcar isn't just in the business of car-sharing. The brand is a full-fledged software company that depends on technology to manage reservations, memberships, logistics, vehicle tracking and maintenance. With one million customers across 10 countries, Zipcar designed a bespoke solution to meet its specific needs, moving away from legacy architecture to a continuous delivery model built on cloud native technologies.

The team credits Cloud Foundry Application Runtime, Cloud Foundry BOSH, Concourse, Diego, Garden, and a customized cloud controller with having helped them standardize across its data centers.

Zipcar is currently running:

- 6,000 containers
- 12 BOSH directors
- 80+ services
- 25+ production environments
- Multiple clouds

STRENGTH

- Polyglot
- Multi-Cloud
- Runs apps at scale
- Interoperability
- Simplifies the developer lifecycle

NOTABLE CUSTOMERS

- American Airlines
- The Home Depot
- United States Air Force
- Verizon Wireless

WEBSITE cloudfoundry.org | **TWITTER** @cloudfoundry | **BLOG** cloudfoundry.org/blog

Executive Insights on Microservices

BY TOM SMITH
RESEARCH ANALYST AT **DEVADA**

QUICK VIEW

01. The most important elements of microservices are business processes, scalability, agility, APIs, decoupling, and management/culture.

02. Microservices have made application development faster, it has given developers and engineers more autonomy, and it has increased the agility and scalability of applications.

03. There is rapid to moderate adoption of microservices driven by the adoption of mesh network management (ISTIO).

To understand the current and future state of microservices, we spoke to 25 IT executives from 21 organizations. Consistent with the early adoption of the technology, the answers to our questions were diverse.

Here's who we spoke to:

- Heikki Nousiainen, CTO, Aiven.io
- Chase Aucoin, Technical Evangelist, AppDynamics
- Assaf Mizrachi, Head of Software, Augury
- Amit Ziv-Kenet, Backend Developer, Augury
- Bernd Ruecker, Co-founder and Developer Advocate, Camunda
- Jaime Ryan, Senior Director, Product Management and Strategy, CA Technologies
- Brian Dawson, DevOps Evangelist, CloudBees
- Viktor Farcic, Senior Consultant, CloudBees
- Chip Childers, CTO, Cloud Foundry
- Gwen Shapira, Chief Data Architect, Confluent
- Matthew Baier, COO, Contentstack
- Anders Wallgren, CTO, Electric Cloud
- Priyanka Sharma, Director of Alliances, GitLab
- Andrew Newdigate, Infrastructure Architect, GitLab
- Ben Sigelman, CEO, LightStep
- Jim Scott, Director, Enterprise Strategy & Architecture, MapR
- Ariff Kassam, Vice President, Products, NuoDB
- Jim Walker, VP of Product Marketing, OverOps
- Bich Le, Chief Architect, Platform9
- Mike LaFleur, Global Head of Solution Architecture, Provenir
- Christian Posta, Chief Architect, Cloud Application Development, Red Hat
- Setu Kulkarni, V.P. Strategy and Business Development, Whitehat Security
- Asanka Abeysinghe, V.P. of Architecture – CTO Office, WSO2
- Karun Indrasiri, Director, Integration architecture, WSO2
- Roman Shaposhnik, Co-founder, Zededa

1. You can see how early it is in the adoption of microservices as the "most important elements" are all over the place with business processes, scalability, agility, APIs, decoupling, and management/culture all mentioned a couple of times.

Think about breaking down a business plan or process. Understand and define different business domains in an isolated manner. Understand the original monolithic process and what problems you're trying to solve by decomposing the monolithic processes. It doesn't matter that a single microservice is elegantly architected If the big-picture business processes aren't executed successfully, since these are what make the customers happy and make the company money.

Solve the real problem around engineering and scaling. Enforce an architecture that will help the organization scale

quickly while being agile. Microservices allow you to pull off the inverse of Conway's law. They should be independently developed, deployed, and scaled, allowing for faster delivery of functionality with little impact to other systems. Improve the ability to make changes and get them to the customer quickly. Think through the responsibility for each microservice and how they will interact with one another for greater agility and scale.

Isolation and API guarantees are key elements. Isolation enables developers to iterate quickly and independently. API compatibility is key so other microservices are not impacted or dependencies are created that slow down iterations. You need well-defined APIs that can be easily discovered, understood, and executed by service clients.

