

APPLICATIONS & MICROSERVICES WITH DOCKER & CONTAINERS

EDITED & CURATED BY ALEX WILLIAMS

The New Stack:

The Docker and Container Ecosystem eBook Series

Alex Williams, Founder & Editor-in-Chief

Benjamin Ball, Technical Editor & Producer

Hoang Dinh, Creative Director

Sam Charrington, Editor, Founder and Principal Analyst of CloudPulse Strategies

Contributors:

Joab Jackson, Managing Editor

Judy Williams, Copy Editor

Lawrence Hecht, Data Research Director

Luke Lefler, Audio Engineer

Michelle Maher, Copy Editor

Patricia Dugan, Director of Community Marketing & Development

TABLE OF CONTENTS

Introduction	4
Sponsors	6
APPLICATIONS & MICROSERVICES WITH DOCKER & CONTAINERS	
From Monolith to Microservices	7
The Ten Commandments of Microservices	20
How Microservices Have Changed and Why They Matter	30
Apcera: Creating Standards for the Container Ecosystem	40
Containers in Production, Part I: Case Studies	41
Cisco: Microservices Frameworks for Handling Complexity at Scale	51
The Developers and Companies Shaping the Open Source Container Ecosystem	52
Docker: Rethinking the Development and Delivery Environments	72
Containers in Production, Part II: Workflows	73
IBM: The Evolution of Architectures at Container-Scale	81
Leveraging Containers to Provide Feedback Loops for Developers	82
Joyent: A Historical Perspective and the Future of Containers	90
How Containers and Microservices Work Together to Enable Agility	91
Pivotal: What Does It Mean to Be Cloud Native?	99
The Role of Platform-as-a-Service in the Container Era	100
VMware: Integrating Containers Allows Businesses to Move Faster	105
Achieving Innovation and Agility With Cloud-Native Application Architectures	106
APPLICATIONS & MICROSERVICES DIRECTORY	
Microservices Frameworks	123
Provision of Capabilities for Microservices	129
Deployment and Continuous Integration	139
Disclosures	147

INTRODUCTION

As an architectural pattern, microservices have been getting a lot of attention in the last several years, reaching a level of adoption and evangelism that would be hard for most practitioners to ignore. But while microservices architectures have received wide praise as changing the application development and production process on the whole, there are many that dismiss microservices as nothing new, or to some, a different name for service-oriented architecture (SOA). It's important to explore both the drivers that shaped how microservices evolved into the pattern we know today, and also to understand what tools and services have made its current popularity so widely accessible.

In this ebook, *The New Stack* explores the ways that container-based microservices have impacted application development and delivery. We learn about what containers mean to the process of creating and utilizing microservices. Through that lens, we look at how container technology has become so important in implementing the pattern of microservices.

Within this ebook, we explore what makes up a microservices architecture, and offer some best practices for creating and managing microservices — as well as some processes around migrating monolithic applications to microservices. We also provide case studies for containers in production.

There is also a great deal of original research we've performed about the open source container ecosystem that we are eager to share with our broader community; this series' focus on open source communities is a continuing topic that we are looking to expand upon, and we welcome feedback on what we've done so far.

We see this second ebook as an important step in building off our first ebook, “The Docker & Container Ecosystem.” Understanding the role of containers and microservices in application development and architecture will allow us to later address the complex topics ahead in our next ebook, “Automation & Orchestration with Docker & Containers,” where we’ll cover orchestration, service discovery, schedulers, cluster management, configuration management, and automation across distributed systems.

This ebook, as well as the three to come in this series, breaks down our analysis of the ecosystem into areas of focus that need the added depth of an entire book. We discovered early on that there’s much more to say about the container ecosystem, and this series is likely just the start for us. We’re constantly looking for new topics in need of greater focus and education, and we welcome any feedback on what areas we should tackle next.

Thanks so much for your interest in our ebook series. Please reach out to our team any time with feedback, thoughts, and ideas for the future.

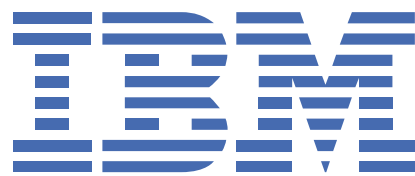
Thanks,
Ben

Benjamin Ball

Technical Editor and Producer
The New Stack

SPONSORS

We are grateful for the support of the following ebook series sponsors:

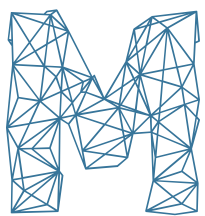


And the following sponsors for this ebook:



FROM MONOLITH TO MICROSERVICES

by **VIVEK JUNEJA**



Most people outside of IT just don't get how difficult it is to manage complex enterprise systems. It is a delicate balancing act, one that rests on understanding how any one change will affect the entire system. New developers spend months learning the system's codebase before they can even begin to work on it. Even the most knowledgeable of development teams is fearful of making changes or adding new code that would disrupt operation in some unforeseen way, so the most mundane of changes is discussed ad nauseum.

When things go wrong, operations blames development and development blames QA. Project managers blame the budget and everyone else. The business loses confidence in IT and begins to look for outsourcers to replace the internal team.

Unless you've been living under a rock, you've heard of how microservices can turn this scenario on its head, enabling a new, more agile world in which developers and operations teams work hand in hand to deliver

small, loosely coupled bundles of software quickly and safely. Instead of a single monolithic system, functionality is carried out by a smaller set of services coordinating their operations.

How do you make it work? You've come to the right place. We'll explain it all. While a one-size-fits-all approach to adopting microservices cannot exist, it is helpful to examine the base principles that have guided successful adoption efforts.

Adopting Microservices

One common approach for teams adopting microservices is to identify existing functionality in the monolithic system that is both non-critical and fairly loosely coupled with the rest of the application. For example, in an e-commerce system, events and promotions are often ideal candidates for a microservices proof-of-concept. Alternately, more sophisticated teams can simply mandate that all new functionality must be developed as a microservice.

In each of these scenarios, the key challenge is to design and develop the integration between the existing system and the new microservices. When a part of the system is redesigned using microservices, a common practice is to introduce glue code to help it to talk to the new services.

An API gateway can help combine many individual service calls into one coarse-grained service, and in so doing reduce the cost of integrating with the monolithic system.

The main idea is to slowly replace functionality in the system with discrete microservices, while minimizing the changes that must be added to the system itself to support this transition. This is important in order to reduce

the cost of maintaining the system and minimize the impact of the migration.

Microservices Architectural Patterns

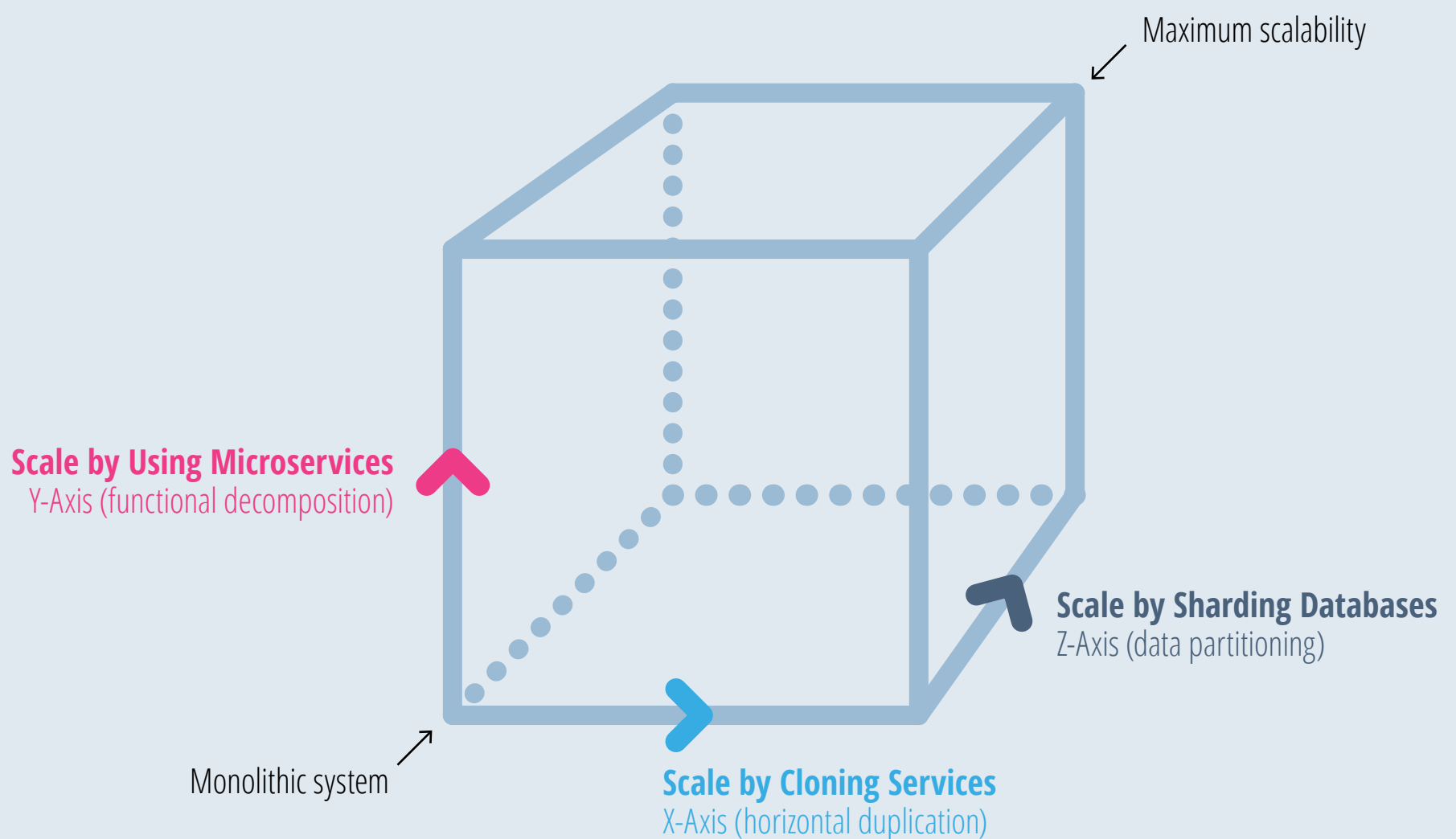
A number of architectural patterns exist that can be leveraged to build a solid microservices implementation strategy.

In their book “[The Art of Scalability](#),” Martin Abbott and Michael Fisher elaborated on the concept of the “[scale cube](#),” illustrating various ways to think about the use of microservices to more easily scale systems (Figure 1). The microservices pattern maps to the Y-axis of the cube, wherein functional decomposition is used to scale the system. Each service can then be further scaled by cloning (X-axis) or sharding (Z-axis).

Alistair Cockburn introduced the “ports and adapters” pattern, also called [hexagonal architecture](#), in the context of building applications that can be tested in isolation. However, it has been increasingly used for building reusable microservices-based systems, [as advocated by](#) James Gardner and Vlad Mettler. A hexagonal architecture is an implementation of a pattern called [bounded context](#), wherein the capabilities related to a specific business domain are insulated from any outside changes or effects.

Examples abound of these principles being put to practice by enterprises migrating to microservices. Click Travel open sourced their [Cheddar framework](#), which captures these ideas in an easy-to-use project template for Java developers building applications for Amazon Web Services. SoundCloud, after a failed attempt at a big-bang refactoring of their application, [based their microservices migration](#) on the use of the

The Scale Cube and Microservices: 3 Dimensions to Scaling



Source: The New Stack. Based on the "The Art of Scalability," by Martin Abbott & Michael Fisher.

THE NEW STACK

FIG 1: Illustrating Martin Abbott and Michael Fisher’s “scale cube” method of scaling systems with functional decomposition.

bounded context pattern to identify cohesive feature sets which did not couple with the rest of domain.

One challenge faced by teams new to microservices is dealing with distributed transactions spanning multiple independent services. In a monolith this is easy, since state changes are typically persisted to a common data model shared by all parts of the application. This is not the case, however, with microservices. Having each microservice managing its own state and data introduces architectural and operational complexity when handling distributed transactions. Good design practices, such as domain-driven design, help mitigate some of this complexity by inherently limiting shared state.

Event-oriented patterns such as [event sourcing](#) or command query responsibility segregation ([CQRS](#)) can help teams ensure data consistency in a distributed microservices environment. With event sourcing and CQRS, the state changes needed to support distributed transactions can be propagated as events (event sourcing) or commands (CQRS). Each microservice that participates in a given transaction can then subscribe to the appropriate event.

This pattern can be extended to support [compensating](#) operations by the microservice when dealing with eventual consistency. Chris Richardson [presented an implementation of this](#) in his talk at [hack.summit\(\)](#) 2014 and shared example code [via GitHub](#). Also worth exploring is Fred George's [notion of "streams and rapids,"](#) which uses asynchronous services and a high speed messaging bus to connect the microservices in an application.

While these architectures are promising, it is important to remember that, during the transition from monolith to a collection of microservices, both systems will exist in parallel. To reduce the development and operational costs of the migration, the patterns employed by the microservices must be appropriate to the monolith's architecture.

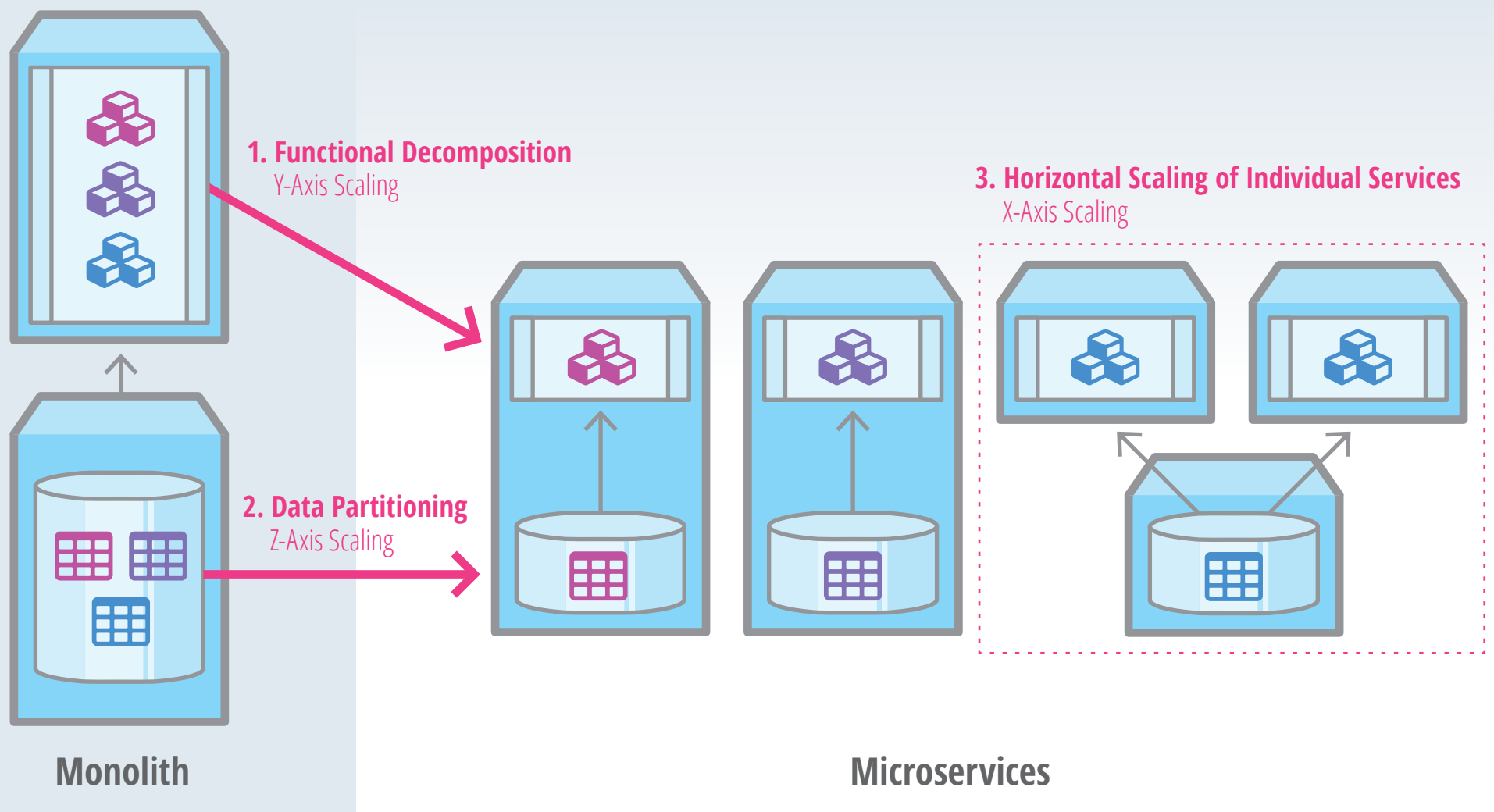
Architectural & Implementation Considerations

Domain Modeling

Domain modeling is at the heart of designing coherent and loosely coupled microservices. The goal is to ensure that each of your application's microservices are adequately isolated from runtime side effects of, and insulated from changes in the implementation of, the other microservices in the system.

Scaling With Microservices

Microservice architectures have 3 dimensions of scalability



Source: The New Stack. Based on a diagram appearing in "Microservices," by James Lewis & Martin Fowler.

THE NEW STACK

FIG 2: With microservices in place as a foundation, additional levels of scalability can be achieved via data partitioning and horizontal scaling.

Isolating and insulating microservices also helps ensure their reusability. For example, consider a promotions service that can be extracted from a monolithic e-commerce system. This service could be used by various consuming clients using mobile Web, iOS or Android apps. In order for this to work predictably, the domain of “promotions,” including its state entities and logic, needs to be insulated from other domains in the system, like “products,” “customers,” “orders,” etc. This means the promotions service must not be polluted with cross-domain logic or entities.

Proper domain modeling also helps avoid the pitfall of modeling the system along technological or organizational boundaries, resulting in data

services, business logic and presentation logic each implemented as separate services.

“All roads to microservices pass through domain modeling.

Sam Newman discusses these principles in his book “[Building Microservices](#).” Vaughn Vernon focuses on this area even more deeply in “[Implementing Domain-Driven Design](#).”

Service Size

Service size is a widely debated and confusing topic in the microservices community. The overarching goal when determining the right size for a microservice is to not make a monolith out of it.

The “[Single Responsibility Principle](#)” is a driving force when considering the right service size in a microservices system. Some practitioners advocate as small a service size as possible for independent operation and testing. Building microservices in the spirit of [Unix](#) utilities also leads to small codebases which are easy to maintain and upgrade.

Architects must be particularly careful in architecting large domains, like “products” in an e-commerce system, as these are potential monoliths, prone to large variations in their definition. There could be various types of products, for example. For each type of product, there could be different business logic.

Encapsulating all this can become overwhelming, but the way to approach it is to put more boundaries inside the product domain and create further services.

Another consideration is the idea of replaceability. If the time it takes to replace a particular microservice with a new implementation or technology is too long (relative to the cycle time of the project), then it's definitely a service that needs further reworking of its size.

Testing

Let's look at some operational aspects of having the monolithic system progressively transformed into a microservices-based system. Testability is a common issue: during the course of developing the microservices, teams will need to perform integration testing of the services with the monolithic system. The idea, of course, is to ensure that the business operations spanning the pre-existing monolithic system and the new microservices do not fail.

One option here is to have the system provide some consumer-driven contracts that can be translated into test cases for the new microservices. This approach helps ensure that the microservice always has access to the expectations of the system in the form of automated tests. The system's developers would provide a spec containing sample requests and expected microservice responses. This spec is then used to create relevant mocks, and as the basis for an automated test suite that is run before integrating the new microservices with the existing system. [Pact](#), a consumer-driven contract testing library, is a good reference for this approach.

Creating a reusable test environment that can deploy a test copy of the entire monolith — and making it available, on-demand, to the microservices teams — is also useful. This eliminates potential roadblocks for those teams and improves the feedback loop for the project as a whole. A common way of accomplishing this is to containerize the entire

monolith in the form of Docker containers orchestrated through an automation tool like [Docker Compose](#). This deploys a test infrastructure of the monolith quickly and gives the team the ability to perform integration tests locally.