Ensure microservices are adequately decoupled but avoid premature decomposition. Start small and evolve. Design applications to small teams can have full control of a service or an app that's fully independent of everything else. If you are breaking down a monolithic application, take an iterative approach, decompose incrementally. Break out, run, and operate with a line of decomposition before you move on. Ensure autonomy for a service. Deploy independently at the finest level of granularity.

Microservices are a management issue moving from hundreds of developers working on an application to no more than 10 to 12 people per team. The culture of the organization is critical as this is where we see the manifestation of communication problems. Without meeting the communication challenge, organizations don't get the outcomes they want. Monitoring is a huge component of microservices with heavily decoupled distribution, service visibility becomes an issue quickly.

2. Microservices have made application development faster, it has given developers and engineers more autonomy, and it has increased the agility and scalability of applications.

Microservices enable everything to move faster – development, onboarding, feature release, adoption, uptake, feedback, and improvement. Things happen faster once you've broken things down. You get speed with quality because there's a smaller surface to test.

It changes how organizations work and meet business requirements. Application developers get a lot of freedom. They get an API and can look at end user, use, and release in a rapid way without dealing with backend complexity.

A team bigger than ten people is not efficient and hinders autonomy. It has given architects more independence and choice in selecting the technologies used by a specific microservice. They can select technologies that are the "best fit" for their specific microservice.

Microservices are beneficial to the agility and vitality of an organization. There is greater agility and alignment along a common, product-focused vision. Microservices force you to adopt an Agile methodology to be flexible and responsive because there are so many moving pieces. Testing is easier. Teams can be moved around, the learning curve is smaller, onboarding is quicker, small teams communicate more efficiently, and smaller code bases are easier to maintain.

3. Respondents are seeing rapid to moderate adoption of microservices and it seems to be driven by the adoption of mesh network management tools, such as Istio. Tools, systems, and platforms are making it easier, reducing the downside, and making applications more consumable. The transition to microservices-oriented networking has shifted thinking about sockets to microservices. The service mesh takes out the shared responsibility by putting everything in a container enabling everyone to use the same sidecar. Serverless computing also allows applications to be even smaller. The concept of a service mesh sprang from nothing last year and is continuing to generate a lot of interest and some adoption. However, these technologies are still in their infancy and adopters are discovering the hidden pitfalls. Patience is required as the industry sorts itself out.

4. The majority of real world use cases provided by our respondents were in financial services though several people pointed out the benefits of microservices are irrespective of domain. Everything being done in financial services can be used by a number of different verticals. The biggest and most relevant use case is fraud detection built out machine learning models and putting in line with microservices. A real challenge is agility around making an easy change to a process – like the credit worthiness of a customer. Data scientists are able to make a change to a model quickly versus changing the entire software development lifecycle. Large financial institutions are able to deliver new functionality using microservices while maintaining and continuing to leverage legacy applications through API-based integrations. They're able to provide new, all-digital products and services because they are better able to respond to customer needs without needs to rip and replace.

5. The most common issues affecting the creation and management of microservices are data, complexity, monitoring, skills, and security. A key challenge is managing data and data sets with stateful and stateless data. You need to understand how to create data compositions. You need to think about monitoring and observability with autonomous services disbursed across an entire network.

Most traditional organizations have to move legacy apps. That's a multi-year journey. You cannot stop supporting the monolith. Simplicity is the most fundamental thing to keep in mind. Have clarity of the domain problem you are trying to solve. Create clean and simple APIs and logic that's easy to wrap your head around to create more complex apps and behaviors. Microservices bring a lot of overhead. Microservices are more difficult to debug and troubleshoot.

There's a pretty big skills gap when it comes to building, delivering and managing microservices. Few developers have the skills necessary to understand the domain, code the services from data to API, understand the overall architecture, integrate with a CI/CD pipeline, and deliver a running service. Most do not understand the gravity of building services as fine grain and collectively using a single application.

6. The biggest concerns with the current state of microservices are observability and maturity. It's really important to have visibility into business processes. How to know something is going wrong when it's going wrong is key. The ability to do distributed training through the whole system is necessary. Without monitoring, you're not able to see where the problem is. Istio helps with communication in a highly distributed network. Enhanced automation and operational visibility help shine light onto the increased complexity.