Service Discovery

A service may need to know about other services when accomplishing a business function. A [service discovery](#) system enables this, wherein each service refers to an external registry holding the endpoints of the other services. This can be implemented through environment variables when dealing with a small number of services; [etcd](#), [Consul](#) and [Apache ZooKeeper](#) are examples of more sophisticated systems commonly used for service discovery.

Deployment

Each microservice should be self-deployable, either on a runtime container or by embedding a container in itself. For example, a JVM-based microservice could embed a [Tomcat container](#) in itself, reducing the need for a standalone web application server.

At any point in time, there could be a number of microservices of the same type (i.e., X-axis scaling as per the scale cube) to allow for more reliable handling of requests. Most implementations also include a software load balancer that can also act as a service registry, such as [Netflix Eureka](#). This implementation allows for failover and transparent balancing of requests as well.

Build and Release Pipeline

Additional considerations when implementing microservices are quite common, such as having a continuous integration and deployment

pipeline. The notable caveat for this in a microservices-based system is having an on-demand, exclusive, build and release pipeline for each microservice. This reduces the cost of building and releasing the application as a whole.

“ We do not need to build the monolith when a microservice gets updated. Instead, we only build the changed microservice and release it to the end system.

Release practices also need to include the concept of [rolling upgrades](#) or [blue-green deployment](#). This means that, at any point of time in a new build and release cycle, there can be concurrent versions of the same microservice running in the production environment. A percentage of the active user load can be routed to the new microservice version to test its operation, before slowly phasing out the old version. This helps to ensure that a failed change in a microservice does not cripple the monolith. In case of failure, the active load can be routed back to the old version of the same service.

Feature Flags

One other common pattern is to allow for [feature flags](#). A feature flag, like a configuration parameter, can be added to the monolith to allow toggling a feature on or off. Implementing this pattern in the monolith would allow us to trigger the use of the relevant microservice for the feature when the flag is turned on. This enables easy A/B testing of features migrated from

the monolith to microservices.

If the monolith version of a feature and the new microservice replicating the said feature can coexist in the production environment, a traffic routing implementation along with the feature flag can help the delivery teams more rapidly build the end system.

Developer Productivity During Microservices Adoption

Monolithic architectures are attractive in that they allow quick turnaround of new business features on a tight schedule — when the overall system is still small. However, this becomes a development and operations nightmare as the system grows up.

“If working with your monolith was always as painful as it is now, you probably wouldn’t have it. Rather, systems become monoliths because adding onto the monolith is easy at first.

Giving power to developers to choose a “microservices first” approach when building a new feature or system is complicated and has many moving parts. Doing so demands strong disciplines around architecture and automation, which in turn helps create an environment that allows teams to quickly and cleanly build microservices.

One approach to building out this developer infrastructure is to create a standard boilerplate project that encapsulates key principles of microservice design, including project structure, test automation, integration with instrumentation and monitoring infrastructures, patterns like circuit breakers and timeouts, API frameworks, and documentation hooks, among others.

Project templates allow teams to focus less on scaffolding and glue code, and more on building business functionality in a distributed microservices-based environment. Projects like [Dropwizard](#), [Spring Boot](#), and [Karyon](#) are interesting approaches to solving this. Making the right choice between them depends on the architecture and developer skill level.

Monitoring and Operations

Coexisting monoliths and microservices require a comprehensive monitoring of performance, systems and resources. This is more pronounced if a particular feature from the monolith is replicated through a microservice. Collecting statistics for performance and load will allow the monolithic implementation and the microservices-based one replacing it to be compared. This will enable better visibility into the gains that the new implementation brings to the system, and improve confidence in pursuing further migration.

Organizational Considerations

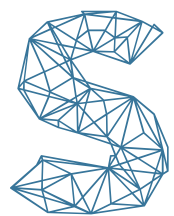
The most challenging aspects of moving from monoliths to microservices are the organizational changes required, such as building services teams that own all aspects of their microservices. This requires creating multidisciplinary units which include developers, testers and operations staff, among others. The idea is to embrace more collective code ownership and care for software craftsmanship.

Conclusion

Most of the ideas shared in this article have either been practiced or have delivered results in organizations of all sizes. However, this is not a one-size-fits-all paradigm. Hence, it's important to keep an eye on evolving patterns and adoption war stories. As more organizations move from monolithic systems to microservices, we will have more to learn along our journey.

THE TEN COMMANDMENTS OF MICROSERVICES

by **JANAKIRAM MSV**



Software development periodically goes through a paradigm shift. In the last two decades, we have experienced the client/server computing model, componentization with component object model (COM) and common object request broker architecture (CORBA), multitier architecture with .NET and Enterprise JavaBeans (EJB), and more recently, web-scale computing.

Microservices 101

With Infrastructure as a Service (IaaS), it is fascinating to implement the concept of scale-out and elasticity on cloud platforms. DevOps can create preconfigured machine images or bootstrap the instances with the dependencies and code. As long as the application servers and web servers are stateless, they can scale from a handful of instances to a few hundred in minutes. Depending on the traffic and the load, the instances either get launched or terminated without much intervention.

This architecture addressed the complex challenge of capacity planning. DevOps could safely define a baseline configuration that avoided underutilization or overprovisioning of resources. Cloud infrastructure made the virtual machine the unit of deployment and execution. An application's scalability factor depended on the ability to rapidly multiply virtual machines (VMs). Though the VM was an ideal choice for the unit of deployment, the unit of code continued to be a module or a component. It was overkill to create a one-to-one mapping between the component and the VM.

With the emergence of containers, the unit of deployment gradually started to shift from away from the VM models. Linux container technologies, such as LXC, Docker, runC and rkt, make it possible to run multiple containers within the same VM. This enables DevOps to package each component or module as a container. Each container has everything—from the OS to the runtime, framework and code—the component needs to run as a standalone unit. Like stateless components, containers can be designed to accept input and send output, if any.

The composition of these containers can logically form an application. The focus of an application becomes orchestrating multiple containers to achieve the desired output.

“ A collection of independent, autonomous containers participating in an application defines the microservices architecture.

This design relies more on containers and less on the underlying infrastructure, such as VMs or physical servers. Since the unit of deployment is a container, there can be more containers per VM or physical server.

Microservices architecture has its roots in proven distributed computing models like COM, CORBA and EJB. Best practices from these technologies are still relevant in the microservices era. Some think of [microservices as service-oriented architecture \(SOA\)](#) with an emphasis on small ephemeral components.

Containers and Microservices

There is a misconception that moving a monolithic application to containers automatically turns it into a microservice. The best way to understand this concept is to think of virtualization and cloud. During the initial days of IaaS, many CIOs claimed they were running a private cloud; but, in reality, they were only implementing virtualization. Commonly, attributes like self-service, elasticity, programmability and pay-by-use differentiated cloud from virtualization.

While microservices may use containerization, not every containerized application is a microservice. This is an important aspect to understand before we proceed to discuss best practices.

1. Clean Separation of Stateless and Stateful Services

Applications composed of microservices contain both stateless and stateful services. It is important to understand the constraints and

limitations of implementing stateful services. If a service relies on the state, it should be separated into a dedicated container that's easily accessible.

One of the key advantages of microservices is the ability to scale rapidly. Like other distributed computing architectures, microservices scale better when they are stateless. Within seconds, multiple containers can be launched across multiple hosts. Each container running the service is autonomous and doesn't acknowledge the presence of other services. This makes it possible to precisely scale the required service instead of scaling the VMs. For this pattern to work seamlessly, services should be stateless. Containers are ephemeral and thus become an ideal choice for microservices.

A microservices-based application may contain stateful services in the form of a relational database management system (RDBMS), NoSQL databases, and file systems. They are packaged as containers with unique attributes. Typically, stateful services offload persistence to the host, which makes it difficult to port containers from one host to another. Technologies, such as Flocker and Docker volume plugins, address this problem by creating a separate persistence layer that's not host-dependent.

Typically, stateful services offload persistence to the host, or use highly available cloud data stores to provide a persistence layer. Both approaches introduce complications: offloading to the host makes it difficult to port containers from one host to another, and highly available data stores trade consistency for availability, meaning that we have to design for eventual consistency in our data model.

Technologies, such as Flocker, help address the host portability problem by creating a persistence layer that's not host dependent. The new cloud datastores, such as Redis, Cassandra, and IBM's Cloudant, maximize availability with minimal delay on consistency. As container technologies evolve, it will become easier to tackle the stateful services problem.

2. Do Not Share Libraries or SDKs

The premise of microservices is based on autonomous and fine-grained units of code that do one thing and one thing only. This is closely aligned with the principle of “don't repeat yourself” (DRY), which states that every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Every service is a self-contained unit of OS, runtime, framework, third-party libraries and code. When one or more containers rely on the same library, it may be tempting to share the dependencies by centrally configuring them on the host. This model introduces complexities in the long run. It not only it brings host affinity, but also breaks the CI/CD pipeline. Upgrading the library or SDK might end up breaking a service. Each service should be treated entirely independent of others.

In some scenarios, the commonly used libraries and SDKs can be moved to a dedicated service that can be managed independently, making the service immutable.

3. Avoid Host Affinity

This point was briefly discussed in the context of shared libraries. No assumptions can be made about the host on which the service would run.

The host includes a directory, IP address, port number, a specific distribution of the OS, and availability of specific files, libraries and SDKs.

Each service can be launched on any available host in the cluster that meets the predefined requirements. These requirements are more aligned with the specifications, like the CPU type, storage type, region and availability zone, rather than the software configuration. Services should function independently of the host on which they are deployed.

In case of stateful services, a dedicated persistent (data volume) container should be considered.

4. Focus on Services with One Task in Mind

Each service must be designed with one task in mind. It may map to one function or a module with a well-defined boundary. This means that there may also be one process per container, but that's not always the case.

Docker encourages the pattern of running one background process/daemon per container. This makes containers fundamentally different from virtual machines. While a virtual machine may run the whole stack, a container owns a subset of the stack. For example, when refactoring a LAMP web application for microservices, the web tier with Apache runs in a dedicated container while MySQL moves to another container.

5. Use Lightweight Messaging Protocol for Communication

There is no hard-and-fast rule on how microservices talk to each other. They can use synchronous or asynchronous channels with any protocol

that's platform agnostic. Each service implements a simple request and response mechanism. It's common for microservices to expose well-known HTTP endpoints that can be invoked through REST API calls.

While HTTP and REST are preferred for synchronous communication, it's becoming increasingly popular to use asynchronous communication between microservices. Many consider the Advanced Message Queuing Protocol (AMQP) standard as the preferred protocol, in this regard. Developing microservices with an asynchronous communication model, while sometimes a little more complex, can have great advantages in terms of minimizing latency and enabling event-driven interactions with applications.

In the market today, RabbitMQ and Apache Kafka are both commonly used message bus technologies for asynchronous communication between microservices. Also, if the message-passing is done on the same host, then the containers can communicate with each other by way of system calls, as they all share the same kernel.

6. Design a Well-Defined Entry Point and Exit Point

In most cases, microservices are treated like a black box, with less visibility into the actual implementation. With inconsistent entry points and exit points, it will be a nightmare to develop a composed application.

Similar to the interface definition in COM and CORBA, microservices should expose a well-defined, well-documented contract. This will enable services to seamlessly talk to each other. Even if a microservice is not expected to return an explicit value, it may be important to send the

success/failure flag. Implementing a single exit point makes it easy to debug and maintain the code.

7. Implement a Self-Registration and Discovery Mechanism

One of the key aspects of microservices is the discovery of a service by the consumer. A central registry is maintained for looking up all available services.

Each microservice handles registration within the central service registry. They typically register during the startup and periodically update the registry with current information. When the microservice gets terminated, it needs to be unregistered from the registry. The registry plays a critical role in orchestrating microservices.

Consul, etcd and Apache Zookeeper are examples of commonly used registries for microservices. Netflix Eureka is another popular registry that exposes registration APIs to services for registering and unregistering.

8. Explicitly Check for Rules and Constraints

During deployment, microservices may need to consider special requirements and constraints that impact performance. For example, the in-memory cache service needs to be on the same host as the web API service. The database microservice may have to be deployed on a host with solid-state drive (SSD) storage. The master and slave containers of the database cannot exist on the same host. These constraints are typically identified during the design of the services.

If rules and constraints are not considered by the SysOps team, services may need to raise alerts or log appropriate messages warning about possible implications and side effects. Under extreme conditions, a microservice may have to shut down if the mandatory rule is not respected at deployment.

9. Prefer Polyglot Over Single Stack

One advantage of using microservices is the ability to choose the best of breed OS, languages, runtimes and libraries. For example, the chat microservice can be implemented in Node.js, exposing the websockets; the web API service can be written in Python and Django; the image manipulation service may be in Java; and the web frontend could be implemented with Ruby on Rails.

As long as each service exposes a well-defined interface that's consistent with other services, it can be implemented using the most optimal technology stack.

With Microsoft adding native container support to Windows, it is also possible to mix and match Linux containers with Win32 and .NET containers.

10. Maintain Independent Revisions and Build Environments

Another benefit of microservices is the ability to code and maintain each service independently.

Though each microservice is part of a large, composite application, from a developer standpoint, it is important to treat each service as an

independent unit of code. Each service needs to be versioned and maintained separately in the source code control system. This makes it possible to deploy newer versions of services without disrupting the application. CI/CD pipelines should be designed to take advantage of the independent versioning.

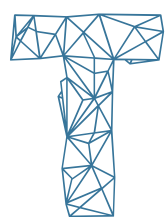
This mechanism makes it possible to implement blue/green testing of each service before rolling out the production version.

Conclusion

While system and application requirements continue to evolve, the methodology behind how we solve these problems is often based on older models and patterns. As mentioned before, microservices architecture has its roots in models like COM, COBRA, EJB and SOA, but there are still some rules to live by when creating microservices utilizing current technologies. While the ten best practices we've laid out here are not entirely comprehensive, they are core strategies for creating, migrating and managing microservices.

HOW MICROSERVICES HAVE CHANGED AND WHY THEY MATTER

by **ALEX WILLIAMS**



he concept of microservices is fueled by the need to develop apps faster, be more resilient and offer a great experience for the customer. It's a concept equated with scaled-out, automated systems that run software on simple, commodity infrastructure. It's the economic efficiencies that containers provide that will make microservices such a major theme in 2016.

The need for fast application development affects the entire organization and how it views the way its business has historically been organized. The new practices that come with microservices means the need for small teams that work iteratively in a manner that is unfamiliar to companies that work in a top-down fashion. This means sweeping changes to how businesses function.

Now we have the container ecosystem emerging as a core theme for a new thinking about application architectures and microservices.

There are some basic tenets to consider about microservices, noted Battery Ventures Technology Fellow Adrian Cockcroft. First, it's now less expensive to build software, and containers have made it even more affordable. Docker is on everyone's roadmap — from software vendors to end users, all trying to figure out how to use containers — because they can accelerate software delivery. But it also means that the systems need to be instrumented at the application level, which means different requirements for developing, deploying and managing applications.

For example, monitoring is more critical than ever for companies dealing with a growing scope of services and stacks. To solve problems, companies have to analyze the data logs — logs that are likely stretched across potentially ephemeral nodes and across multiple services. This need to have granular monitoring and better tooling helps practitioners better grasp how these building blocks are affecting the potential dozens of microservices that the application depends upon.

So what works? It starts with the organization and the API: A microservices-based product team and a separate backend-based platform team [with an API between them](#), where the API calls are made and the infrastructure responds consistently and accordingly.

[Microservices](#) is defined as a loosely-coupled, service-oriented architecture with bounded context. It allows updates without needing to understand how everything else works.

Services are built across organizations, and ownership stays in one place. Microservices architecture serves more as point-to-point calls between systems. You must have flexible message formats; irrespective of the versions, everything still works. That means when building a microservices

architecture, you need some tooling to configure, discover, route traffic and observe and build systems.

IBM's [Andrew Hately](#) offers the context that fifteen years ago, people might check their bank balance once a week. In time, the Internet allowed people to check their balances, and taking that accessibility further, smartphones drove, perhaps, the most change. Today, people can get instant access to every spend on their accounts. That speed and immediacy means that businesses have to respond with services that are developed on the same scale that the social networks and search companies developed their services on over the past five to ten years.

Businesses have to deal with a constant interaction between their employees, customers, systems, and all possible combinations imaginable — fully connected and available all the time, Hately said. That means a reinvention of business processes that require everything to be connected. If you do not experiment, and do not have a way to quickly get features out, then revenues will suffer and you will be irrelevant.

“Instrumentation is critical,” Hately said.

Code is not supported over hundreds of sites, Hately said. The feedback comes in and consumers use that in the next set of test cases. This rigorous development process provides a way to work as a company. It is also a way to think about microservices. It is the ops side of DevOps that will do this. If you have a small piece of code and define metrics for it, you can microsegment it down to what is succeeding and what is failing.

Building off the feedback and success of consumers as well as their own internal teams, IBM combined best practices from agile, DevOps, lean, and other iterative processes to create an enterprise methodology called IBM

Bluemix Garage Method. The IBM Bluemix Garage Method combines the reliability and testability of enterprise solutions with the latest open community best practices about quality at scale, making innovation repeatable, creating continuous delivery pipelines, and deployment on cloud platforms. It's a valuable, open resource for anyone to improve their DevOps skills for individuals, teams and entire organizations, all with governance-compliant management and monitoring abilities.

Contracts Around Software

The first generation of container management platforms are supporting these accelerated development processes.

In Docker Compose, the tooling is facilitated by microservices, said [Scott Johnston](#), senior vice president of product at Docker, Inc. The YAML file acts as a manifest to describe the different components. Compose allows developers to describe multicontainer apps in an abstract fashion. It can describe the web container, database container, load balancer and the logical connections between them. It does not require networking or storage implementation.

Microservices are a contract around software, said Engine Yard's [Matt Butcher](#). Some would argue that they are service-oriented architecture (SOA) done correctly. Developers want usefulness, feature richness and elegance. It returns software development to its Unix roots of doing one thing very well. The output of a command is arbitrary with Unix.

Microservices is more contractual in that it shows how to do one thing very well, but also in how it interacts with an environment. If it works well, it is similar to what can be done with a good Unix shell script.

For example, the Kubernetes manifest file format serves as a contract. The manifest provides the details about the resources needed, the volume definitions, storage needs, etc. That serves as a powerful DevOps-style contract. It tells the developer and the operations professional what to expect. It's not this medieval style of developer and operations relationship that forces the developer to throw the code over the wall.

A manifest may contain metadata about an application, plus descriptive parameters about its specific version, and potentially multiple manifests. This may be an instance, a pod manifest, a replication controller or a service definition — and the known resource locations for constituent files. Arbitrary labels may be defined for components contained in a chart.

“There’s the quintessential problem... where you have the DevOps people who are responsible for running all of this stuff in production, and you have the developers who are responsible for building it, and there’s this handoff process that, all too often, becomes throwing something over the wall.”

— Matt Butcher, Engine Yard

When developers build containers, Butcher said, there's a certain assurance level — provided by the abstraction layer — that those containers will run much the same way in the production phase as in the development phase. This already alleviates much of the headaches among DevOps professionals, who understand the basic instrumentation of containers. Containerization already provided this assurance level, but products like Helm, a new service from Engine Yard, could help to formalize this relationship further, by presenting it from team to team as a kind of contract — one that isn't "thrown over the wall," but instead blows right through it.

From VMs and Monoliths to Containers to Microservices

Containers provide the foundation for cloud-native architectures and symbolize a new form of application architecture compared to what virtualization has traditionally offered, said [Bryan Cantrill](#), chief technical officer at Joyent. Hardware-based virtualization, or traditional VMs, served a time when computing was done on much larger machines.

VMs provided a way for the operations teams to manage large monolithic applications that were "morbidly obese," as Cantrill said, and hardware defined enterprise architectures. The virtual machine sat on top of the substrate, carrying the load of the operating system. Containers, however, have created a new and more agile abstraction.

“The app went on a crash diet.”

— Bryan Cantrill, Joyent

Today, the complexity comes with moving from VMs and monoliths to containers and microservices. Companies struggle with how to make the shift as it requires a different thinking about application architectures, the infrastructure and the overall organization itself.