Microservices in production is more complex than we initially thought. We are confronting a lack of maturity in tools to monitor and troubleshoot. This delays the ability to move to production. Organizations adopting microservices need to know what they are trying to accomplish rather than a desire to be on the cutting edge. We need independent business logic but not technology choices. A lack of standards is resulting in confusion but as more organizations adopt, best practices and standards will fall into place.

7. The future of microservices is FaaS and serverless with improved tooling. Serverless is gaining traction too, and microservices are an important step on the learning journey that will lead us closer to serverless. The promise for the software developer of getting rid of operational detail is great, while the promise of huge cost savings by only paying for the exact amount of compute power used is even more appealing. FaaS is a few years behind microservices. Infrastructure will be less of barrier to entry. Serverless and FaaS will enable developers to go smaller. Microservices are going to become even more micro with serverless. Serverless and FaaS are different but are the next generation of application development. There's a lot of value in going to a serverless model.

There will be improvements in the development of Kubernetes and tools provided by AWS and GCS to help go from development to deployment and production more quickly and easily. Simplifying as much as possible so developers, operations, and DevOps can run the applications they want in a way that scales automatically, and everything is at their fingertips to troubleshoot any problems without worrying about the underlying infrastructure and orchestration.

8. When working on microservices, developers need to keep a lot in mind. Those items mentioned more than once are APIs, security, reducing complexity, and remembering SDLC principles. Every developer needs to have API development and management skills in their toolbox. Make service calls as autonomous as possible. Use APIs to minimize duplication of functionality across the organization. Concentrate on clean interfaces between modules.

Focus on secure design patterns. End point security is crucial. Remove as much operational and design complexity as possible. Look for trends that hide complexity from developers and make them as productive as they can be without worrying about infrastructure. Dive into core SDLC principles. Understand the end-to-end lifecycle of a microservice to make yourself more valuable to employers and more marketable in the industry.

TOM SMITH is a Research Analyst at DZone who excels at gathering insights from analytics—both quantitative and qualitative—to drive business results. His passion is sharing information of value to help people succeed. In his spare time, you can find him either eating at Chipotle or working out at the gym.

diving deeper

INTO **MICROSERVICES**

twitter

 **@jessfraz**

 **@rhein_wein**

 **@b0rk**

 **@crichardson**

 **@simona_cotin**

 **@jldeen**

 **@christianposta**

 **@michelebusta**

 **@martinfowler**

 **@ThoHeller**

videos

What Are Microservices?

Learn what microservices are, why people are moving to microservices, and how you can create effective microservices.

Mastering Chaos – A Netflix Guide to Microservices

Get insight into how Netflix does microservices, what the anatomy of a microservice is, what cultural methods can help achieve microservice mastery, and more.

Introduction to Microservices, Docker, and Kubernetes

Learn about using microservices with two of the most popular container platforms out there.

zones

Microservices dzone.com/microservices

The Microservices Zone will take you through breaking down the monolith step-by-step and designing microservices architecture from scratch. It covers everything from scalability to patterns and anti-patterns. It digs deeper than just containers to give you practical applications and business use cases.

Integration dzone.com/integration

The Integration Zone focuses on communication architectures, message brokers, enterprise applications, ESBs, integration protocols, web services, service-oriented architecture (SOA), message-oriented middleware (MOM), and API management.

Cloud dzone.com/cloud

The Cloud Zone covers the host of providers and utilities that make cloud computing possible and push the limits (and savings) with which we can deploy, store, and host applications in a flexible, elastic manner. The Cloud Zone focuses on PaaS, infrastructures, security, scalability, and hosting servers.

refcardz

Getting Started With Spring Boot and Microservices

This Refcard will show you how to incorporate Spring Boot and Hazelcast IMDG into a microservices platform, how to enhance the benefits of the microservices landscape, and how to alleviate the drawbacks of utilizing this method.

Patterns of Modular Architecture

Covers 18 modularity patterns to help developers incorporate modular design thinking into development initiatives.

Getting Started With Microservices

In this updated Refcard, you will learn why microservices are becoming the cornerstone of modern architecture, how to begin refactoring your monolithic application, and common patterns to help you get started.

books

Building Microservices: Designing Fine-Grained Systems

Get examples and practical advice to help you build, manage, and evolve your microservice architectures.

Spring Microservices in Action

Learn how to build microservice-based applications using Java and the Spring platform.

The Tao of Microservices

Get yourself on the path to microservices with a conceptual view of microservice design, core concepts, and microservices applications.

LIGHTSTEP

Solving the Microservices Murder Mystery

When you run microservices, every outage can feel like you're solving a murder mystery. It's hard to sift through performance data and find the true culprit.

LightStep [x]PM helps you pinpoint the root cause and gain control over microservices complexity. It has tracing that scales and is a unique approach to application performance management.

[x]PM analyzes 100% of the performance data, 100% of the time, with no upfront sampling, so you never miss an outlier.

IN PRODUCTION AT

twilio · DigitalOcean · lyft · GitHub · Medium · BIGCOMMERCE

UNDER ARMOUR · indeed · Airtable · skyscanner · Segment · invision

Try the [x]PM guided demo

https://go.lightstep.com/Try-xPM.html

lightstep.com · LightStepHQ · LightStep · LightStepHQ

LightStep [x]PM: APM for Microservices and Serverless

LightStep® [x]PM™ provides APM for today's complex systems including microservices and serverless functions. [x]PM helps organizations running microservice architectures maintain control over their systems, so they can reduce MTTR during firefighting situations and make proactive application performance improvements.

[x]PM's unique design allows it to analyze 100% of unsampled transaction data from highly-distributed, large-scale production software to produce meaningful distributed traces and metrics that explain performance behaviors and accelerate root cause analysis. In addition, [x]PM's intuitive user experience makes it easy to discover, categorize, and explain distinct latency behaviors occurring throughout an application. Users can even visually compare current performance against past performance along any application dimensions, so it's immediately clear whether the behavior being observed is normal or not. Finally, [x]PM follows entire transactions from clients making requests down to low-level services and back, revealing how every service interacted and responded. [x]PM automatically computes a transaction's critical path, so users can quickly navigate to bottlenecks in the system.

[x]PM was built for the enterprise:

- Integrations with CNCF technologies including OpenTracing, Envoy, gRPC, Istio, Linkerd, and Kubernetes
- Workflow integrations with Grafana, Slack, PagerDuty, and Sumo Logic
- RBAC, SSO with SAML, and OAuth support
- Secure TLS communication and robust APIs

[x]PM is used in production by engineering leaders at Twilio, Lyft, Medium, DigitalOcean, GitHub and many others to help them gain control over their complex systems.

WRITTEN BY DENNIS CHU
DIRECTOR OF PRODUCT MARKETING, LIGHTSTEP

PARTNER SPOTLIGHT

LightStep [x]PM

LightStep [x]PM: Taming Complexity with Distributed Tracing that Scales

PRODUCT

Application Performance Management for companies adopting microservices and serverless

OPEN SOURCE?

No

NEWEST RELEASE

Monthly

PRODUCT OVERVIEW

Twilio Improves Mean Time to Resolution (MTTR) by 92% with LightStep [x]PM. Twilio wanted to go beyond traditional monitoring approaches and deliver a flawless experience to their 1.6M developers registered on the Twilio platform, as well as provide additional service and insights for premium clients. Twilio uses [x]PM to help them solve production-level issues, meet their operational excellence goals, and deliver a higher level of operational maturity to their customers. Results include:

- Improved MTTR by 92% for production issues
- Reduced latency by 70%
- Saved Insight Engineering team 20 hours per week
- Launched segmented and detailed performance monitoring for each top Twilio customer, including customer-specific root cause analysis

STRENGTHS

- Provides end-to-end distributed tracing that scales
- No upfront sampling so teams never miss a transaction that matters
- Segments with unlimited cardinality
- Streamlines root cause analysis by revealing service dependencies
- Visualizes system performance and provides insight into what's normal (and not)

NOTABLE CUSTOMERS

- Lyft
- Twilio
- GitHub
- DigitalOcean
- Segment

WEBSITE lightstep.com **TWITTER** @LightStepHQ **BLOG** lightstep.com/blog

Solutions Directory

This directory contains platforms, middleware, service meshes, service discovery, and distributed tracing tools to build and manage applications built with microservices. It provides free trial data and product category information gathered from vendor websites and project pages. Solutions are selected for inclusion based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability.