The objective of Joyent's open source Triton solution is to simplify and accelerate a company's transition to containers and microservices, Cantrill said. It allows developers to simplify architectures. You provision only containers and never provision a virtual machine. You are able to take the cookbook for microservices and deploy it in seconds, because you don't have to do things such as network configuration.

Cantrill said Joyent is a fan of Docker Compose, as it can talk to a single Docker Engine — a Docker remote endpoint implemented by Triton, which virtualizes the entire data center. It allows quick and easy spin up of a full, resilient operating service. "This is the big trend," Cantrill said. "How do we uplevel thinking from containers to services?"

VMware Chief Technical Officer [Kit Colbert](#), looks at the market from a perspective of how to move along the container journey. VMware has been focused on the operations space. It is now developing an approach to meet the new emergence of developers and their needs, but as an infrastructure provider.

For VMware, the company sees itself as an infrastructure provider, not an application-centric, architecturally-oriented company. Colbert sees customers interested in Cloud Foundry, but others that want a DIY approach. VMware is seeking to support application technologies with vSphere Integrated Containers (VIC) and Photon platform.

To accommodate customers using containers, vSphere Integrated Containers (VIC) makes containerized workloads first-class citizens on vSphere. VIC fits on the run side of the development process, and applies one of the most valuable aspects of virtualization to containers: flexible, dynamic resource boundaries. With virtualization, VMware turned commodity hardware into simple, fungible assets. Likewise, by applying a Docker endpoint within a virtual machine, vSphere Integrated Containers create virtual container hosts with completely dynamic boundaries. The result is an infrastructure supportive of both traditional and microservices-based applications, with accessibility to both IT and developers.

By contrast, VMware's Photon Platform is intended specifically for cloud-native applications. Comprised of a minimal hypervisor and control plane, Photon Platform is focused on providing speed and scale for microservices. Photon Platform has also been designed for developer ease of use via APIs, giving them a self-service platform with which to provision applications and, ultimately, speed deployment.

From VMware's perspective, operations teams are also pushing to make deployment faster. It's now more about the digital experience, or how the software is more functional, than anything else. It's comparable to how we view the apps we use on our smartphones. The provider may be known for the great sound of the speakers, but is the app for the service functional?

"Can I rely on it?" Colbert asked. Companies have to figure out how to build apps in order to serve the customer, who is continually seeking out the quality app. Companies need to figure that out in order to move faster. For many customers who have built-out, virtualized infrastructure, they

are looking to meet organizational challenges with the advent of this faster application development process.

Development in the Time of Microservices

Software development is iterative and requires continual feedback loops to work. This is what tools such as the IBM Bluemix Garage Method offer. But most organizations work according to a model that is different than the way the developer works. Developers do not work in the same manner as people in sales, marketing, finance, etc., who work according to a plan, a schedule. In software development, the process is much more iterative, and not top-down.

“I don’t know what to call this but the real world and software world impedance mismatch,” said Pivotal Principal Technologist Michael Coté. Coté argues that figuring out how software development works can seem paradoxical, but it does prevent people from trying to understand how everything works in one giant machine and according to one document. By following the principles of software development, organizations are allowed to find their way, as opposed to staying rigidly attached to one plan.

There is no one way of doing microservices, Coté said. With microservices you get runtime and architectural resiliency. Microservices build upon simple principles to build really complex things. The simpler you can create concepts, the greater the complexity of things you can make.

But what happens when the complexity shifts to some place else? With Pivotal, the platform manages the complexity. It takes away the choices so the customer does not have to think about the networking, operating

systems and such. It allows the customer to put the complexity at the top of the application stack where they can better differentiate their offering for the end-user.

“We’re seeing another renaissance moment in the technology industry.”

— Andrew Hatelly

Likewise, IBM Bluemix Garage Method aims to abstract the complexity away from the developer, with the aim of making them efficient and let them better enjoy the jobs they are actually there to do. All of these efforts are adding up to big changes in the enterprise, both on the technical and cultural levels.

CREATING STANDARDS FOR THE CONTAINER ECOSYSTEM



This discussion with Josh Ellithorpe at DockerCon EU in Barcelona revolves around standardization within the Docker and container ecosystem, with a focus on the importance of compatibility with standardized image formats. Making containers and tooling work for the entire ecosystem is only possible by controlling environments and standards, and you'll hear more about the difficulties and obstacles to those goals in this podcast.

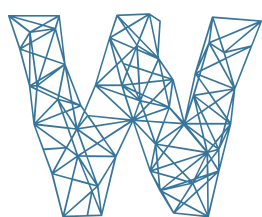
[Listen on SoundCloud](#) or [Listen on YouTube](#)



Josh Ellithorpe, lead software architect at Apcera, is a Chicago native who began his career in the late nineties working in all aspects of the tech stack. As an open source advocate, he released his first open source project, *Throttled Pro*, in 2001. Specializing in Ruby development, Josh decided to acquaint himself with San Francisco's tech scene, and made the city his home. After relocating, Josh worked on some of the biggest emerging social applications for companies like Facebook and Involver. He's now joined Apcera to revisit his networking roots and revolutionize the cloud.

CONTAINERS IN PRODUCTION, PART I: CASE STUDIES

by **MARK BOYD**



What do music streaming service Spotify, video streaming company DramaFever, Backend as a Service Provider Built.io, and the education think-tank “Instituto de Investigacion, Innovacion y Estudios de Posgrado para la Educacion” (IIIEPE) have in common? They all depend on containers to run critical web infrastructures.

Their stories illustrate some of the benefits and challenges of utilizing microservice architectures, creating web-scale applications, and running containers in production.

Running a Global Application at Scale

Spotify has a whole host of features, including logging in with social media credentials, seeing recommendations based on previously listened to tracks, and sharing music. But it’s playing the music itself that is most important. To keep Spotify’s 60 million users happy, the music must keep

streaming even if, say, a developer introduces a bug into the album cover art feature, or one of its more than 5,000 production servers in its four different data centers bites the dust.

To accomplish this, the company packages each discrete feature of Spotify as a microservice. Containers are then used to allow Spotify to bundle some of these microservices together so that a single request call can take all the relevant information and display it in the interface that an end user — and possible subscriber — sees, without making too many repeat calls to match specific information. This also allows other microservices to run independently at the same time.

At DockerCon 2014 in San Francisco, Spotify software engineer Rohan Singh mapped out Spotify's game plan for packaging microservices into a stream of containers that can be spun up and down, depending on user needs, around the globe. Singh's talk at DockerCon coincided with the first week that Spotify was planning to deploy containers in production. Nine months later — in March 2015 — Spotify's Evgeny Goldin took the stage at DevCon in Tel Aviv to provide on-the-ground insights into how containers are central to the streaming service's continuous delivery culture.

Goldin says Spotify's problem was what he calls NxM: "You have 'N' services and you want to deploy them on 'M' hosts," he told the DevCon audience.

The NxM problem refers to the problems at scale that occur when a global audience of users are all using Spotify in their own ways, turning on and off features, playing music, or shutting down their app usage for the night. So, as a Spotify user's app requires a particular microservice, it makes a request to Spotify's servers and gets the data it needs. As new features are added, or more users are simultaneously asking for their own app services

to start, Spotify needs to spin up more containers to meet the additional demand, and then drop them when there isn't the need to maintain processing power on the server.

To address this, Spotify created the [open source project Helios](#).

“We call it a Docker orchestration framework. It solves one problem, and it solves it really well,” says Singh. “Given a list of hosts and given a list of services to deploy, it just deploys them there.”



Evgeny Goldin

Spotify software engineer

Containers and the Helios model has been used in production at Spotify since July 2014, and is a key pillar in their scalability strategy. “We’re already at hundreds of machines in production,” the team announced in their Helios README notes on GitHub. “But we’re nowhere near the limit before the existing architecture would need to be revisited.”

DramaFever’s Video Demands

DramaFever runs white-label sites such as AMC’s SundanceNow Doc Club and horror streaming site Shudder, and has contracts to deliver its content to other streaming services. Kromhout, who worked in site reliability at DramaFever before her recent move to work as a principal technologist for

Cloud Foundry at Pivotal, said of her time there: “We are not Netflix, we are not Hulu, but if you watch a Korean drama on Hulu, they are streaming content they licensed from DramaFever.”

During her work at DramaFever, Kromhout took a “[cattle, not pets](#)” mindset to server architecture, with everything in the request path containerized to enable autoscaling up and down. DramaFever started using Docker in production in October 2013, at version 0.6. Using Docker enforces consistent development and repeatable deployment; containers help keep the code and configuration environments used by developers the same, even as they move code from their laptops through to the cloud-based QA, staging and production environments. The code in the containers can be trusted every time the architecture autoscales a new instance of the application or microservices components, as it is going to have the same operational context as any testing or development arrangement.

“When I started in July 2014, using Docker in production was a sign that it would be an exciting place to work.”



Bridget Kromhout

Co-organizer of DevOps Days and co-host of the podcast “Arrested DevOps”

One of the initial obstacles DramaFever had to address in autoscaling sufficiently at a production scale has been building a robust container

registry. Kromhout says that in a distributed architecture, you often have a situation where multiple containers are launching at the same time.

This is done dynamically, so the architecture is only firing up instances and launching containers as needed based on usage. What happens is that these ‘docker pull’ requests are sent to the private Docker registry. The registry tended to fall over when around twenty containers were being started at the same time.

To solve this problem, DramaFever runs the registry container everywhere, on each host locally, so that the image pull is done locally, with AWS S3 serving as the backing storage.

“DramaFever could have gone in the direction of scaling up,” Kromhout says. But she says that Tim Gross, director of operations at DramaFever, realized that “if he had to scale the registry, why not do so in a way that you would never have to think ‘can our registry server handle this many image pulls?’”

She admits this is a unique situation, but it’s exactly the sort of problem that you’ll face running containers at scale with a self-hosted registry. Kromhout spoke about DramaFever’s container strategy story at OSCON 2015, “[Docker in Production: Reality, Not Hype](#)”.

Managing Containers Via API in Production

Nishant Patel, CTO of Integration and Backend as a Service (BaaS) provider [Built.io](#) has been using containers in production for almost two years now, and believes that it is helping them differentiate the company’s service offering from other providers in their market.

Patel says that for the most part, the majority of IaaS and BaaS customers are really looking for a database-as-a-service solution with out-of-the-box authentication and similar features which enable the database to be provided to end customers as an application. But 20% of their customers require something a bit more rigorous, with additional business logic wrapped around the database. Built.io's customers need to be able to write the custom logic and have it added to built.io's servers.

"We didn't want it to get too complicated for our customers to have to set up their own server, etc. So we wanted an easy way to let our customers upload their custom code and run it alongside their database. So, essentially, we wanted a container in the cloud," says Patel.

That led built.io's Mumbai-based engineering team to look at available options and two years ago, one of their engineers came across Docker when it was still at about 0.9 version release. "We did a quick proof of concept and set up these containers, uploaded code and you know what, we were pretty impressed with what it provided. It was a secure container that didn't mess with anyone else's code. So then we went a step further and looked at writing our own management layer."

Patel estimates that built.io's customers spin up thousands of containers at any given time, as each customer's account by default comes with a Docker container.

"It gave us a competitive advantage to go with containers. Our competitors had the same requirements as us: they also have to cater to these 20% of customers with specific needs, but because we use Docker we were able to create a platform as a service which is something our competitors couldn't do. It made our story much more powerful. Using

our MBaaS with Docker in production, you can upload full node.js applications, for example.”

Like Spotify and DramaFever, Built.io found existing tools lacking, given the early adoption nature of their containers-in-production environments. This led to Built.io choosing to build their own management layer in the same way that Spotify built their own orchestration service and how DramaFever built a horizontally scalable host-local Docker registry architecture.

“What we wanted was the ability through APIs is to make an API call to set the containers up. Then we built all sorts of stuff, for example, we wanted higher paid customers to be able to set up bigger containers. We wanted customers to be able to have a load balancer container, we wanted to add additional security provisions, and enable through the API to manage starting and stopping containers, and using an API to get container logs and put it into our customer management UI.”

Patel says built.io continues to keep an eye on the container ecosystem for other tools that could take over the management layer for them, and while [Deis](#) comes close, for now they have decided to continue doing this themselves.

Leveraging Containers to Introduce Continuous Delivery

There is no shortage of those wanting to experiment with using containers in production, as Luis Elizondo found after he published [a series of blog posts on his Docker production workflow](#). Elizondo works for the IIIPE — a government-funded agency dedicated to education research — where

he manages everything from small static HTML sites to full-blown learning management systems for teachers.

“ We have 25 web applications, and most of them are run on Drupal, but we also use Node.js and Python. The biggest problem is that we have a lot of issues when we are moving a web app from dev to production.”



Luis Elizondo

IIIEPE software developer

Elizondo was looking for solutions that enabled use of their private cloud infrastructure that could scale horizontally. They expected their web applications to grow in both number and usage, so Docker was an appealing option for infrastructure solutions that could scale with their intended growth.

At the time, IIIEPE wasn't using continuous delivery methodologies in their application architecture, so moving to a Docker container infrastructure also gave them the chance to implement new continuous delivery workflows in conjunction with their orchestration flow.

Elizondo's blog series documented his approach to storage, orchestration, service discovery and load balancing in production as a way of sharing the workarounds and challenges he had to solve in making his workflow production-ready.

The team has two servers. One runs MaestroNG and is responsible for starting stock Docker containers. The other runs a virtual machine running Jenkins. “We use it as a kind of robot that performs repetitive work,” he says. “We programmed Jenkins so that when we push a new change to any of the projects, Jenkins detects the new change, rebuilds the whole image, adds the code to the new image, then orders MaestroNG to pull the image from Docker Hub, and that’s basically our workflow.”

Elizondo used the DigitalOcean platform to test the new architecture and container workflow towards the end of 2014 and, amazingly, it only took about a month and a half to test it and work out all the kinks.

“The biggest discovery in our experiment was that in our containers and in our Git repos, this workflow doesn’t just work for Drupal,” he says. “It also works for Node.js applications. I also tested it with our Ruby application. So it is a standard workflow. Anyone can use it.”

Managing Container Complexity with Kubernetes in Production

For Elizondo, the solution to containers in production was Docker. Elizondo says he had experimented with Kubernetes, for example, but found it too complex for his needs. But for others, such as the core engineering team at e-commerce giant Zulily, the reverse has been true. Zulily first began experimenting with containers in production in May 2014, according to Rachael King at the Wall Street Journal. King reports that Zulily’s software lead Steve Reed said at OSCON in July 2015 “the hardest part is operating the container, especially in a production environment.”

When first assessing the benefits of containers in production, Zulily ended

up shelving their plans for a Docker-based production use case due to the complexity inherent in orchestration. With the maturing Kubernetes platform now able to manage orchestration of Docker containers, however, Zulily has been able to return to their production goals. “Kubernetes is production ready, even for a deployment like ours at Zulily where we’re not talking about hundreds of nodes,” Reed now says.

Conclusion

At the start of 2015, using containers in production was seen as either an experiment or a bold choice for enterprise. Now, only twelve months later, containers in production are being deployed not just for pilot or specific projects, but are being woven into the architectural fabric of an enterprise.

MICROSERVICES FRAMEWORKS FOR HANDLING COMPLEXITY AT SCALE



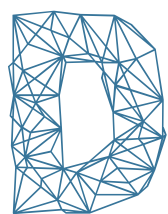
In this interview, Ken Owens discusses the role of microservices in developer and operations teams, and how practices are emerging around microservices, scheduling, and service discovery — but currently they're still dependent on the company creating these practices, and the tools they're using. Ken talks about solving some of these complexities with Mantl, which serves to help DevOps teams working with both enterprise and small-scale applications. [Listen on Sound Cloud](#) or [Listen on YouTube](#)



Ken Owens is chief technical officer of Cloud Infrastructure Services (CIS) at Cisco Systems. Ken is responsible for creating and communicating technical/scientific vision and strategy for CIS business. He brings a compelling view of technology trends in enterprise IT (e.g., infrastructure, computing, SaaS, virtualization and cloud) and evangelizes the technology roadmap for the business. Before joining Cisco in 2014, Ken spent over 7 years at Savvis as the chief scientist, chief technical officer, and vice president VP of Security and Virtualization Technologies.

THE DEVELOPERS AND COMPANIES SHAPING THE OPEN SOURCE CONTAINER ECOSYSTEM

by **LAWRENCE HECHT**



Developers have come to rely on code from many different sources as they build out microservices to be deployed on containers.

While not all of that code may be free and open source software (FOSS), there is significant reliance on prominent open source projects. In fact, when The New Stack surveyed executives of 48 container ecosystem companies in 2015, 46 said their technologies depended on open source.

Companies are now asking themselves if, and how, they should utilize these open source projects. If you are an enterprise developer, your decisions about what microservice functionality to add will likely be based on your existing infrastructure. If you are a vendor, your technology choices may be related to your partner relationships. Few people can now claim that open source projects are inherently anarchic, insecure or not ready for enterprise-level implementation.

So, how, exactly, can someone evaluate an open source project?

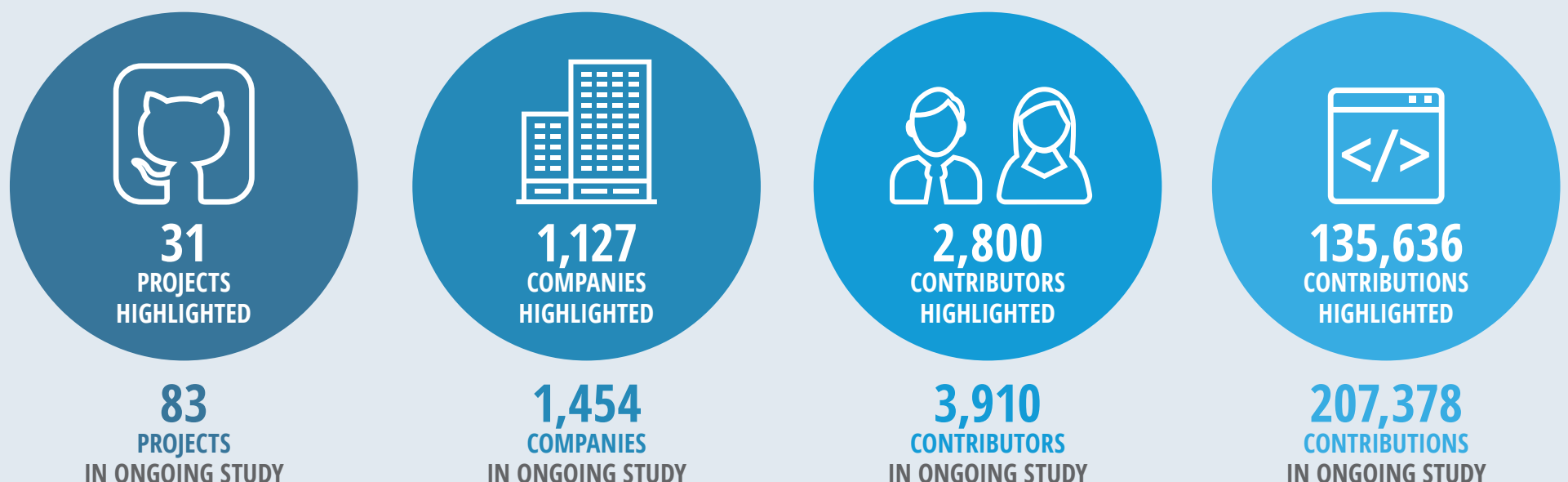
Of course, technical merit is an essential requirement. Without a clear cost-benefit win, most projects won't even get short-listed. Other criteria, like an open source community's "health," also become more important. There are many ways to determine if an open source community is healthy, but the essential factors are governance, the people doing the work, the organizations providing support, and dependencies on other projects and technologies. Consistent development activity and increasing numbers of committers are good signs. Financially speaking, sponsorship from either a nonprofit foundation or a corporation is an indication that developers will not abandon the project in the short-term.

Looking at these variables, a developer can better evaluate the risk of a project producing inadequate code or dissipating altogether.

In Book 1 of this series, The New Stack included 71 open source projects in its directory. Since then, we have continued to monitor these projects and have now collected data about 83 projects with 3,910 contributors.

FIG 1: For ongoing study, the median developer is providing two contributions for all the projects covered. Yet, due to the most active participants, the average number of total contributions is 53.

Stats for Open Source Projects Covered by The New Stack



We have chosen to highlight 31 projects in this chapter that relate to the deployment of container-based microservices.

Most projects have a small group of contributors from only a handful of companies. This is part of the story. Often, there is a large total amount of companies that contribute code, but in a far more limited way than the people from the organizations leading the project.

Overall, we will look at three metrics:

- Contributions coming from the leading contributors.
- Contributions coming from the leading companies.
- Contributors coming from the leading companies.

Using these metrics, The New Stack developed a way to rate how leading companies compare to the total pool of contributors. This provides an added dimension to create a panoramic view of the project and open it up to further exploration.

Each variable is given a name and a score. Using this system, three scores are applied to each open source project that The New Stack has researched. The scores show the range and depth of the project according to the variables we have defined. A company participating in an open source project may have different scores based upon the company's leadership position and the developers that are making commits.

The Animal Factors

We present these three variables as factors that represent the methodology behind our assessments — two of which are existing

methodologies, and one an original research product from The New Stack. We think these three factors together represent a holistic view of an open source project, a combined view that we call the Animal Factor Index. The three animal factors are:

Pony Factor

The Apache Software Foundation's Daniel Gruno describes the first score as the [Pony Factor](#). For each project, it equals the lowest number of **contributors** whose total number of **contributions** constitute the majority of the codebase. A project with a low Pony Factor gets most of its contributions from just a few people, whereas a project with a high Pony Factor has a larger number of contributors.

Elephant Factor




The second measure of diversity has been termed the [Elephant Factor](#) by open source, community metrics company [Bitergia](#). For each project, it equals the lowest number of **companies** whose total **contributions** from their employees constitutes the majority of the contributions. A project with a low Elephant Factor gets most of its contributions from just a few companies, whereas a project with a high Elephant Factor has a larger number of companies contributing to the codebase.




Whale Factor

The concentration of developers employed by specific companies is something that we will henceforth call the Whale Factor. For each project, it equals the lowest number of **companies** whose employees constitute a majority of the project's **contributors**. A project with a low Whale Factor gets most of its contributors from just a few companies; whereas a project with a high Whale Factor gets more of its contributors from a larger number of companies.

The Animal Factor Index

Looking at these three factors together produces a holistic view of open source projects — a view that builds upon the first two existing factors and allows for further conclusions to be made about individual projects. When viewing this index, we encourage you to compare the first two factors (Pony and Elephant) to the additional dimension added by the Whale Factor. We believe that this added layer of depth reveals that many projects have a larger, more varied community than it first appears. Regardless, it allows greater distinction between open source projects and how we view them.

The Animal Factor Index of Projects Researched			
Project Name	Pony Factor 	Elephant Factor 	Whale Factor 
	# of contributors whose combined contributions constitute a majority	# of companies whose combined contributions constitute a majority	# of companies whose combined contributors constitute a majority
appc Spec	1	1	11
Aurora	2	1	7
Brooklyn	2	1	8
Chronos	4	1	21
Cloud Foundry Release	23	1	1
Crate	3	1	2
Deis	2	1	51
Docker	8	1	88
Docker Engine for SmartDataCenter	3	1	1

The Animal Factor Index of Projects Researched			
Project Name	Pony Factor 	Elephant Factor 	Whale Factor 
	# of contributors whose combined contributions constitute a majority	# of companies whose combined contributions constitute a majority	# of companies whose combined contributors constitute a majority
etcd	2	1	30
Flannel	1	1	2
Fleet	1	1	8
Flocker	3	1	1
Kubernetes	12	1	33
LXD	2	1	12
Mantl.io	2	1	12
Marathon	4	1	40
Mesos	2	1	25
OpenStack Magnum	5	3	10
Pachyderm	1	1	3
Packer	1	1	141
Photon	3	1	1
Powerstrip	1	1	2
Project Calico Docker	3	1	1
Rancher	1	1	1
rkt	3	1	19
runC	4	2	15




The Animal Factor Index of Projects Researched			
Project Name	Pony Factor 	Elephant Factor 	Whale Factor 
	# of contributors whose combined contributions constitute a majority	# of companies whose combined contributions constitute a majority	# of companies whose combined contributors constitute a majority
Serviced	5	2	1
Tsuru	2	1	6
Vagrant	1	1	163
Weave	2	1	2

FIG 2: Although projects usually get most of their **contributions** from employees of one company (see the Elephant Factor), the Whale Factor goes deeper by looking at a measure of how many other companies are represented through **contributors**. Combined with the Pony Factor, these three scores comprise the Animal Factor Index.

We encourage readers to view the [public data set](#). Feel free to leave comments and utilize the data we’ve collected. Our primary goal in creating the Whale Factor, and referring to the three factors together as the Animal Factor Index, is to inspire others to pursue additional lines of research and investigation.

Data Collection Methodology

Data about project contributors was collected using a tool called [Blockspring](#) that accessed the [GitHub API](#) to pull information about contributors to specific repositories. Although each project has multiple repositories, The New Stack chose to focus on the primary repository for each. Data for 70 projects was collected in late August 2015, with the contributor information for the other 13 projects being collected or updated in November, December and January.

Since GitHub does not identify a contributor's employer, we identified this information as follows: The New Stack used company domain names that were in the email or website fields. However, because a majority of contributors provided Gmail addresses or no email address at all, we used other means to identify their employer's name and username across several social networks and databases. [Clearbit](#) and [FullContact](#) APIs were also used to collect information.

Blockspring, for instance, has an algorithm that cross-checks a person's email address. While none of these methods are perfect, they were accurate a vast majority of the time. For those people that still did not have company information, every personal website that was provided was reviewed. Additionally, if a real username was provided, a search for the person on LinkedIn was conducted and then verified that their picture and other information was similar to what was included on their GitHub profile. For the 83 projects being tracked, we identified 78 percent of the contributors, which account for 93 percent of contributions.

Note that the number of contributions reviewed differs from that seen on GitHub's own dashboards, because of how we counted contributions from merged repositories and those handled by bots.

The Contributors

The heart and soul of open source is its people. Believers in the "great man" view of history will want to start by looking at a project's contributors. No credible discussion of Linux can avoid Linus Torvalds and no analysis of container-based open source projects can ignore Solomon Hykes. As the heart and soul of a project, core contributors are a reason projects succeed or fail.

Using data pulled from GitHub, we can track the number of commits and lines of code submitted, yet tracking those raw numbers fails to measure how important those contributions are. The context of these contributions, along with the dynamics of ticket resolution, can show how healthy relationships are within the community.

Another way to measure contributors' influence on a project's success is to look at their relationships with other developers, place of employment and involvement with other projects. We identified developers that contributed to the greatest number of projects among the 83 projects being investigated (see Figure 2), but it is beyond the scope of this chapter to do a comprehensive social network analysis. Unseen in the table included below is that developers often work together on multiple projects. Reasons why they are involved in the same projects include affinity of interest, employment by the same company and because they have already established a good working relationship.

Developers With 700+ Contributions to at least 4 of 83 Projects

Contributor (# of Projects)	Employer	Project Names			# of Contributions
Brandon Philips (11)	CoreOS	/coreos/etcd /coreos/rkt /appc/spec /coreos/fleet	/docker/docker /opencontainers/specs /coreos/flannel /kubernetes/kubernetes	/opencontainers/runc /mitchellh/vagrant /docker/swarm	915
Jonathan Boule (9)	CoreOS	/coreos/rkt /coreos/fleet /appc/spec	/coreos/etcd /apache/aurora /opencontainers/specs	/coreos/flannel /kubernetes/kubernetes /docker/docker	2019
Jonathan Rudenberg (7)	Flynn	/flynn/flynn /docker/docker /opencontainers/runc	/coreos/flannel /mitchellh/vagrant	/docker/swarm /opencontainers/specs	2488
Brian Waldon (7)	CoreOS	/coreos/fleet /coreos/etcd /coreos/flannel	/coreos/rkt /kubernetes/kubernetes	/appc/spec /mitchellh/vagrant	1941

Developers With 700+ Contributions to at least 4 of 83 Projects

Contributor (# of Projects)	Employer	Project Names			# of Contributions
Jessie Frazelle (7)	Docker	/docker/docker /docker/notary /opencontainers/runc	/docker/docker-bench- security /progrum/dokku	/docker/compose /mitchellh/packer	900
Michael Crosby (6)	Docker	/docker/docker /opencontainers/runc	/opencontainers/specs /docker/docker-bench- security	/mesosphere/marathon /docker/libnetwork	2261
Sven Dowideit (6)	Docker	/docker/docker /docker/machine	/docker/swarm /kitematic/kitematic	/docker/compose /opencontainers/runc	700
Victor Vieux (5)	Docker	/docker/docker /docker/swarm	/opencontainers/runc /flynn/flynn	/docker/libcontainer /docker/machine	2123
Alexander Morozov (5)	Docker	/coreos/flannel /docker/docker	/opencontainers/runc /opencontainers/specs	/docker/libnetwork	918
Tianon Gravi (5)	InfoSiftr	/docker/docker /docker/boot2docker	/opencontainers/runc /docker/swarm	/docker/machine	847
Francisco Souza (4)	Globo.com	/tsuru/tsuru /docker/docker	/kubernetes/kubernetes	pachyderm/pachyderm	4714
Solomon Hykes (4)	Docker	/docker/docker /docker/swarm	/flynn/flynn	/opencontainers/runc	1500
Evan Hazlett (4)	Docker	/docker/machine /shipyard/shipyard	/docker/docker	/docker/boot2docker	1479
Victor Marmol (4)	Google	/kubernetes/kubernetes /google/cadvisor	/opencontainers/runc	/docker/docker	1288
Matthew Fisher (4)	Engine Yard	/deis/deis /docker/swarm	/flynn/flynn	/coreos/fleet	1222
Clayton Coleman (4)	Red Hat	/kubernetes/kubernetes /openshift/geard	/google/cadvisor	/docker/swarm	1213
Lewis Marshall (4)	UK Home Office	/flynn/flynn /progrum/dokku	/docker/docker	/opencontainers/runc	914

FIG 3: Jonathan Rudenberg, Lewis Marshall and Matthew Fisher worked together on Flynn. Many of these contributors are still involved with runC.

This network of relationships and shared projects is something The New Stack will look at in the future to help identify the future possibilities of OSS projects.

Like most open source projects, container-related ones usually have just a few developers that provide most of the effort. Looking at our subset of 31 projects listed above in Figure 2, eight had a Pony Factor of one, and nine had a Pony Factor of two. In other words, over half the projects under investigation had more than half of their contributions coming from one or two people. Interestingly, Cloud Foundry and Kubernetes, both projects initiated within companies (Pivotal and Google) have high Whale Factors, as contributions come from a relatively diffuse set of developers.

The Companies

Just because most of the work and contributions come from just a few companies does not mean the project is closed source. Technically, that is defined by the license being used. Remarkably, we found that most of the projects in this space use the Apache License, Version 2. As we discussed in “[How Open Source Communities Power Docker and the Container Ecosystem](#)” and “[Developing a Methodology for Analyzing Open Source Communities](#),” governance rules and boards may also impact how “open” a project is. Yet, a concentration of power, whether due to financial backing or by the number of employees a company has involved, can influence the direction of a project. This is not necessarily bad. As a general rule, the fewer companies and developers involved, the more agile the project.

For this chapter we selected 31 prominent open source projects that many container-related companies and solutions rely on. Immature projects

were excluded because we only reviewed projects with current activity and more than just a couple of contributors. Consequently, many repositories managed by independent developers and hobbyists are not part of this analysis. While there is an inherent selection bias in this approach, in reality this becomes a screening process in which we only evaluate open source projects that have already been popularized, whether because of technical merit or sponsorship.

Another similarity we saw was that 90 percent (28 of 31) of the projects have over half of their “contributions” coming from employees of just one company. Not surprisingly, Rancher Labs, Pachyderm, Crate.io and Weaveworks account for 89 percent or more of their eponymously named projects. Yet, that is just one of several measures of openness regarding a project’s relationship with a company. Another factor is the number of people participating in a project. In this regard, there is more variation, with only 26 percent (8 of 31) of projects having a majority of its “contributors” coming from one company. Across all these projects, there are some that started in the open source community, and others that began as corporate projects which were later opened up.

Contributions From Developers and Associated Companies					
Project Name and # of Contributions		Leading Employer and % of Contributions		Second Leading Employer and % of Contributions	
OpenStack Magnum	733	IBM	26%	Cisco	13%
runC	1,657	Docker	36%	Red Hat	20%
ServiceD	5,955	Zenoss	41%	StackEngine	28%
Mesos	4,838	Mesosphere	50%	Twitter	22%
Mantl.io	1,659	Asteris	51%	Mesosphere	25%

Contributions From Developers and Associated Companies					
Project Name and # of Contributions		Leading Employer and % of Contributions		Second Leading Employer and % of Contributions	
rkt	1,922	CoreOS	52%	Endocode	29%
Cloud Foundry Release	3,953	Pivotal	53%	IBM	11%
Chronos	766	Mesosphere	55%	Airbnb	11%
Docker	17,116	Docker	58%	Agrarian Labs	7%
Marathon	3,394	Mesosphere	66%	Facebook	11%
Kubernetes	16,622	Google	72%	Red Hat	15%
appc Spec	785	CoreOS	74%	Red Hat	4%
Docker Engine for SmartDataCenter	628	Joyent	74%	Human API	23%
Packer	4,857	HashiCorp	75%	Google	4%
Powerstrip	76	ClusterHQ	82%	Red Hat	14%
Vagrant	7,465	HashiCorp	83%	PetroFeed	2%
Brooklyn	11,213	Cloudsoft	84%	Hazelcast	2%
Deis	4,357	Engine Yard	85%	SOFICOM SpA	3%
Tsuru	9,869	Globo.com	85%	FundedByMe	12%
Flocker	12,472	ClusterHQ	86%	Red Hat	0.02%
etcd	6,311	CoreOS	87%	Red Hat	5%
Flannel	351	CoreOS	87%	Red Hat	7%
Crate	4,448	Crate.io	89%	Lovely Systems	9%
Weave	3,172	Weaveworks	89%	VMware	10%
Aurora	2,723	Twitter	90%	CoreOS	3%
LXD	1,855	Canonical	90%	JR I.T. Solutions	6%

Contributions From Developers and Associated Companies					
Project Name and # of Contributions		Leading Employer and % of Contributions		Second Leading Employer and % of Contributions	
Project Calico Docker	1,339	Metaswitch Networks	91%	Red Hat	1%
Photon	677	VMware	91%	EMC	9%
Pachyderm	1,458	Pachyderm	96%	Dakis	1%
Fleet	2,325	CoreOS	97%	Red Hat	1%
Rancher	640	Rancher Labs	98%	Spirula Systems	0.47%

FIG 4: *Most projects receive a majority of their contributions from employees of one company.*

As pointed out above, all but three projects receive a majority of their contributions from employees of one company. OpenStack Magnum and runC had the lowest concentration of contributions coming from one company.

If we look past the Elephant Factor, there is more variation regarding the percentage of contributors employed by a single company. For example, one of the biggest Whales is Docker — 88 companies represent half of its contributor base. Although many of these developers are only making one contribution, this is a signal that they are engaged enough in the community to get their commits accepted. The project with the highest Whale score was Vagrant, a mainstay of many open source developers. Although HashiCorp employees make most of the contributions, the project has open source credibility as evidenced by so many people buying into the project enough to actually contribute to it.

There are many smaller companies in the ecosystem that are providing most of the engineering effort behind a project.

Contributors and Associated Companies					
Project Name and # of Contributors		Leading Employer and % of Contributors		Second Leading Employer and % of Contributors	
Packer	329	HashiCorp	3%	Google	2%
Vagrant	413	Red Hat	3%	HashiCorp	2%
Deis	138	Engine Yard	9%	Mozilla	2%
Docker	466	Docker	9%	Red Hat	8%
OpenStack Magnum	62	IBM	10%	Cisco, Hewlett-Packard (tie)	8%
runC	127	Docker	12%	Red Hat	11%
rkt	75	CoreOS	15%	Red Hat, Endocode (tie)	5%
Mantl.io	41	Cisco	15%	Asteris	7%
LXD	33	Canonical	18%	Bearstech, DrEd Online Doctor EMC, JR IT Solutions, Kismatic, Mayflower, PagerDuty, Plusnet, Sisa-Tech, VMware, XACT (tie)	3%
Chronos	78	Mesosphere	18%	Airbnb	5%
Kubernetes	425	Google	18%	Red Hat	16%
Marathon	141	Mesosphere	20%	Disqus, eBay, Google, Shopify (tie)	1%
Mesos	100	Mesosphere	20%	Twitter	6%
etcd	221	Red Hat	21%	CoreOS	9%
Fleet	53	CoreOS	25%	Red Hat	9%
Flannel	30	CoreOS	27%	Red Hat	23%
Brooklyn	47	Cloudsoft	28%	Cloudera, University of Málaga (tie)	4%
Pachyderm	10	Pachyderm	30%	Barnes & Noble, Dakis, DataStax, Globo.com, Townsquare Media (tie)	10%
Aurora	52	Twitter	37%	TellApart	4%
Tsuru	53	Globo.com	38%	CWI Software, Government Digital Service (tie)	4%

Contributors and Associated Companies					
Project Name and # of Contributors		Leading Employer and % of Contributors		Second Leading Employer and % of Contributors	
Weave	22	Weaveworks	41%	Red Hat, VMware (tie)	9%
Powerstrip	7	ClusterHQ	43%	Red Hat	14%
Rancher	16	Rancher Labs	44%	BBC, Docker, Proximis, Red Hat, Spirula Systems (tie)	6%
Crate	22	Crate.io	45%	Lovely Systems	9%
Cloud Foundry Release	179	Pivotal	50%	IBM	8%
Flocker	20	ClusterHQ	55%	Huawei, Rackspace, Red Hat, ShoreTel (tie)	5%
appc Spec	56	Red Hat	57%	Apcera, Google, Huawei (tie)	5%
ServiceD	35	Zenoss	57%	StackEngine	6%
Docker Engine for SmartDataCenter	23	Joyent	65%	Human API	9%
Photon	23	VMware	74%	EMC	4%
Project Calico Docker	16	Metaswitch Networks	81%	Red Hat	6%

FIG 5: *With Project Calico and Photon coming out of internal development, it is not surprising that most of their developers come from Metaswitch and VMware, respectively.*

Companies like ClusterHQ, Crate.io and Rancher have products that are primarily based on open source projects, and each company accounts for most of their own contributions, as well as contributors.

The Crate project seems to be isolated from other projects. None of the developers working on it have contributed to any other project we evaluated. Note that Crate.io is based in Berlin, and Lovely Systems, the other provider with employees working on the project, is based in Austria.

Although RancherOS is doing most of the development on their project, with 7 of 12 contributors working for Rancher, several of those developers have made contributions to other projects, like Docker Machine and Flynn.

Docker: The Reason Behind the “Whale Factor” Nomenclature

Containers have been around for a long time, but it wasn't until the success of Docker that the trend really took off. While 58 percent of contributions to Docker come from Docker, Inc, the corporation employs only nine percent of the actual people that have contributed to the project. Similarly, the Open Container Initiative's runC, which itself is based on Docker's libcontainer, has a broad base of contributors, with Docker and Red Hat employees accounting for a combined 23 percent. There are 46 Docker employees that have contributed to 10 of the 31 projects covered in Figure 2. However, outside of Docker and runC, the other projects each only saw one employee participating. Admittedly, many Docker repositories are not included in this analysis.

CoreOS: A Driving Force in the Cloud Native Computing Foundation

At its foundation, Google donated the Kubernetes project to the Cloud Native Computing Foundation (CNCF) with the hope that companies and individuals would build on the effort. In December, CoreOS went further by donating the etcd and flannel projects to CNCF. By looking at the contributors of these projects, we can make some judgement about how independent CNCF is. Although Google and CoreOS engineers account for the lion's share of contributions for the aforementioned projects, these employees account for no more than half of the actual contributors.