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Aiven	Aiven	Cloud database & management services	Available by request	aiven.io
Amazon Web Services	Amazon EC2	IaaS	Free tier available	aws.amazon.com/ec2
Amazon Web Services	Amazon API Gateway	API gateway	Free tier available	aws.amazon.com/api-gateway
Amazon Web Services	Simple Query Service (SQS)	ESB	Free tier available	aws.amazon.com/sqs
Amazon Web Services	AWS Application Discovery Service	Service discovery	Free tier available	aws.amazon.com/application-discovery
Amazon Web Services	Amazon ECS	Container orchestration	Free tier available	aws.amazon.com/ecs
Apache Foundation	Kafka	Distributed streaming platform	Open source	kafka.apache.org
Apache Foundation	Zookeeper	Service discovery	Open source	zookeeper.apache.org
Apache Foundation	HTrace	Distributed tracing	Open source	htrace.incubator.apache.org
Apache Foundation	ActiveMQ	Message queue	Open source	activemq.apache.org
Apcera	NATS	Message-oriented middleware	Open source	nats.io
Apigee	Apigee	API gateway, API management	Free tier available	apigee.com/api-management/#/products

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Aspen Mesh	Aspen Mesh	Containerized apps management system	Available by request	aspenmesh.io
Axway	Axway AMPLIFY	API management, API gateway, API builder	Available by request	axway.com/en/products/amplify
Buoyant	Linkerd	Service mesh	Open source	linkerd.io
CA	CA API Management	API management, API gateway	Available by request	ca.com/us/products/api-management.html
Canonical	LXD	Container management	Open source	linuxcontainers.org/lxd
Cisco	AppDynamics	Performance & monitoring tool	15 days	appdynamics.com
Cloudentity	Cloudentity	Identity & API security	Available by request	cloudentity.com
Cloud Foundry	Diego	Container runtime system	Open source	github.com/cloudfoundry/diego-release
Cloud Foundry	CF Container Runtime	Container deployment & management	Open source	cloudfoundry.org/container-runtime
Cloud Native Computing Foundation	Envoy	Edge & service proxy	Open source	envoyproxy.io
Cloud Native Computing Foundation	OpenTracing	Distributed tracing APIs	Open source	opentracing.io
Cloud Native Computing Foundation	containerd	Container runtime system	Open source	containerd.io
Docker	Docker	Container platform	Free tier available	docker.com/get-docker
Docker	Docker Swarm	Container orchestration & clustering	Open source	github.com/docker/swarm
Dropwizard	Dropwizard	Web services development framework	Open source	dropwizard.io
Dynatrace	Dynatrace	Performance & monitoring tool	15 days	dynatrace.com
Eclipse Foundation	Vert.x	Reactive application development platform	Open source	vertx.io
Elastic	Elasticsearch	Search & analytics	Open source	elastic.co/products/elasticsearch

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Elastic	Kibana	Elastic stack configuration & management	Open source	elastic.co/products/kibana
Fluentd	Fluentd	Unified logging layer	Open source	fluentd.org
Google	Kubernetes Engine	Container orchestration	$300 credit	cloud.google.com/kubernetes-engine
gRPC	gRPC	Protocol buffers, RPC system	Open source	grpc.io
HashiCorp	Consul	Service discovery, configuration, & monitoring	Open source	consul.io
IBM	IBM API Connect	API management	Free tier available	ibm.com/cloud/api-connect
Instana	Instana Infrastructure Quality Management	Infrastructure monitoring	14 days	instana.com/infrastructure-management
Istio	Istio	Service mesh	Open source	istio.io
Jaeger	Jaeger	Distributed tracing	Open source	github.com/jaegertracing/jaeger
JHipster	Jhipster	Spring Boot & Angular application + microservices development platform	Open source	jhipster.tech
Kong	Kong	API gateway & microservices management	Demo available by request	konghq.com
Kubernetes	Kubernetes	Container orchestration	Open source	kubernetes.io
Lightbend	Akka	Services communication	Open source	lightbend.com/akka
Lightbend	Lagom	Microservices development platform	Open source	lightbend.com/lagom-framework
Lightbend	OpsClarity	Reactive systems monitoring	N/A	opsclarity.com
LightStep	LightStep	Microservices management	Demo available by request	lightstep.com
Linux Foundation	The Linux Foundation	Open-source project hosting	Open source	linuxfoundation.org
Macaw Software	Macaw	Microservices development platform	Free tier available	macaw.io