As long as this broad base of support continues, commercial products like CoreOS Tectonic and Kismatic can be successful, and other products from Mesosphere and Red Hat's OpenShift will make sure they are compatible with Kubernetes.

Our sample of 31 projects is over-represented with four CoreOS projects: appc Spec, etcd, Fleet and rkt. It is also remarkable that we identified 13 projects with 28 CoreOS employees contributing to them. If LinkedIn is correct and CoreOS only has 43 employees, then 65 percent of its workforce is involved with these projects. CoreOS's CTO Brandon Philips appears to be very involved, contributing to 12 of the 31 projects.

Google: Can You Say Kubernetes?

With Kubernetes' development coming from within Google, it is not surprising that 76 of the 90 contributors have been Google based. Interestingly, five employees have a history of contributing to the Packer project associated with HashiCorp.

Red Hat: The Godfather of Open Source Provides the Manpower

164 Red Hat employees worked on 16 of the 31 of the projects we looked at. It was the leading source of contributors to Vagrant, etcd and the appc specification. It was the second leading contributor to eight other projects.

Red Hat's base of engineers are not just tangentially involved in many of these projects: in two important container-related projects, runC and Kubernetes, Red Hat employees accounted for 20 percent and 15 percent of contributions, respectively. The raw manpower being contributed can be significant. Seventy Red Hat developers have worked on Kubernetes and 47 have worked on etcd.

Mesosphere: A Favorite of Web-Scale Companies

Although Mesosphere originated Apache's Mesos and Marathon projects, only 20 percent of those projects' contributors are actually employees. Of the seven projects it is has participants in, it is making significant contributions to four of them. Its core software, DCOS, is the basis of orchestration tools, so it is no surprise that its employees have also been making the most contributions to Chronos.

Furthermore, developers at Twitter and Facebook are tied to Mesosphere, as evidenced by them being the second leading source of contributions to Mesos and Marathon, respectively.

Cisco: Mantl Not Being Dominated By Its Sponsor

Cisco's Mantl infrastructure has broad-based participation. The leading source of contributions on this project is Asteris, which is actually a small collective of developers that are doing work for Cisco. Even considering this, Cisco and Asteris account for only 22 percent of contributors.

The breadth of companies contributing to the project is proof that their strategy of actively engaging the open source community is working. As Mantl continues to mature, there is a possibility that this free, curated list of microservices will generate interest in Cisco's paid Project Shipped service.

It is also noteworthy that with five developers contributing, Cisco is heavily involved with OpenStack Magnum, an API service that provides availability to orchestration engines from Docker and Kubernetes.

Areas for Future Research

We want to apply this research methodology to a broader universe of projects. Those projects may focus on a related topic area like microservices or focus on a specific programming language.

Another way to expand the scope of the research would be to drill-down into all the subprojects within a repository, or look at the GitHub organization as the unit of analysis.

Measuring involvement with a project cannot, by itself, demonstrate a person or company's influence on a project. However, these factors might be related to the speed in which decisions are made, which can be ascertained by the time it takes to resolve an issue or accept a pull request.

It would also be interesting to investigate the degree to which company affiliation influences technology decisions. Although each project has its own unique technology matrix, it is possible to count the number of plugins and subprojects that have been written to provide compatibility with different programming languages and platforms. Demonstrating a causal relationship between company involvement and decision-making would help inform a developer's decision when choosing a project to get involved with.

RETHINKING THE DEVELOPMENT AND DELIVERY ENVIRONMENTS



In this podcast, Scott Johnston talks to Alex Williams of The New Stack about the state of software delivery prior to Docker, and how large code blocks took weeks and months to ship. But it wasn't, and still isn't, just about how fast your team delivers: it's also about the programmable infrastructure that's available. Following the success of configuration management tools and the need for consistent servers, Docker helped teams rethink the development environment, and eased friction in the delivery pipeline. Scott also talks about containers as a reasonable starting point for microservices and distributed computing, and how tools like Compose and Universal Control Plane are looking to elevate the usage of microservices.

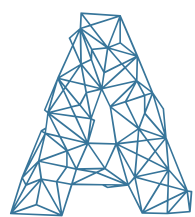
[Listen on SoundCloud](#) or [Listen on YouTube](#)



Scott Johnston is the senior vice president of Product Management at Docker, responsible for strategy of products and technologies. With over 25 years' experience, Scott previously was vice president of marketing, product management, and strategy at Puppet Labs. Before that, he served in leadership and operational roles in product management, engineering, business development, and marketing for disruptive companies, including Netscape, Loudcloud, Cisco and Sun Microsystems.

CONTAINERS IN PRODUCTION, PART II: WORKFLOWS

by **MARK BOYD**



A variety of companies are already using containers at scale in production. In the previous chapter, we looked at how and why Spotify, DramaFever, Built.io, and the Institute of Investigation, Innovation and Postgraduate Studies in Education (IIIEPE) use containers. Now let's dive in and take a closer look at each organization's workflows.

Building the Application and Managing Pull Requests

One of the appeals of using containers in production is the capacity to create a seamless development-to-production environment that may originate on a developer's laptop, and can be moved wholesale to testing and deployment without errors being introduced due to changes in the infrastructure environment.

In most production use cases we surveyed, code is stored initially in private repositories, with devs making pull and merge requests from there.

Then, images are created and these builds are sent to continuous integration tooling to continue with testing and merging the new code.

What IIEPE Uses

Luis Elizondo, lead developer at the Institute of Investigation, Innovation and Postgraduate Studies in Education, says that, before Docker, moving apps from development to production was one of their biggest issues. Now developers build base images and publicly release them to Docker Hub. All applications have a standard structure, including application, log and files subdirectories. These are the subdirectories that the developers will mostly use, while a Dockerfile, YAML file for Docker Compose, and Makefile hide complexities about the application container environment that developers don't necessarily need to know about.

What DramaFever Uses

Video streaming site DramaFever uses Python and the Django framework for their main site and Go for microservices handling functions like image resizing, player bookmarking, and DRM. In either case, the versioned deployable artifacts are stored and deployed as container images.

For Bridget Kromhout, former operations engineer at DramaFever, using containers helped ensure a standard development environment for a team of developers located around the world. She said that containers “help with the distributed nature of my team. It helps us to have fewer problems, as it reduces the number of inconsistencies that slow down our velocity.”

What Spotify Uses

At Spotify, developers are encouraged to use Java, ZooKeeper and [VirtualBox](#) on their local machines during builds. Engineers use GitHub to

store and share their code and pull requests, and use [Bintray](#) to store container images.

Continuous Integration Workflow Configuration

Once developers have written code or merged pull requests, a new base image needs to be built and then tested through various stages of QA. For many of the application environments using containers in production, this is done using continuous integration tools, notably Jenkins, to automate as much of this process as possible.

What IIEPE Uses

Elizondo is adamant that developers should not be developing outside the container environment and then building the image for deployment. “Let Jenkins do it for you,” he says.

As explained in the previous chapter, IIEPE has two servers: one running MaestroNG, which handles stock Docker containers, and another running Jenkins to automate repetitive tasks.

“Jenkins also allows you to have a history of fails to replace your Cron submission, and I programmed Jenkins so it is running Cron on each of the applications so you have a log and a history of everything that goes wrong,” Elizondo says.

What DramaFever Uses

Like Elizondo, Kromhout used Jenkins for all container image builds. “We have weekly base builds created, as Python-Django builds can take forever,” she said. So instead, if they have a weekly base build already created, the time for the continuous integration builds for the

development branch is reduced; the Dockerfile for the main www project uses “FROM www-base”. When working locally, devs pull from the master in GitHub and mount that code into the container they’ve docker pulled to have the latest build environment. Kromhout says this is faster than using Vagrant. “Jenkins just builds the one image for all of the platform, it just builds and tags one image,” she says. “It is really nice for consistency.”

What Spotify Uses

Goldin says Spotify takes images built in Bintray and then deploys them to AWS. Helios can then be used for service discovery, but is also pluggable with more specific, full-featured service discovery tools including SkyDNS and Consul.

Orchestration and Management

Orchestration and management challenges increase as companies find more uses for containers. Technically there are three processes here: first, how containers are connected together is typically referred to as orchestration. Then there is the need for a scheduling and placement engine that can decide which hosts those containers should run on. And finally, container management involves noticing if containers go down, declaring the state of services when they are deployed and similar functions. However, there is often much blurring between the terms, and orchestration is often used as a catch-all for those three functions.

What IIEPE Uses

Many of Elizondo’s potential orchestration choices were curtailed by his institution’s need to choose an open source solution that could also be used with Drupal, the framework in which many of IIEPE’s web applications were built. Shortcomings in the feature set also limited

Elizondo's choices, although he stitched together features from a number of options. For example, while Shipyard lacked the capacity to manage containers automatically, it is useful as a viewer and is used by IIIEPE to monitor the status of their container and Docker services, to identify when a container crashes, and to reboot dead containers.

MaestroNG was chosen as their core orchestration tool as it has multi-host capabilities, a command line interface, and uses a YAML file to describe everything. For security precautions, MaestroNG was installed on a server and is the only service that connects to Docker. Once Jenkins has finished testing new image builds, it is pushed to IIIEPE's private registry where Jenkins then connects to the MaestroNG server using SSH security protocols, and completes the deployment to production.

What DramaFever Uses

DramaFever's orchestration is currently very simple. Because DramaFever is 100% in AWS, they use DNS and ELBs, with instances in a specific autoscaling group launching, doing the docker pull to run the necessary containers, and answering in a specific ELB.

New images are built by Jenkins and after automated and manual testing, are tagged for staging and then prod via fabric, a Python-based tool. So the same Docker image that passed QA is what is pushed out into production (from the distributed private Docker registry).

What Spotify Uses

Orchestration is at the core of why Spotify built and uses Helios. Goldin says that Helios solves one problem, and solves it really well: essentially deciding where to spin up container instances across a cloud-clustered network. While Helios is open source, Spotify also has an internal helper

tool. In order to aid with continuous delivery, Spotify is currently extending the feature set of Helios to be able to show visualizations of what was deployed, by whom and where.

Other Alternatives

[Built.io](#) also created their own orchestration/management layer for their mobile-backend-as-a-service (MBaaS) container architecture. This management layer uses an API as the main communication channel to manage setting up new containers, and can also review customer profiles and create larger containers for higher-paying customers.

By using a management layer connected via an API, they can also enable their customers access to some orchestration and management functions direct from the built.io product's user interface.

Service Discovery and the Load Balancer

The maintenance of applications in production needs service discovery — that is, insight into the current state of containers running across an application's infrastructure. A load balancer then ensures that application traffic is distributed across a number of servers in order to ensure high performance of the application in production.

What IIIEPE Uses

Elizondo says IIIEPE uses Consul for service discovery, as etcd still requires a lot of additional learning. Consul stores the IP, port and state of an application. When Consul recognizes changes to the application, Elizondo has established a trigger to create a [Consul-template](#), which in turn can reconfigure the load balancer configuration. Elizondo has been using NGINX as the load balancer, but plans to switch to the more complex

HAProxy. “NGINX is great,” he says. “I know how to solve about 80 percent of the problems that NGINX throws at you. But NGINX is a web server. It can act as a load balancer, but is not what it does best.”

What DramaFever Uses

Kromhout used a Chef client run via Packer in a Jenkins job to generate Amazon Machine Images for host instances populated with the necessary upstart template so that the right Docker images would be pulled and run, for instances launched from a specific autoscaling group as members of a specific Elastic Load Balancer (ELB). Using NGINX for proxy_pass to different upstream ELBs per microservice, Kromhout said, allowed DramaFever to defer the more complex problem of dynamic service discovery.

“We used HAProxy for some database connection pooling, but ELBs + NGINX simplify a lot in the traffic-routing space. Location blocks work well when you can predict where the containers are launching,” she explained. “Of course, that meant our utilization wasn’t optimal. Today, Peter Shannon, the current head of ops at DramaFever, is looking into more dynamic service discovery options. Application container clustering is how to realize highly efficient utilization.”

Tying It All Together

It’s too early to codify best practices, but these are a few examples of emerging methodologies which show what a container production workflow could look like. Some of the tools mentioned in this article are not container-specific technologies; rather, containers make it almost trivial to deploy code.

In all cases, the final chosen workflow combines containers with some emerging ecosystem open source tools. This approach exists alongside some private workarounds and self-created solutions that address the individual enterprise's application programming languages, data server infrastructure, security policies and continuous delivery commitments.

THE EVOLUTION OF ARCHITECTURES AT CONTAINER-SCALE



In this podcast, Jason McGee talks about how the monolithic approach to architecture came out of the service-oriented architecture (SOA) movement. Now we're trying to decompose the monolith to uncover the complexities of this evolving pattern.

[Listen on SoundCloud](#) or [Listen on YouTube](#)



Jason McGee, IBM Fellow, is vice president and chief technical officer, Cloud Foundation Services. He is currently responsible for technical strategy and architecture across all of IBM Cloud, with specific focus on core foundational cloud services, including containers, microservices, continuous delivery and operational visibility services.



In this podcast, Andrew Hatelly discusses the shift to scale and what that actually means for infrastructure technologies, as well as how containers facilitate a move beyond web scale.

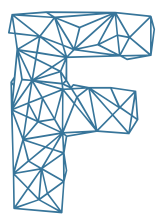
[Listen on SoundCloud](#) or [Listen on YouTube](#)



Andrew Hatelly is an IBM distinguished engineer and chief technical officer of IBM Cloud Architecture. He's currently working to define IBM's Open Cloud architecture strategy using Docker, OpenStack and Cloud Foundry. Andrew also leads a team of open source developers, encouraging them to build a better cloud for everyone.

LEVERAGING CONTAINERS TO PROVIDE FEEDBACK LOOPS FOR DEVELOPERS

by **VIVEK JUNEJA**



Fast feedback accelerates decision making and provides more time for making observations and identifying new directions. Observe, orient, decide and act ([OODA](#)), pioneered by John Boyd, is increasingly used to model feedback loops in software development processes. Successful companies, like Netflix, have adopted this decision-making framework and [turned it into a commodity](#).

“Feedback loops are an essential ingredient of Agile and DevOps mindsets.

Feedback loops not only provide immediate results on activities in the development process, they also instill good habits among developers. Scientifically speaking, it's been proven feedback loops [change human behavior and actions](#).

Stages of the Feedback Loop

Building a feedback loop for developers requires attention to four distinct stages. It starts with collecting observations, including the feature usage metrics, code quality, static analysis, performance metrics and testing results, among others. We can term it as the first stage of the feedback loop.

This information is then fed into an engine that provides valuable inferences based on the observations. These inferences avoid the clutter of raw data and produce practical metrics, including test coverage, cyclomatic complexity, user preferences and more. This can be termed as the second stage of the feedback loop.

The data generated from the first stage, and the inferences produced through the second, are used to create directions. This is the third stage, wherein each direction is a possible path of calibration and diagnosis. The directions represent multiple choices for decisions that could be taken based on the result of the first and second stage.

Finally, an action is taken in the fourth stage, depending upon which decision is selected. The actions taken are then fed back into the loop. This circular process is very effective in creating a flow in the software development process, and in the long run, reduces the [cost of changes and emergence of issues](#).

“ The most important aspect of the feedback loop is the time it takes for each step to complete.

The goal is to automate as much as possible in each stage of the feedback loop. A fast feedback loop is essential to move and produce reliable software for end users. This reliability breeds from the actions that are continuously taken based on current observations and decisions thereafter.

This article, however, is not an introduction to agile engineering and continuous delivery, both being pillars of building a fast feedback loop. We will, instead, be addressing the roadblocks that one faces while creating and managing the stages of feedback loop.

Integration and Diverse Deployments

One of the major roadblocks to building faster feedback loops is the pain of integrating and deploying large codebase. A large codebase increases the cost of compilation, bundling and deploying on an infrastructure. The cost of installing the required dependencies for a software environment to run the code can be prohibitive and slow the process. Also, with regards to diverse deployment environments running on heterogeneous infrastructure, supporting this diversity can become another issue that leads to further latencies. All of this adds up to longer decision intervals and reaction times, both of which are crucial to delivering production-quality software to users. .

The Cost of Change

At the heart of a fast feedback loop is reducing the cost of change. Improving feedback time from minutes to seconds, although seemingly trivial, can have a large impact on reducing this cost of change. Developers can confidently build new infrastructure and code with the promise that

no change is too big. Facebook’s mantra of “done is better than perfect” fits into this ideology of being able to build software at a rapid pace while minimizing costs.

As communities and organizations start experimenting with [microservices](#), they also need a new way to bring the “minutes to seconds” change to their feedback loop. That’s where [containers](#) come in. The container ecosystem is promising this change, and ultimately allowing developers to see their code in action by accelerating all four stages of the feedback loop.

Hyper-Agile Feedback Loops and Developer Experience

A hyper-agile feedback loop is one where the transition time from one stage to the other is minimal and measured in seconds.

Creating a hyper-agile feedback loop is nothing short of an extraordinary developer experience. Such an experience encourages risk taking, alternative solutions and a cycle of [Kaizen](#). The easiest way to understand how the container ecosystem produces this hyper-agile feedback loop is to be aware of the ecosystem participants and their contribution to the loop.

We can do this by mapping the container ecosystem to the necessary aspects of the hyper-agile feedback loop. This feedback loop will typically involve the following:

- 1. Quick and Consistent Development Environment Provisioning:**

The developer on a team obtains the local test environment and source code for their development environment. The test environment

is itself a declarative manifest that is used to provision the environment for the developer. The provisioned environment is a containerized set of service instances each running independently. The local test environment contains all the necessary services and related dependencies for the product.

2. **Loosely Coupled Development:** The developer changes or adds a new service, along with the required test cases. The local test cases are then run to test the changes. To avoid breaking clients, developers could also insist on having [consumer-driven contracts](#) to validate the change before publishing.
3. **Automated Deployment and Tests:** The developer then deploys this change to the local test environment and triggers the integration tests for the participating services that use this particular service.
4. **Version-Controlled Development Artifacts:** Once the changes are tested with the local test environment, the developer then commits the changes to the repository. If the service also requires a configuration change or a new dependency, the developer adds that to the environment manifest and commits the same.
5. **Automated Build and Production Release:** Committing the code leads to scheduling a build task on the build and deployment service. This service only deploys the relevant changed service as a new service version, instead of replacing the old version, which follows the principles of [canary releasing](#). The appropriate load balancers are informed of the new service version and a progressive load shifting takes place to allow a certain amount of requests to the new changes in the service.

6. **Migration and Deprovisioning:** The new change is then observed and metrics are collected based on A/B testing and user analytics, vis-a-vis the older version of the same. If the results are promising, more load is shifted and eventually the new service takes prominence over the old one. The old version is gracefully retired. If the change is not promising, then the new service is gracefully deprovisioned, and the load is moved off of it.

The above steps merely scratch the surface of what is possible when creating a feedback loop for developers. The container ecosystem provides a rich set of services covering a wide range of capabilities for developers, and accelerates each of the above steps.

Platforms and Version Control

One of the most interesting aspects of the container ecosystem is the rise of niche PaaS options, like [Tsuru](#), [Giant Swarm](#) and [Deis](#), among others. Some of these PaaS options are transparent solutions, while others provide a layer of abstraction in the form of developer-friendly DSLs to architect and deploy complex applications. With rapid change cycles and an iterative development process, PaaS solutions like these come in handy when considering quick developer feedback. Existing PaaS implementations, like [IBM Bluemix](#), [Pivotal Cloud Foundry](#), [Apcera Platform](#) and [OpenShift](#), have already geared up for integration with the Docker and container toolchain.

Most organizations adopting containers for purposes of speeding up their development process have the inherent need to manage multiple versions of their services at the same time. This is primarily needed to support canary releasing and service immutability. The latter is an

interesting proposition, in that it is considered a change to deploy a new service instead of updating an existing one. This reduces rollback time and provides an opportunity for A/B testing.

The Need for Standardization

To enable fast feedback for changes, it is essential to provide developers with a template that has all the underpinnings of the various container ecosystem toolchains, including monitoring, governance and PaaS integration and solutions.

“At the heart of all these concerns is the idea of standardization.

A project like [ZeroToDocker](#) is an initial step towards creating a boilerplate for developers — one that is already preconfigured to provide microservices-based development practices. This allows developers to invest their time developing and releasing reliable code instead of managing the complexities of a service-oriented system.

The idea is to have a standardized “hello world” boilerplate project for different language types. It would consolidate best practices of service-oriented development, including preconfigured logging, monitoring, service registration and discovery, and integration with container ecosystem toolsets. This forms the service unit of a microservices system, which can be quickly developed and released without reinventing the wheel.

Reaching Maturity

Despite its roots and vibrant history, the current container ecosystem is

still in its infancy, and changing nearly every day. The criteria for creating a mature platform for fast feedback loops will also, surely, change in the course of container ecosystem maturation. However, it does already provide the building blocks for creating a hyper-agile feedback loop, which signals how software will be built and released in the coming years.

A HISTORICAL PERSPECTIVE AND THE FUTURE OF CONTAINERS



In this podcast, Bryan Cantrill starts with a broad history of how technologies, such as AJAX, JSON, COBRA and SOA, contributed to the rise of the monolith. And from this historical perspective, he talks about how the idea of microservices is significantly older than the word itself. A very similar situation happened with containers and technologies like jails, FreeBSD and Solaris Zones. But whereas containers in the past were used to support very large applications, containers have found their fit now with small applications (microservices). Bryan goes on to talk about the history of legacy providers, the “Uber-ization” of software-driven enterprises, and creating optimal development environments for microservices.

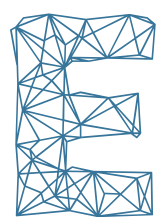
[Listen on SoundCloud](#) or [Listen on YouTube](#)



Bryan Cantrill is the chief technical officer at Joyent, where he oversees worldwide development of the Triton Elastic Container Service, as well as SmartOS, SmartDataCenter and Node.js platforms. Prior to joining Joyent, Bryan served as a distinguished engineer at Sun Microsystems, where he spent over a decade working on system software. In particular, he codesigned and implemented DTrace, a framework for dynamic instrumentation of production systems. Bryan also cofounded the Fishworks group at Sun, where he designed and implemented the DTrace-based analytics facility for the Sun Storage 7000 series of appliances.

HOW CONTAINERS AND MICROSERVICES WORK TOGETHER TO ENABLE AGILITY

by **ALEX WILLIAMS**



Enterprise chief information officers (CIOs) used to think about their IT assets primarily in terms of hardware, with software serving as the system of record. Today, the emergence of an application-centric approach is causing many to rethink that view, placing the application first, not the machine. Open source projects — from new technology companies to established enterprise providers — are driving this innovation. These projects serve as the core to a new stack economy, and they are the underlying foundation for automated, programmable infrastructure.

When IT systems were designed around the client/server paradigm, software was required to run on individual servers. With this approach, server sprawl proliferated. Virtualization brought relief, allowing servers to be used more efficiently.

As virtualization brought about greater efficiencies and reliabilities, CIOs started to look at how application development could go faster, especially as the Internet became more widely available.

Virtualization was not meant to speed application development, nor was it intended as a way to build real-time services. It was an era of complex systems hardware, with application development processes that were built on waterfall methodologies. Application development took months, and updates to the application required tedious planning and execution processes.

At first, many look at service-oriented architectures (SOA), which relies on the premise that the application has components communicate over a network. Today, SOA can be considered a precursor to microservices.

IBM's [Jason McGee](#) said at DockerCon EU 2015 that SOA missed something, though. It focused on the interface and defined how to talk to the service, but it did not define how teams should be organized. Further, it missed how to encompass the delivery lifecycle.

Today, with microservices, we see the decomposition happening and the teams getting organized accordingly. Small teams are working on many little parts. Now there are teams of 5, 10 or 15 people working on parts of the service. For the most part, the old approach was defined according to the way monolithic systems were built, McGee said. The interface and lifecycle were not separated. It kept the approaches of the time. It was built on Java, with a particular stack on a versioned environment.

Service-oriented strategies are now looked at more holistically. REST and JSON allow ways for services to work together, independent of how the team works, McGee said. The model allows the developer to use the language that makes sense for the task or for the people on the team. There still exists a dominance of a software that a team uses, but there is more of a polyglot approach.

Microservices are reflecting a trade-off, McGee said. That tradeoff is embodied in the complexities that come with microservices. A tool and management system is needed around the microservices and containers that are getting packaged.

Fundamentally, developers have to figure out how the services find each other, McGee said. And then there is the monitoring and visibility. How do you recognize what is happening? It becomes a mapping project. There are multiple failure points. It gives rise to immutable infrastructure and speaks to looking at the overall ecosystem more holistically.

Frameworks and Fundamentals in Microservices

Companies that had to build scalable systems to meet the unpredictable and sometimes heavy demands of Internet traffic were the first to think about this problem, and they turned to frameworks. A de facto standardization for how services are delivered had evolved from these systems, McGee observed.

But how does the developer know all the approaches? As it is now, existing frameworks define the processes, which forces the developer to start again if there is a decision to change the framework. It becomes an evolutionary problem.

Standardization is key, and groups, such as the Cloud Native Computing Foundation (CNCF), aim to pave the way for faster code reuse, improved machine efficiency, reduced costs and an increase in the overall agility and maintainability of applications.

CNCF has a host of member companies on board with this idea, including

Apcera, Apprenda, Cisco, Cloudsoft, ClusterHQ, CoreOS, Datawise.io, eBay, Engine Yard, Docker, Google, IBM, Intel, Joyent, Mesosphere, NetApp, Portworx, Rancher, Red Hat, Twitter, VMware, Weaveworks and more.

CNCF is looking at the orchestration level, followed by the integration of hosts and services by defining APIs and standards through a code-first approach. At the CoreOS Tectonic Summit in New York, Google product manager Craig McLuckie [talked about offering Kubernetes to the CNCF project](#) and Google's hope to make software a de facto standard for managing containers.

From Google's point of view, container-based architectures will have to be the standard for companies needing to build scaled-out systems and services. The microservices approach served as the foundation for what we are starting to call "cloud-native computing." By breaking the application into containers, small teams can specialize and become accomplished in simple, clear ways.

The nuance of this approach is reflected in the container itself, which will vary in scope, depending on the resources that are required. There may be different resources demands, such as more processing or I/O. Packaging them separately allows for more efficient use of resources. It also makes it easier to upgrade components without taking down the application as a whole.

As businesses increasingly tie their products, services and devices to the Internet, more data will be generated and, with it, new ways to use it. That, in turn, will mean new thinking about how applications are developed.

Frameworks will be critical to that evolution, as will the different aspects of what those frameworks do. That means the need, for example, to

develop schedulers based upon business requirements. It will also mean a way for more nuances, or semantics, to build directly into the architecture itself.

[Mantl](#) is an open source framework Cisco has developed with the Mesos ecosystem that offers integrations from services, such as Consul and Vault by Hashiworks. It's representative of the frameworks that are emerging from Kubernetes, Mesos and the numerous open source projects that support the microservices ecosystem.

Mantl uses tools that are industry-standard in the DevOps community, including [Marathon](#), Mesos, Kubernetes, Docker and [Vault](#). Each layer of Mantl's stack allows for a unified, cohesive pipeline between support, managing Mesos or Kubernetes clusters during a peak workload, or starting new VMs with [Terraform](#). Whether you are scaling up by adding new VMs in preparation for launch, or deploying multiple nodes on a cluster, Mantl allows the developer to work with every piece of a DevOps stack in a central location, without backtracking to debug or recompile code to ensure that the microservices the developer needs will function when needed.

The obstacles on the journey to simplify IT systems through automation are familiar, Cisco's [Ken Owens](#) noted. How do you connect disparate systems? How do you deal with information security policies around data security and governance? This is where containers enter the picture. Containers have allowed developers and infrastructure teams to work together on internal, test and developer environments.

There is a major misconception that monoliths can't be broken into containers or be cloud native. From an application architecture point of

view, there are monoliths and there will be monoliths for some time.

“You can’t take a monolithic ticketing system and containerize it today,” Owens said. “But there are pieces of the app, such as DNS services, security services, networking services — aspects around it that can be containerized.”

The scheduler is the most important part of the system when considering how microservices are used. From Owens’ perspective, the most standard schedulers are focused around Mesos with Zookeeper and Marathon. Kubernetes is applicable for use in data science, offering higher-speed schedulers.

There are two aspects to schedulers: the functioning of scheduling the jobs, and the efficiencies and availability of requests. Cisco has a minimum of three control nodes per set of six main service nodes. This provides a 2-to-1 ratio — allowing for failure and continued high availability.

This speaks to the conversation we will hear in the year ahead about this emerging microservices layer and how you orchestrate services, the schedulers, the control plane underneath, and how they all connect together. It is, in essence, a new stack of orchestration that puts an emphasis on matters, such as application portability and self-organized containers.

Addressing Security and Containers in the Enterprise

Creating their own container allows customers to write around one standard container. Customers get the tools they need for multi-tenant environments, allowing them to control complex topologies across public cloud services and on-premises environments.

Apcera is a platform for enterprises to deploy containers with security and policy enhancements included. It is all automated in the packaging phase, and there are granular controls over components, such as network policy and resource allocations, so that the customer has a general understanding of their cluster's security footprint noted Apcera's [Josh Ellithorpe](#).

In addition, Apcera has developed a semantic pipeline that offers capabilities to guarantee data integrity. For example, Apcera technology acts as a proxy to the database. Drop requests do not go to the database directly. Apcera also offers what it calls “policy grammar,” which focuses on resource allocation, network controls, scheduling policies and authorizations. This gets to the heart of how semantics define how a data pipeline is managed. The policy grammar helps describe the fundamental business rules and how the application is deployed for the organization.

A Docker image has multiple layers. Apcera detects the framework and calls into a package manager that has a tag system to describe dependencies. When software is audited in Apcera, the user can get an accurate look at what is there. Audit logs are captured for everything for complete understanding. The developer spends less time thinking about what dependencies are needed.

Their whole idea is to inject transparent security into the process so that it is invisible to the developer. The developer just connects to a database, but they do not see the policy enforcement in action. The industry is now realizing that the policy itself needs a common grammar. Apcera is now, in turn, making the grammar openly available.

Conclusion

Microservices reflect a trade-off that is embodied in the complexities that come with distributed services. There needs to be a tool and management system around the microservices and the containers that are getting packaged. The roots of today's microservices platforms and frameworks come out of the machines that ran the software for what we once knew as service-oriented architectures.

The emergence of microservices can be viewed as an evolution of the earliest frameworks, developed originally by companies that were the first to scale-out programmable and dynamic services that served millions of users. Today, the approach developed by these pioneers is the foundation for the mainstream company that understands that software and people are now the heart and soul of their entire organization.

WHAT DOES IT MEAN TO BE CLOUD NATIVE?



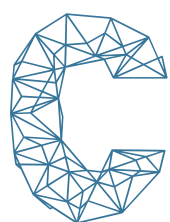
In this podcast, Michael Côté talks about using the “cloud native” moniker to describe three primary principles: utilizing microservices architectural patterns, implementing a DevOps approach to running production systems and development environments, and looking at continuous delivery as the primary vehicle for software delivery. Côté goes on to elaborate more on each of these cloud-native principles and what it means to address them. He also talks about how Pivotal Cloud Foundry has all the tools and services on their platform to meet and exceed the needs of cloud-native businesses. [Listen on SoundCloud](#) or [Listen on YouTube](#)



Michael Côté works at Pivotal in technical marketing. He’s been an industry analyst at 451 Research and RedMonk, worked in corporate strategy and M&A at Dell in both software and cloud, and was a programmer for a decade before all that. He blogs and podcasts at [Cote.io](#) and is [@cote](#) on Twitter.

THE ROLE OF PLATFORM-AS-A-SERVICE IN THE CONTAINER ERA

by **SUSAN HALL**



Containers and platform-as-a-service (PaaS) systems have long had a symbiotic relationship. Many PaaS offerings have long depended on Linux Containers. And, of course, Docker was created at dotCloud to help it run its own service, making the connection between PaaS and the emergent container ecosystem a direct one. Containers made PaaS possible, and PaaS put containers on the map, enabling much of today's most popular technology.

But now containers and PaaS are coming into conflict, says Derek Collison, founder and CEO of Apcera, and one of the original creators of Cloud Foundry at VMware before it was spun-out as part of Pivotal. As he explains it, the original idea behind PaaS is that you simply hand your app to the PaaS system, and it takes care of everything else you need for a production environment. But containers offer a different approach. By packaging an application and all of its requirements before it's ever time to reach production, it may remove the need for a traditional PaaS.

“[PaaS vendors have] got to figure out what to do about Docker,” he says. “And in a constantly shifting landscape, they have to figure out, ‘Where are we relevant now?’” PaaS vendors are fairly confident that not only do they fit in this landscape, they’re still helping to define it and make it work in production environments.

The Real Value of PaaS

If you accept that the primary purpose of a PaaS is to make it easier for developers to deploy applications into production, then the question of relevancy is easily assuaged. Pivotal Senior Director of Technology Andrew Clay Shafer says there’s far more to PaaS than some give it credit for — he says it’s also an important part of how applications are managed in production.

“If you’ve got one container, what are you going to do with it? You’ve got to figure out how to schedule it, how to monitor it, a bunch of other things. And as you build those things out or use components to do it, it starts to look suspiciously like a platform.”

— Andrew Clay Shafer, senior director of technology at Pivotal

Jonathan Baier, senior cloud architect at the New York City-based consultancy Cloud Technology Partners, agrees. “The value is around orchestration, around management,” he says. “Once you’ve got containers, you need something to help you manage them. Managing communication between them, managing the failures, managing the change, all that churn, there’s this operations side to it, and that’s where PaaS [vendors] are trying to add value.”

To that end, the Cloud Foundry team created Lattice, a tool for creating and managing clusters of containers. And that’s certainly the direction Red Hat is heading as well. The company has been working with Google’s open source container management system, Kubernetes, and others on OpenShift 3, which was released for general availability in June. The company is also including build automation and support for automating rolling deployment.

“In the Docker world, the container is an image,” Joe Fernandes, director of product management for OpenShift at Red Hat explains. “So even to update your application, to push new code to modify the configuration, you have to rebuild the container to make those changes. So we’re building an automated image build as part of OpenShift 3.”

Where’s All This Going?

BuildFax founder and CTO Joe Emison sees all of this as part of a continuous evolution towards giving developers more control over their workflows and reducing friction. It all started with virtual machines, which made it possible for developers to create new development environments without having to wait for the IT department to set up physical machines for them. Cloud computing made things even easier by removing the need

to own physical servers altogether. Things like PaaS, containers and backend-as-a-service (BaaS) offerings continued the evolution by reducing the amount of configuration developers have to do, and cutting down on IT's involvement with application deployment. But he thinks there's more work to be done on the management side.

Collison agrees, saying PaaS vendors should now focus on making it easier to deploy secure and safe workloads. He says that the biggest threat to established PaaS vendors will be infrastructure-as-a-service systems creeping up the stack as they provide new tools to handle tasks like provisioning, managing and monitoring applications.

All of this change is surely stressful to virtual machine or PaaS vendors, but for developers, and the companies that employ them, all of this is good news. As Emison puts it: "If software is eating the world, you want to take your software developers and weaponize them."

Meanwhile, even though Docker is only two years old, Collison is already thinking about what comes next. "The virtualization wave has been a 10-plus year journey," he says.

“When you get something new, the lifetime of the successor is always shorter. So containers’ lifetimes might be two to four years max, and we’re already two years in.”

— Derek Collison, founder and CEO, Apcera

Looking at the overall evolution of IT towards lighter and faster solutions for developers, Collison thinks we can make some general forecasts as to where we're heading. "It's things like micro-task virtualization and security constructs that are being driven out of chips themselves," he says.

Whatever the case, the one thing everyone agrees on is that things will keep changing. "With these cycles – we saw this with NoSQL a few years ago — you have an explosion of solutions," Shafer says. "People learn new things, then some of those consolidate, then collapse back into a new steady state. Right now I think we're in a state where we're going to see a proliferation of solutions."

INTEGRATING CONTAINERS ALLOWS BUSINESSES TO MOVE FASTER



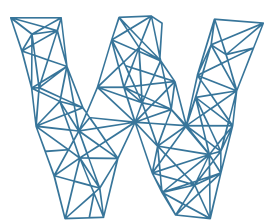
In this podcast, Kit Colbert discusses how IT operations influence how companies are adopting container technologies. Containers present an opportunity for businesses to move faster and present greater value to their customers, and it depends heavily on how you integrate the rest of your tooling with these technologies. Colbert talks about VMware as an infrastructure provider — and how their dedication to integration affects the way businesses deliver their products. [Listen on SoundCloud](#) or [Listen on YouTube](#)



Kit Colbert is vice president & general manager, [Cloud-Native Apps](#) at VMware, driving strategy and product development of Cloud-Native Application solutions within VMware. Previously, he was chief technical officer for End-User Computing, the chief architect and principal engineer for Workspace Portal, and the lead management architect for the vRealize Operations Suite. At the start of his career, he was the technical lead behind the creation, development and delivery of the vMotion and Storage vMotion features in vSphere.

ACHIEVING INNOVATION AND AGILITY WITH CLOUD-NATIVE APPLICATION ARCHITECTURES

by **SAM CHARRINGTON, FOUNDER AND PRINCIPAL ANALYST,
CLOUDPULSE STRATEGIES**



While most enterprises have deployed one or more workloads to public or private clouds, far fewer have embraced the technical, process and cultural change required to fully harness the power of cloud computing. This is unfortunate, because truly achieving the agility that enterprises seek requires more than forklift-migrating existing environments or applications to the cloud.

But, “agility” is a word that gets thrown around far too often in the context of IT. What does it really mean to achieve agility with cloud computing? Agility gives the enterprise the ability to out-innovate its competition. And, as technology both raises the bar for competition and lowers barriers to innovation, those enterprises that fail to out-innovate will themselves be out-innovated into oblivion.

Industry data suggests that an increasing number of enterprise IT organizations are retooling to better enable agility. According to research

conducted by Daryl Plummer, vice president and Gartner Fellow, 45 percent of CIOs state their organizations currently support a [second, faster, more agile mode of operation](#). Gartner expects 75 percent of IT organizations to support what they call “Bimodal IT” by 2017.

The ability to iterate quickly allows organizations to incorporate feedback loops into product and service development, ensuring that they are continuously and rapidly improved. By way of example, Facebook [pushes new code to production](#) twice per day, and Amazon has reported [making changes to production every 11.6 seconds](#).

Of course, these are not new ideas. That rapid iteration and feedback are necessary for continuous improvement is a notion familiar to most enterprises (on the business/manufacturing side) through concepts like Kaizen, Lean and Six Sigma. And many enterprises are adopting agile software development methodologies as a step towards applying these ideas to IT.

While there is no single technical magic bullet, momentum is building around a set of technologies and approaches that enable this kind of rapid iteration:

- Microservices
- Containers
- Container-based application platforms, like PaaS

Through embracing microservices, containers and platforms for continuous delivery, enterprises can fully tap into the transformative power of cloud-native applications and, in so doing, dramatically enhance innovation and competitiveness.

Key Enablers of Agile Cloud-Native Applications

One mistake we hear is thinking of these things as “new technologies.” Rather, microservices, containers and container-based application platforms are complementary new twists on established ideas that have, together, emerged as a key enabler of agility for modern, cloud-native applications (Figure 1).

To better understand these ideas and how they can be combined to enable a more innovative enterprise, we will explore each of them in turn.

Microservices

As is often the case with emerging architectural approaches, defining microservices is easier said than done.