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Mesosphere	Marathon	Container orchestration	Open source	mesosphere.github.io/marathon
Micro Focus	Artix	ESB	30 days	microfocus.com/products/corba/artix#
Micrometer	Micrometer	JVM application monitoring	Open source	micrometer.io
MicroProfile	MicroProfile	Java optimization project for microservices development	Open source	microprofile.io
Microsoft	Azure API Management	API gateway, API management	Free tier available	azure.microsoft.com/en-us/services/api-management
Microsoft	Azure Container Service	Container orchestration	Free tier available	azure.microsoft.com/en-us/services/container-service
Microsoft	Azure Service Bus	ESB	Free tier available	azure.microsoft.com/en-us/services/service-bus
Microsoft	Azure Service Fabric	Microservices development platform	Free tier available	azure.microsoft.com/en-us/services/service-fabric
Mulesoft	Anypoint Platform	Integration platform	Available by request	mulesoft.com
Netflix	Archaius	Configuration managmeent	Open source	github.com/Netflix/archaius
Netflix	Eureka	Service discovery	Open source	github.com/Netflix/eureka
Netflix	Hystrix	Latency & fault tolerance library	Open source	github.com/Netflix/Hystrix
Netflix	Ribbon	Load balancing library	Open source	github.com/Netflix/ribbon
Netflix	Zuul	Dynamic routing & service monitoring	Open source	github.com/Netflix/zuul
Neuron ESB	Neuron ESB	ESB	30 days	neuronesb.com/
NGINX	NGINX Application Platform	Microservices development & management	30 days	nginx.com/products/
NGINX	nginmesh	Service mesh	Open source	github.com/nginmesh/nginmesh

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Nutanix	Xi Epoch	Monitoring multi-cloud apps	Available by request	nutanix.com/products/epoch
OCI	Grails	Web application framework	Open source	grails.org
OpenESB	OpenESB	ESB	Open source	open-esb.net
OpenLegacy	OpenLegacy	API management	N/A	openlegacy.com
Oracle	Java EE	Java specifications	Free solution	oracle.com/technetwork/java/javaee/overview/index.html
Oracle	Oracle Service Bus	ESB	Free solution	oracle.com/technetwork/middleware/service-bus/overview
Oracle	Oracle SOA Suite	SOA governance, integration PaaS	30 days	oracle.com/middleware/application-integration/products/soa-suite.html
OW2 Middleware Consortium	Petals ESB	ESB	Open source	petals.linagora.com
Particular Software	NServiceBus	ESB	Free tier available	particular.net/nservicebus
Pivotal Software, Inc.	RabbitMQ	Message queue	Open source	network.pivotal.io/products/pivotal-rabbitmq
Pivotal Software, Inc.	Spring Cloud Sleuth	Distributed tracing	Open source	spring.io/projects/spring-cloud-sleuth
Pivotal Software, Inc.	Spring Boot	Spring application development platform	Open source	projects.spring.io/spring-boot
Prometheus	Prometheus	Systems monitoring & alerting	Open source	prometheus.io
Red Hat	CoreOS Fleet	Container orchestration	Open source	github.com/coreos/fleet
Red Hat	CoreOS Flannel	Container-defined networking	Open source	github.com/coreos/flannel
Red Hat	CoreOS etcd	Service discovery	Open source	github.com/etcd-io/etcd
Red Hat	JBoss Fuse	ESB	Open source	redhat.com/en/technologies/jboss-middleware/fuse

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Red Hat	Red Hat Enterprise Linux Atomic Host	Container management	Open source	redhat.com/en/resources/enterprise-linux-atomic-host-datasheet
Red Hat	Thorntail	Java EE services development	Open source	thorntail.io
Rogue Wave Software	Akana API Management	API management	Demo available by request	roguewave.com/products/akana/solutions/api-management
SignalFX	SignalFX	Monitoring, alerts, & analytics	14 days	signalfx.com/products
SimianViz	SimianViz	Microservices simulation	Open source	github.com/adrianco/spigo
Sysdig	Sysdig Falco	Behavioral activity monitor w/ container support	Open source	sysdig.com/falco
TIBCO Software Inc.	Mashery	API management	30 days	mashery.com/api-management/saas
Twistlock	Twistlock	Container security	30 days	twistlock.com
Twitter	Finagle	RPC system	Open source	twitter.github.io/finagle
Tyk.io	Tyk	API management	Open source	github.com/TykTechnologies/tyk
VMWare	Photon	Container-optimized operating system	Open source	vmware.github.io/photon
WSO2	WSO2	API management	Open source	wso2.com/api-management
X-Trace	X-Trace	Distributed tracing	Open source	github.com/rfonseca/X-Trace
Zapier	Zapier	API management	Free tier available	zapier.com
Zipkin	Zipkin	Distributed tracing	Open source	zipkin.io

GLOSSARY

ANGULAR: A platform for building HTML & JavaScript web applications, led by Google & based on TypeScript; a rewrite of the AngularJS JavaScript framework.