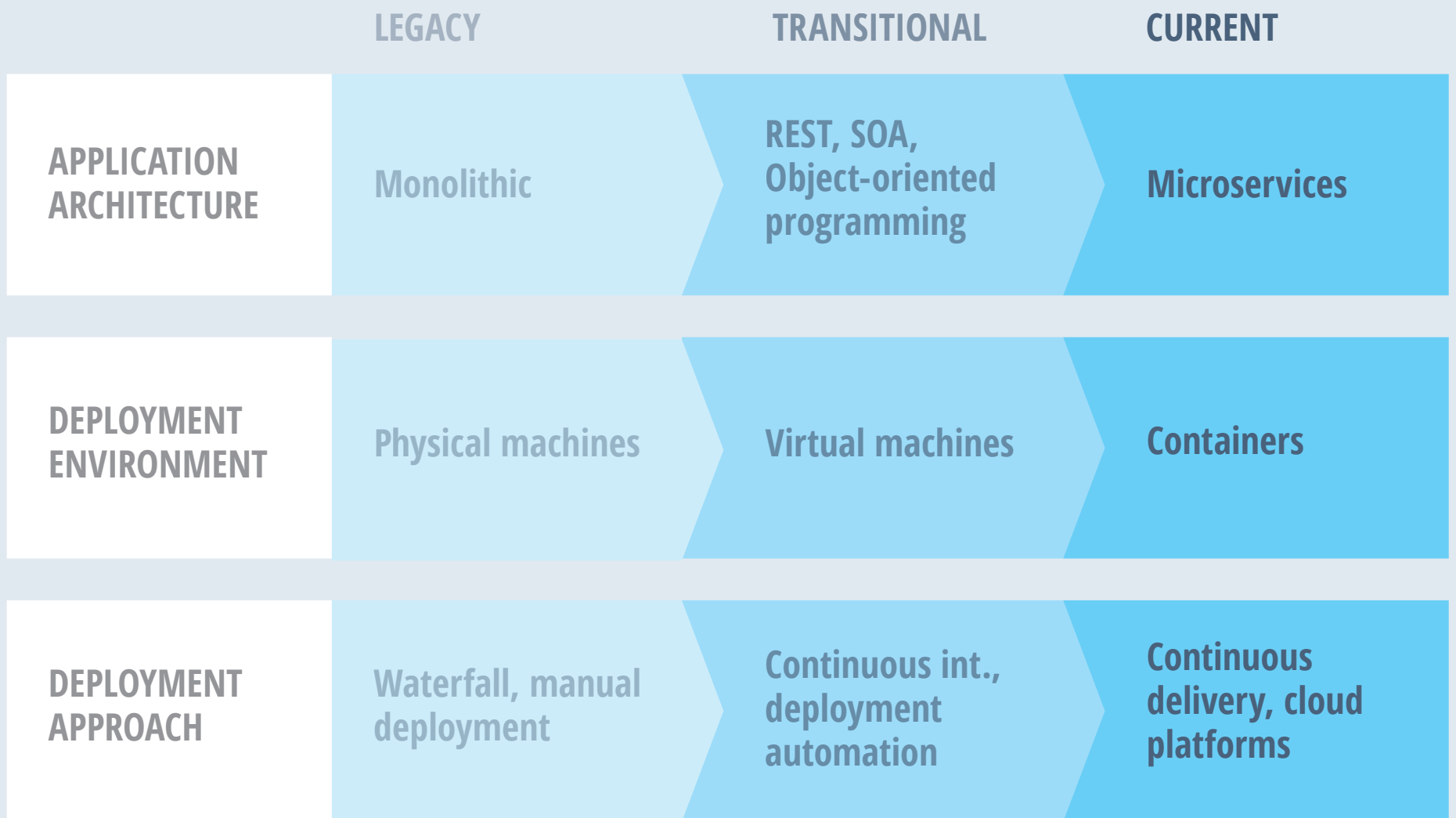
In their article on the topic, James Lewis and Martin Fowler [describe](#) the microservice architectural style as:

“... An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery.”

While this definition is necessarily high level, there’s actually quite a bit to unpack within it:

- **An approach:** As stated previously, microservices is an approach, not a technology. If you find yourself approached by someone trying to sell you “microservices” as a stand-alone tool, proceed carefully.

The Evolution of Application Architecture and Deployment



Source: CloudPulse Strategies, LLC.

THE NEW STACK

FIG 1: The previous names of strategies, models and patterns that are sometimes thought of as new infrastructure technologies.

- **Small services:** While it can be counter-productive to prescribe specific criteria for how small a microservice needs to be, size certainly matters since the size of your services will generally be a good proxy for how tightly focused they are.

Opinions vary widely as to how small is small enough. One blogger, James Hughes, [cites a consensus](#) between 10 and 100 lines of code. One company that's been using microservices for years used a different metric: you should be able to rewrite a microservice in about two days. However you define it, it's important to note that there is such a thing as too small. The "nanoservice" anti-pattern is used to describe scenarios where services are made so granular that

maintenance, network and other overhead outweighs the utility of the service.

- **Own process:** That microservices run in their own process is important so as to distinguish them from object-oriented programming (OOP), components, frameworks and other related ideas focused on the separation of concerns within an application.
- **Built around business capabilities:** Melvyn Conway [observed in 1967](#) that “organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations,” an adage that has come to be known as Conway’s Law. One unfortunate implication of Conway’s Law for many organizations is siloed and inefficient applications.

To turn Conway’s Law in their favor, some companies are organizing application development and delivery around microservices. At these companies, microservices are built and maintained by cross-functional teams that have full ownership of the service and all its dependencies, and in some cases are independently responsible for keeping the service up and running. The resulting teams are in a much better position to architect lean and efficient applications according to business concepts, not organizational charts.

- **Independently deployable:** As we’ll see in a later section, the “vertical integration” of individual microservices makes them deployable and maintainable independent of the other services that comprise the application. It also gives microservices teams a tremendous degree of flexibility when it comes to the technologies they use.

Why Microservices?

The microservices approach yields numerous important business and technical benefits compared to traditional monolithic architectures. While many of these benefits are possible to achieve with monolithic architectures, doing so tends to be difficult to accomplish consistently. This is due to the high level of discipline they require, without the helpful constraints provided by a microservices architecture.

Microservices architectures excel at helping organizations achieve:

- **Simplified application updates:** With traditional monolithic applications, even the slightest changes trigger deploys of the entire application or large pieces of it. As a result, changes to the application are batched into large releases or version updates, creating greater risk and increasing downtime for users.
- **Faster and easier deployment:** Small services are faster and easier to deploy than large, monolithic applications, and deployments are easier to validate, again reducing risk.
- **Easier scaling:** Independent microservices running in their own processes need not necessarily run on their own hardware but are easily shifted to do so, providing an easy approach to horizontal scaling. In the same manner, microservices allow application components with different scaling profiles (e.g., memory intensive vs CPU intensive) to be more easily deployed in their optimal environments than with monolithic applications.
- **Easier code maintenance:** Maintaining a collection of microservices each responsible for one thing is much easier than maintaining a monolithic application. This is in large part due to the fact that each of

the microservices can be more easily understood and has a well-defined interface, and the nature of microservices limits opportunities for side effects or hidden interactions to creep in. In addition, small services with limited scope are easier to debug than large monolithic applications.

- **Freedom to use the right tool for the job:** Because each microservice is developed, runs and is maintained independently, each microservices team has the flexibility to select the tools and technologies it feels will best enable it to meet the needs of the service. This means that in the same application, one microservice might use a JVM-based language and a NoSQL datastore while another uses Node.js and a relational database.
- **Reduced technical debt:** With individual microservices that are easily rewritten or refactored without impacting the larger application, microservices make it much more difficult for a team to accumulate a large technical debt.
- **Greater team ownership:** Self-contained services allow teams to take greater responsibility for their design and operation, promoting greater pride of ownership.
- **Rapid engineer onboarding:** Similarly, a new engineer joining an existing microservices team can gain an understanding of the service rapidly. Many companies using microservices and continuous deployment pride themselves on achieving day-one productivity for new engineers.
- **Greater resilience:** Because microservices are operated independently, greater emphasis is placed on monitoring their ongoing

operation than is typically the case with components in monolithic applications, leading to greater visibility and resilience in microservices-based architectures.

And, while building resilience into the consumers of microservices represents additional work and complexity for their users, the fact that this work cannot be avoided promotes a much higher degree of robustness in the system as a whole.

While the benefits of microservices architectures are many, the microservices approach is not without its drawbacks. These can be broadly categorized as either development or operational challenges.

Development Challenges

- **Communications issues:** One obvious challenge for microservices architectures lies in inter-service communications, which introduces a degree of latency that is not present in monolithic applications. Applications must be engineered to tolerate not just additional latency, but variable latency, retransmissions and service failure.
- **Development inefficiencies:** With many independent teams in operation, there is greater potential for reinventing the wheel. Organizations like Netflix have addressed this challenge by publishing internal tools as open source and promoting their use internally.
- **Architectural sophistication:** Architecturally speaking, microservices is a form of distributed computing and introduces the many challenges inherent to that domain; for example, traditional transactions can be difficult to implement in a microservices-based application.

Operational Challenges

- **Deployment:** Many small, independently deployed services increase dramatically the number of environments required to fully support the applications various versions and promotion levels. As a result, the burden on an organization to get such concepts as configuration management and deployment automation right is very high.
- **Monitoring:** Similarly, the many independent service endpoints that must be monitored on an ongoing basis can result in an increased system administration and maintenance burden for organizations using microservices.

In many ways, the development-oriented challenges are the most difficult to address, in that the only way to do so is the development of a set of competencies, processes and practices within the organization that supports the microservices style of development and architecture.

However, organizations considering or adopting microservices can take solace in the fact that the operational challenges of microservices, while also requiring the development of a new set of skills and approaches within the organization, have benefited significantly from the advent of a great many tools and technologies for easing the burden of managing large-scale microservices applications. Two of the most important of these are containers and container-based application platforms.

Containers

To illustrate the nature of the deployment challenges impacting microservices, consider a single legacy monolithic application that is being gradually modernized by migrating new and existing functionality to a more modern microservices-based implementation.

A single new feature might consist of 30 individual microservices, whereas previously, the single application required three distinct environments to be provisioned and managed (test, stage, prod). The landscape for the new feature alone consists of nearly 100 distinct environments.

At the same time, each of the microservices projects a given developer is on may use varying and independent sets of technologies. Developers need an easy way to manage these distinct development environments, hopefully without the performance overhead of virtual machines. These challenges call out for a deployment environment equally lean, focused and efficient as the microservices they will support. This context has fueled part of the dramatic resurgence of interest in containers within the technology community.

Whereas hypervisor-based virtualization abstracts the entire physical machine and provides a way to host independent virtual machine instances within a single piece of physical hardware, container technologies do much the same thing, one level higher in the stack.

Container technologies are based on a number of open source technologies, most predominately [Linux Containers](#) (LXC). LXC is an operating system-level virtualization tool providing an isolated environment for applications based on features built directly into the Linux kernel. Tools, like Docker, make LXC and the like easier to use by adding developer-friendly, command-line tools and/or APIs, flexible image formats and support for distributed image repositories.

Container Benefits

Containers offer a variety of benefits relative to virtual machines, particularly for microservice deployments:

- **Increased compute efficiency:** Because containers eliminate the need to instantiate multiple “guest” operating systems, they don’t have as much processing overhead as VMs. This means that your application can get more done with a given set of physical or virtual servers.
- **Increased storage efficiency:** Along the same lines, VM images are snapshots of the entire virtual machine, including the guest operating system. This means that they tend to be large, even for simple applications, and each incremental copy or version of the image results in more wasted space. The layered nature of popular container implementations (based on technologies like union file systems) means that container files can depend on other container files, (e.g., app depends on OS, app v2 depends on app v1) eliminating the duplication between each.
- **Faster application deployment and instantiation:** Because starting a container doesn’t require an OS boot, it typically takes milliseconds instead of seconds or minutes. This, plus the fact that containers are smaller and more easily transmitted across the wire, enables faster application deployment into production, and smoother rollback, failover and recovery. Pre-deployment, the tighter code/build/deploy/test loop they enable increases developer throughput and eliminates annoying thumb-twiddling and forced water cooler breaks for developers. In QA, automated tests can be run quickly and be made more thorough.
- **Improved isolation:** While containers don’t provide the full isolation of a virtual machine, they can simplify running multiple application stacks or microservices on a single physical or virtual machine.

- **Greater portability:** With containers, the application and its dependencies are bundled into a single package that can be run on any Linux machine and an increasing number of cloud platforms.
- **Simplified collaboration:** Because application containers are lightweight, they are easily shared. Container tools facilitate this further by supporting remote image repositories.

For these reasons and more, containers have gained incredible momentum as an alternative approach to packaging and deploying applications. Analysts at Forrester Research expect container technologies to [disrupt the virtualization market and drive cloud computing adoption](#).

While containers do address the overhead of running many small services within traditional environments, they don't address the operational challenges and automation needs of large-scale microservices architectures.

This is where CBAPs come in.

Container-Based Application Platforms

In order to iterate rapidly, we need to streamline the process of developing software, getting it into the hands of users (deployment), collecting feedback, and integrating this feedback into the planning for future versions of our products. Agile software methodologies focus on enabling the latter two and, as we've discussed, microservices and containers help with the first.

But, how do we reduce the overhead of deploying the software, particularly while we're increasing the number of moving parts with microservices and containers? And, even if we haven't yet adopted these

technologies, how do we make the delivery of software to users more efficient, such that we can do it in much less time, and with fewer resources, greater consistency and less risk?

One proven approach to achieving these objectives is to put in place a system for automating the delivery and execution of our software products. This is the role played by container-based application platforms such as platform-as-a-service systems.

CBAP Requirements

In the emerging microservices and container-centric world, organizations need an efficient linkage between the build systems employed by the development team and the infrastructure on which their software products are ultimately deployed.

Container-based application platforms provide this by offering features such as:

- **Developer-centric workflow:** CBAPs integrate with existing build and continuous integration tools to seamlessly deploy new code out to staging and production environments.
- **Automated deployment:** CBAPs interface with infrastructure management tools to provision servers, network and storage automatically and on-demand, and deploy containers and other software packages out to this infrastructure.
- **Automated scale up and down:** Applications running on the CBAP are managed according to policies set by the application owner. Desired scaling and redundancy levels are passively enforced by the CBAP.

- **Service discovery:** A directory of deployed services is maintained by the CBAP, allowing other services to more easily consume them.
- **Centralized logging and monitoring:** The CBAP provides a central logging facility to ease the management of distributed services. Services are continually monitored and policies allow the CBAP to remediate issues or to alert operators as needed.
- **Single point of control:** Operators can manage system performance at scale, from a single pane of glass or command line, and set policies that span applications, services and infrastructure.

IT organizations thus have in the container-based application platform a single, coherent environment that:

- **Supports self-contained microservices teams:** By abstracting application deployment and ops from infrastructure management, CBAPs help organizations build self-contained microservices teams without burdening them with low-level infrastructure management responsibilities.
- **Eliminates the operational complexity of microservices:** By automating service deployment and simplifying management, CBAPs help organizations overcome the operational challenges of deploying and managing microservices at scale.
- **Reduces the risk of building and maintaining large applications:** Microservices and continuous delivery each serves to reduce the risk of building and maintaining large software products. CBAPs make it easier to be successful with both.
- **Supports greater scalability and reliability:** CBAPs actively

enforce scalability and redundancy policies for microservices, leading to better user experience and uptime for applications.

- **Facilitates portability across infrastructures:** Modern, enterprise-oriented CBAPs support a wide choice of on-premises and cloud-based deployment, offering both choice and portability.

Conclusion

IT organizations are under increasing pressure to enable and drive businesses' innovation. In the context of software development, innovation is largely predicated on the ability to iterate quickly, particularly in an environment rife with unknowns and change — in other words, the real world.

Microservices architecture is an important trend that helps software development organizations iterate more quickly, adapt more fluidly to change, and reduce the risk of large software development projects. Adopting microservices is not without difficulty, however. Organizations that adopt microservices architectures face numerous development and operational challenges. Fortunately, though, in containers and container-based application platforms, there is help.

Container technologies address many of the issues associated with deploying individual services. Containers offer greater runtime efficiency than hypervisor-based virtual machines, less wasteful packaging than VM images, and greater portability and simpler deployment relative to traditional binaries.

Modern CBAP offerings pick up where containers leave off, and provide an automated platform for operationalizing microservices architectures.

CBAPs provide a highly automated, highly repeatable linkage between CI tooling and the deployment environment, while at the same time simplifying the full lifecycle management of deployed software products.

Together, microservices, containers and platform-as-a-service solutions offer significant advantages to organizations seeking to drive business innovation through innovative software.

*Published pursuant to a license granted by CloudPulse Strategies, LLC.
©2015, CloudPulse Strategies. ALL RIGHTS RESERVED.*

APPLICATIONS & MICROSERVICES DIRECTORY

MICROSERVICES FRAMEWORKS

	Product/Project (Company or Supporting Org.)	Languages Targeted
Open Source	actionhero.js (N/A) Actionhero.js is a multi-transport Node.js API server (framework) with integrated cluster capabilities and delayed tasks.	Node.js
Open Source	Airlift (N/A) Airlift is a framework for building REST services in Java.	JVM
Open Source	Akka (Typesafe) Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.	JVM
Open Source	AnyRPC (N/A) AnyRPC provides a common system to work with a number of different remote procedure call standards, including: JSON-RPC, XML-RPC, MessagePack-RPC.	C++, C
Open Source	Apollo (Spotify) Apollo is a set of Java libraries used when writing microservices. Apollo includes features such as an HTTP server and a URI routing system, making it trivial to implement RESTful services.	JVM
Open Source	Baucis (N/A) Baucis is used to build and maintain scalable HATEOAS/Level 3 REST APIs.	Node.js
Open Source	Beego (N/A) Beego is an open source, high performance web framework for the Go programming language.	Go
Open Source	C++ Micro Services (N/A) C++ Micro Services is an OSGi-like C++ dynamic module system and service registry.	C++
Open Source	Colossus (Tumblr) Colossus is an I/O and microservices library for Scala.	JVM

	Product/Project (Company or Supporting Org.)	Languages Targeted
Open Source	Compojure (N/A) Compojure is a concise routing library for Ring/Clojure.	JVM
Open Source	Cowboy (Nine Nines) Cowboy is a small, fast, modular HTTP server written in Erlang.	Erlang VM
Open Source	Dropwizard (N/A) Dropwizard is a Java framework for developing ops-friendly, high performance, RESTful web services.	JVM
Open Source	Duct (N/A) Duct is a minimal framework for building web applications in Clojure, with a strong emphasis on simplicity.	JVM
Open Source	Erlang/OTP (Ericsson) Erlang is a programming language used to build massively scalable, soft, real-time systems with requirements on high availability. OTP is set of Erlang libraries and design principles providing middleware to develop these systems. It includes its own distributed database, applications to interface towards other languages, debugging and release-handling tools.	Erlang VM
Open Source	Express (IBM) Express is a fast, unopinionated, minimalist web framework for Node.js.	Node.js
Open Source	Finagle (Twitter) Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers.	JVM
Open Source	Finatra (Twitter) Finatra is a fast, testable, Scala HTTP service built on TwitterServer and Finagle.	JVM
Open Source	Gen Microservice (N/A) Gen Microservice is a library for implementing microservices with Erlang.	Erlang VM
Open Source	Gin (N/A) Gin is a web framework written in Golang.	Go
Open Source	Go kit (N/A) Go kit is a distributed programming toolkit for microservices.	Go
Open Source	Gocraft (N/A) Gocraft is a toolkit for building web apps. Includes routing, middleware stacks, logging and monitoring.	Go

Product/Project (Company or Supporting Org.)		Languages Targeted
Open Source	Goji (N/A) Goji is a minimalistic web framework that values composability and simplicity.	Go
Open Source	Gorilla (N/A) Gorilla is a web toolkit for the Go programming language.	Go
Open Source	gRPC (N/A) gRPC is a high performance, open source, general RPC framework that puts mobile and HTTP/2 first.	Other
Open Source	H2 (Hailo) H2 is Hailo's microservices platform (framework).	Go
Open Source	Hapi (N/A) Hapi is a rich framework for building applications and services.	Node.js
Open Source	Jersey (Oracle) Jersey is RESTful web services in Java. It's a Jersey framework that provides its own API to extend the JAX-RS toolkit.	JVM
Open Source	Karyon (Netflix) Karyon is the base container for applications and services built using the NetflixOSS ecosystem.	N/A
Open Source	Kite (Koding) Kite is a microservices framework in Go.	Go
Open Source	Koa (N/A) Koa is a web framework for Node.js.	Node.js
Open Source	Kore (N/A) Kore is a web application framework for writing scalable web APIs in C.	C++, C
Open Source	Liberator (N/A) Liberator is a library that helps you expose your data as resources, while automatically complying with all the relevant requirements of the HTTP specification.	JVM
Open Source	Meteor (Meteor) Meteor is an ultra-simple environment for building JavaScript applications.	Other (JavaScript)
Open Source	Micro (N/A) Micro is a microservices toolkit. It simplifies writing and running distributed applications.	Go