APACHE KAFKA: An open-source distributed stream processing platform based on an abstraction of a distribtued commit log.

APPLICATION PROGRAMMING INTERFACE (API): A software interface that allows users to configure &interact w/other programs, usually by calling from a list of functions.

CONTAINER: Resource isolation at the OS (rather than machine) level, usually (in UNIX-based systems) in user space. Isolated elements vary by containerization strategy &often include file system, disk quota, CPU &memory, I/O rate, root privileges, &network access. Much lighter-weight than machine-level virtualization & sufficient for many isolation requirement sets.

CONTINUOUS DELIVERY: A software engineering approach in which continuous integration, automated testing, & automated deployment capabilities allow software to be developed & deployed rapidly, reliably, & repeatedly w/minimal human intervention.

DEPENDENCY INJECTION: This can position a class to be independent of its dependencies, as one object supplies the dependencies of another object.

DISTRIBUTED SYSTEM: Any system or application that operates across a wide network of services or nodes.

DISTRIBUTED TRACING: A category of tools & practices that allow developers to analyze the behavior of a service & troubleshoot problems by creating services that record information about requests & operations that are performed.

DOCKER: A computer program that performs operating-system-level virtualization & specializes in running software packages within standalone containers.

DOMAIN-DRIVEN DESIGN: A philosophy for developing software in which development is focused primarily on the business logic (the activities & issues that an application is supposed to perform or solve).

ENTERPRISE SERVICE BUS (ESB): A utility that combines a messaging system w/middleware to provide comprehensive communication services for software applications.

EVENTUAL CONSISTENCY: A data consistency model used to make distributed applications highly available by keeping data in sync & up-to-date across all services or nodes.

FUNCTION-AS-A-SERVICE: A relatively new concept to describe a category of cloud computing services that deliver a platform allowing users to develop, run, & manage application functionalities.

HOLACRACY: A management practice for organizations that are separated into autonomous & independent departments based on roles, which can organize themselves &make decisions based on their duties. Holacracies are focused on rapidly iterating.

JAVA VIRTUAL MACHINE (JVM): Abstracted software that allows a computer to run a Java program.

MESSAGE BROKER: Middleware that translates a message sent by one piece of software to be read by another piece of software.

MICROPROCESSOR: An integrated circuit that contains all the functions of a central processing unit of a computer.

MICROSERVICES ARCHITECTURE: A development method of designing your applications as modular services that seamlessly adapt to a highly scalable & dynamic environment.

MONOLITHIC: An architecture that is reliant on a large, single code base that lends to a unified application.

ORCHESTRATION: The method to automate the management & deployment of your applications & containers.

SERVERLESS: A platform providing computing, networking, & storage without the need of managing (virtual) machines.

SERVICE DISCOVERY: The act of finding the network location of a service instance for further use.

SERVICE MESH: An infrastructure layer focused on service-to-service communication, primarily used for distributed systems & cloud-native applications.

SOCIOCRACY: A mode of governance without a centralized power structure, aiming for less independence between teams to focus on organization-wide strategy.

WEB SERVICE: A function that can be accessed over the web in a standardized way using APIs that are accessed via HTTP & executed on a remote system.

INTRODUCING THE

Open Source Zone

Start Contributing to OSS Communities and Discover Practical Use Cases for Open-Source Software

Whether you are transitioning from a closed to an open community or building an OSS project from the ground up, this Zone will help you solve real-world problems with open-source software.

Learn how to make your first OSS contribution, discover best practices for maintaining and securing your community, and explore the nitty-gritty licensing and legal aspects of OSS projects.

COMMITTERS & MAINTAINERS

COMMUNITY GUIDELINES

LICENSES & GOVERNANCE

TECHNICAL DEBT

Visit the Zone

BROUGHT TO YOU IN PARTNERSHIP WITH Flexera