	Product/Project (Company or Supporting Org.)	Languages Targeted
Open Source	Microserver (AOL) Microserver is Java 8 native, zero configuration, standards based, battle hardened library to run Java REST microservices.	JVM
Open Source	Mochiweb (N/A) Mochiweb is an Erlang library for building lightweight HTTP servers.	Erlang VM
Open Source	Modularity (JUXT) Modularity is JUXT's Clojure-based modular system.	JVM
Open Source	Nameko (N/A) Nameko is a microservices framework for Python that lets service developers concentrate on application logic and encourages testability.	Other (Python)
Open Source	Negroni (N/A) Negroni is an idiomatic HTTP middleware for Golang.	Go
Open Source	Orbit (Electronic Arts) Orbit is a modern framework for JVM languages that makes it easier to build and maintain distributed and scalable online services.	JVM
Open Source	Phoenix (DockYard) Phoenix is a framework for building HTML5 applications, API backends and distributed systems.	Erlang VM
Open Source	PIGATO (N/A) PIGATO is a high performance Node.js microservices framework based on ZeroMQ. PIGATO aims to offer a high performance, reliable, scalable and extensible service-oriented framework supporting multiple programming languages: Node.js, Io.js and Ruby.	Node.js
Open Source	Play (Typesafe) Play is a high velocity web framework for Java and Scala.	JVM
Open Source	Plug (N/A) Plug is a specification for composable modules between web applications.	Erlang VM
Open Source	QBit (N/A) QBit is a reactive programming library for building microservices. QBit uses reactive programming to build elastic REST, WebSockets-based, cloud-friendly web services.	JVM

	Product/Project (Company or Supporting Org.)	Languages Targeted
Open Source	Ratpack (N/A) Ratpack is a set of Java libraries (toolkit) that facilitate fast, efficient, evolvable and well tested HTTP applications. Specific support for the Groovy language is provided.	JVM
Open Source	RestExpress (N/A) RestExpress is a minimalist Java framework for rapidly creating scalable, containerless, RESTful microservices.	JVM
Open Source	Restify (N/A) Restify is a Node.js module built specifically to enable you to build correct REST web services.	Node.js
Open Source	Restlet Framework (Restlet) The Restlet Framework helps Java developers build web APIs that follow the REST architecture style.	JVM
Open Source	Revel (N/A) Revel is a high productivity, full-stack web framework for the Go language.	Go
Open Source	RIBS2 (N/A) RIBS2 is a library which allows building high performance Internet serving systems.	C++, C
Open Source	Sails.js (Balderdash) Sails.js is a web framework that makes it easy to build custom, enterprise-grade Node.js apps. It is designed to resemble the MVC architecture from frameworks, like Ruby on Rails, but with support for the more modern, data-oriented style of web app development. It's especially good for building real-time features like chat.	Node.js
Open Source	Scalatra (N/A) Scalatra is a simple, accessible microframework.	JVM
Open Source	ScaleCube (N/A) ScaleCube is a scalable microservices framework for the rapid development of distributed, resilient and reactive applications.	N/A
Open Source	Scotty (N/A) Scotty is a micro web framework inspired by Ruby's Sinatra, using WAI and Warp.	Haskell
Open Source	Seneca (nearForm) Seneca is a microservices toolkit for Node.js.	Node.js
Open Source	Skinny Micro (N/A) Skinny Micro is a web framework used to build servlet applications in Scala.	JVM

	Product/Project (Company or Supporting Org.)	Languages Targeted
Open Source	Spray (Typesafe) Spray is a toolkit for building REST/HTTP-based integration layers on top of Scala and Akka.	JVM
Open Source	Spring Boot (Pivotal Software) Spring Boot makes it easy to create stand-alone, production-grade Spring-based applications.	JVM
Open Source	Spring Cloud (Pivotal Software) The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications.	JVM
Open Source	StrongLoop LoopBack (IBM) LoopBack is an open source Node.js framework built on top of Express. It's optimized for creating APIs and easily connecting to backend data sources.	Node.js
Open Source	System (N/A) System is built on top of Stuart Sierra's component library, and offers a set of ready-made components.	JVM
Open Source	Tesla (OTTO) Tesla is a common basis for some of Otto.de's Clojure microservices.	JVM
Open Source	Vert.x (N/A) Vert.x is a toolkit for building reactive applications on the JVM.	JVM
Open Source	Vibe.d (Rejected Software) Vibe.d is synchronous I/O that doesn't get in your way. It is written in D.	Other (D)
Open Source	Yesod (N/A) Yesod is a Haskell RESTful web framework.	Haskell

PROVISION OF CAPABILITIES FOR MICROSERVICES

Product/Project (Company or Supporting Org.)		Capability Type
Open Source	ActiveMQ (Apache Software Foundation) ActiveMQ is a messaging and integration-patterns server.	Messaging
	API Management (IBM) The API Management service enables developers and organizations to manage and enforce policies around the consumption of their business services. Use an existing API or design a new API; then apply security controls, set rate limits, test APIs in place and finally publish these "managed APIs" on Bluemix.	API Gateways / Edge Services
Open Source	Apollo (Apache Software Foundation) Apollo is a messaging broker built from the foundations of the original ActiveMQ.	Messaging
	App Monitoring and Analytics (IBM) Build jobs, compile and package your app source code from Git or Jazz source control management (SCM) repositories. The build jobs produce deployable artifacts, such as WAR files or Docker containers, for IBM Containers for Bluemix. In addition, you can run unit tests within your build automatically. Each time the source code changes, a build is triggered.	Monitoring and Debugging
Open Source	Ascoltatori (N/A) Ascoltatori is a simple publish/subscribe library for Node.	Messaging
Open Source	Baker Street (Datawire) Baker Street is an HAProxy-based, client-side load balancer that simplifies scaling, testing and upgrading microservices.	Elasticity
Open Source	Beanstalk (N/A) Beanstalk is a simple, fast work queue.	Messaging
Open Source	BooPickle (N/A) BooPickle is a binary serialization library for efficient network communication.	Serialization

	Product/Project (Company or Supporting Org.)	Capability Type
Open Source	Camel (Apache Software Foundation) Camel helps define routing and mediation rules in a variety of domain-specific languages, including a Java-based fluent API, Spring or Blueprint XML configuration files, and a Scala DSL.	API Gateways / Edge Services
Open Source	Carbon (Common Sense Education) Carbon is one of the components of Graphite, and is responsible for receiving metrics over the network and writing them to disk using a storage backend.	Storage
Open Source	CBOR (N/A) The Concise Binary Object Representation (CBOR) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation.	Serialization
Open Source	Cereal (N/A) Cereal is a header-only C++11 serialization library.	Serialization
Open Source	Cheshire (N/A) Cheshire is fast JSON encoding, based off of clj-json and clojure-json, with additional features like Date/UUID/Set/Symbol encoding and SMILE support.	Serialization
Open Source	Chronos (Apache Software Foundation) Chronos is a distributed and fault-tolerant scheduler that runs on top of Mesos that can be used for job orchestration. Chronos is natively able to schedule jobs that run inside Docker containers.	Elasticity
Open Source	Consul (HashiCorp) Consul is a tool for service discovery and configuration. Consul is distributed, highly available and extremely scalable.	Configuration and Discovery
Open Source	Crate (Crate.io) Crate uses SQL syntax for distributed queries across a cluster.	Storage
Open Source	Crtauth (Spotify) crtauth is a public, key-backed, client/server authentication system.	Security
	Datomic (Cognitect) Datomic is a fully transactional, cloud-ready, distributed database.	Storage
Open Source	Denominator (Netflix) Denominator allows users to portably control DNS clouds using Java or Bash.	Configuration and Discovery

Product/Project (Company or Supporting Org.)		Capability Type
Open Source	Disque (N/A) Disque is an in-memory, distributed message broker.	Messaging
Open Source	Elliptics (Reverbrain) Elliptics is a fault tolerant, distributed key-value storage. It allows distributed storage of medium and large objects with streaming support.	Storage
Open Source	etcd (CoreOS) etcd is a distributed, consistent, key-value store for shared configuration and service discovery.	Configuration and Discovery
Open Source	Etch (Apache Software Foundation) Etch is a cross-platform, language- and transport-independent framework for building and consuming network services.	Serialization
Open Source	Eureka (Netflix) Eureka is a REST-based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.	Configuration and Discovery
Open Source	Fastjson (Alibaba Group) Fastjson is a JSON processor.	Serialization
Open Source	Fenzo (Netflix) Fenzo is an extensible scheduler for Mesos frameworks.	Elasticity
Open Source	ffjson (N/A) ffjson provides fast JSON serialization for Go.	Serialization
Open Source	Fluentd (Treasure Data) Fluentd is an open source data collector for unified logging layer.	Logging
Open Source	FST (N/A) FST is a fast Java serialization drop in replacement.	Serialization
	Galaxy (Parallel Universe) Galaxy is an open source, high performance, in-memory data grid that can serve as a basis for building distributed applications that require fine-tuned control over data placement and/or custom, distributed data structures.	Elasticity
Open Source	Geode (Apache Software Foundation) Geode is an open source, distributed, in-memory database for scale-out applications.	Storage

	Product/Project (Company or Supporting Org.)	Capability Type
Open Source	Grafana (N/A) Grafana is a metrics dashboard and graph editor for Graphite, InfluxDB and OpenTSDB.	Monitoring and Debugging
Open Source	Graphite-Web (Common Sense Education) Graphite-Web is a Django-based web application that renders real-time graphs and dashboards.	Monitoring and Debugging
Open Source	Graylog (Graylog) Graylog is a fully integrated platform for collecting, indexing, and analyzing both structured and unstructured data from almost any source.	Logging
Open Source	HAProxy (N/A) HAProxy is a reliable, high performance TCP/HTTP load balancer.	API Gateways / Edge Services
Open Source	Hazelcast (Hazelcast) Hazelcast is an open source, in-memory data grid. It allows you to distribute data and computation across servers, clusters and geographies, and to manage very large data sets or high data ingest rates. It's a mature technology.	Elasticity
Open Source	Helix (Apache Software Foundation) Helix is a generic, cluster management framework used for the automatic management of partitioned, replicated and distributed resources hosted on a cluster of nodes.	Elasticity
Open Source	Hystrix (Netflix) Hystrix is a latency and fault-tolerance library designed to isolate points of access to remote systems, services and third-party libraries, stop cascading failure, and enable resilience in complex distributed systems where failure is inevitable.	Resilience
Open Source	Ignite (Apache Software Foundation) Ignite is a high performance, integrated and distributed in-memory platform for computing and transacting on large-scale data sets in real time — orders of magnitude faster than is possible with traditional disk-based or flash technologies.	Elasticity
	IronMQ (Iron.io) IronMQ is a messaging queue that provides scheduling and communication between services and components.	Messaging
Open Source	Jackson (N/A) Jackson is a multipurpose Java library for processing JSON data format.	Serialization
Open Source	Jackson Afterburner (N/A) Jackson Afterburner adds dynamic bytecode generation for standard Jackson POJO serializers and deserializers, eliminating the majority of remaining data binding overhead.	Serialization

	Product/Project (Company or Supporting Org.)	Capability Type
Open Source	JSON Web Tokens (JWT) JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.	Security
Open Source	Kafka (Apache Software Foundation) Kafka provides publish-subscribe messaging as a distributed commit log.	Messaging
Open Source	Kibana (Elastic) Kibana is a flexible analytics and visualization platform.	Logging
Open Source	Kong (Mashape) Kong is a management layer for APIs. It has the capability of orchestrating Dockerfiles.	API Gateways / Edge Services
Open Source	Kryo (Esoteric Software) Kryo is a fast and efficient object graph serialization framework for Java.	Serialization
Open Source	Logstash (Elastic) Logstash is a tool for managing events and logs.	Logging
Open Source	Manta (Joyent) Manta is a HTTP-based object store that uses OS containers to allow running arbitrary compute on data at reset.	Storage
Open Source	Marathon (Apache Software Foundation) Marathon is an Apache Mesos framework for long-running applications. Marathon provides a REST API for starting, stopping and scaling applications. It lets users deploy, run and scale Docker containers.	Elasticity
Open Source	Mesos (Apache Software Foundation) Apache Mesos is a cluster manager that provides efficient resource isolation and sharing across distributed applications or frameworks.	Elasticity
	Message Hub (IBM) IBM Message Hub is a hosted, highly scalable, distributed, high throughput message bus based on Apache Kafka. The distributed nature of microservices can lead to latency challenges as the microservices systems grow. Use IBM Message Hub to implement asynchronous communication patterns to reduce latency and connect microservices across all of the IBM cloud platforms.	Messaging
Open Source	MessagePack (N/A) MessagePack is an object serialization library. It's like JSON, but very fast and small.	Serialization

Product/Project (Company or Supporting Org.)		Capability Type
Open Source	Mitmproxy (N/A) Mitmproxy is an interactive console program that allows traffic flows to be intercepted, inspected, modified and replayed.	Testing
Open Source	Mountebank (ThoughtWorks) Mountebank provides cross-platform, multi-protocol test doubles over the wire.	Testing
Open Source	Nanomsg (N/A) Nanomsg library is an implementation of several "scalability protocols." It is a socket library that provides several common communication patterns for building distributed systems.	Messaging
Open Source	NATS (Apcera) NATS is an open source, high performance, lightweight, cloud-native messaging system.	Messaging
Open Source	Nomad (HashiCorp) Nomad is a distributed, highly available, datacenter-aware scheduler.	Elasticity
Open Source	NSQ (N/A) NSQ is a real-time, distributed messaging platform.	Messaging
Open Source	OAuth (N/A) OAuth 2.0 is protocol that focuses on client/developer simplicity, while providing specific authorization flows for web applications, desktop applications, mobile phones and living room devices.	Security
Open Source	Onyx (N/A) Onyx is a platform for distributed, masterless, high performance, fault-tolerant data processing for Clojure.	Elasticity
Open Source	OpenID Connect (OpenID Foundation) OpenID Connect provides libraries, products and tools that implement current OpenID specifications and related specifications.	Security
Open Source	OpenResty (N/A) OpenResty is a web application server built on top of NGINX.	API Gateways / Edge Services
Open Source	Pathod (N/A) Pathod is a collection of pathological tools for testing and torturing HTTP clients and servers.	Resilience
Open Source	Prometheus (SoundCloud) Prometheus is an open source, service monitoring system and time series database.	Monitoring and Debugging

Product/Project (Company or Supporting Org.)		Capability Type
Open Source	Protostuff (N/A) Protostuff is a serialization library with built-in support for forward-backward compatibility (schema evolution) and validation.	Serialization
Open Source	Qpid (Apache Software Foundation) Qpid offers cross-platform messaging components built on AMQP.	Messaging
Open Source	RabbitMQ Server (Pivotal Software) RabbitMQ is an Erlang-based message broker that just works.	Messaging
Open Source	Raft Consensus (N/A) The Raft Consensus algorithm is designed to be easy to understand. It's equivalent to Paxos in fault-tolerance and performance.	Resilience
Open Source	Reactive Kafka (SoftwareMill) Reactive Kafka provides a Reactive Streams API for Apache Kafka.	Reactivity
Open Source	ReactiveX (N/A) ReactiveX is a library for composing asynchronous and event-based programs by using observable sequences. It provides an API for asynchronous programming with observable streams. Available for idiomatic Java, Scala, C#, C++, Clojure, JavaScript, Python, Groovy, JRuby and others.	Reactivity
Open Source	Reborn (N/A) Reborn is a proxy-based, high performance, Redis cluster solution. It is a distributed database.	Storage
Open Source	Redisson (N/A) Redisson is a distributed and scalable Java data structure on top of a Redis server.	Elasticity
Open Source	Resilient HTTP (N/A) Resilient HTTP is a smart HTTP client with superpowers, like fault tolerance, dynamic server discovery, auto balancing and reactive recovery — designed for distributed systems.	Resilience
Open Source	Riemann (N/A) Riemann monitors distributed systems. Riemann aggregates events from your servers and applications with a powerful stream-processing language. It allows you to review statistics from Riak nodes and forward them to Graphite.	Monitoring and Debugging
Open Source	Saboteur (N/A) Saboteur is a network, fault-injection tool that aims to simplify resilience and stability testing. Its core component is an agent that accepts commands over HTTP and configures its host's network stack for various common fault scenarios.	Resilience

	Product/Project (Company or Supporting Org.)	Capability Type
	SBinary (N/A) SBinary is a library for describing binary protocols in the form of mappings between Scala types and binary formats. It can be used as a robust serialization mechanism for Scala objects or a way of dealing with existing binary formats found in the wild.	Serialization
Open Source	SCIM (Internet Engineering Task Force) System for Cross-domain Identity Management (SCIM) is an open API for managing identities.	Security
Open Source	Sensu (Sensu) Sensu is a monitoring framework that aims to be simple, malleable and scalable.	Monitoring and Debugging
	Sensu Enterprise (Sensu) Sensu Enterprise is the commercial version of Sensu Core, a monitoring framework that aims to be simple, malleable and scalable.	Monitoring and Debugging
Open Source	Simian Army (Netflix) Simian Army is a suite of tools for keeping your cloud operating in top form. Chaos Monkey is a resiliency tool that helps ensure that your applications can tolerate random instance failures.	Resilience
Open Source	Simple React (AOL) Simple React provides powerful future streams and asynchronous data structures for Java 8.	Reactivity
Open Source	SkyDNS (N/A) SkyDNS is a distributed service for announcement and discovery of services built on top of etcd.	Configuration and Discovery
Open Source	SmartStack (Airbnb) SmartStack is Airbnb's automated service discovery and registration framework.	Configuration and Discovery
Open Source	Spring Cloud Config (Pivotal Software) Spring Cloud Config provides server- and client-side support for externalized configuration in a distributed system.	Configuration and Discovery
Open Source	Spring Cloud Netflix (Pivotal Software) Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.	Configuration and Discovery
Open Source	Strongloop API Gateway (IBM) The StrongLoop API Gateway acts as an intermediary gateway between API consumers (clients) and backend providers (API servers) that externalizes, secures and manages APIs.	API Gateways / Edge Services

Product/Project (Company or Supporting Org.)		Capability Type
Open Source	Suro (Netflix)	Logging
Suro is a distributed data pipeline which enables services for moving, aggregating, routing and storing data.		
Open Source	Tachyon (N/A)	Storage
Tachyon is a memory-centric, distributed storage system, enabling reliable data sharing at memory-speed across cluster frameworks.		
Open Source	Tengine (Alibaba Group)	API Gateways / Edge Services
Tengine is a distribution of NGINX with some advanced features. It powers Taobao, an ecommerce site in China.		
Open Source	Thrift (Apache Software Foundation)	Serialization
Thrift is a software framework for scalable, cross-language services development.		
Open Source	Trace (RisingStack)	Monitoring and Debugging
Trace is a visualized stack-trace platform designed for microservices monitoring.		
Open Source	Tyk (Tyk Technologies)	API Gateways / Edge Services
Tyk is a scalable API gateway, portal and API management platform.		
Open Source	Vcr (Relish)	Testing
Record your test suite's HTTP interactions and replay them during future test runs for fast, deterministic, accurate tests. See the list of ports for implementations in other languages.		
Open Source	Vulcand (N/A)	API Gateways / Edge Services
Vulcand is a programmatic load balancer backed by Etcd.		
Open Source	Wilma (EPAM Systems)	Testing
Wilma is a combined HTTP/HTTPS service stub and transparent Proxy. Its main purpose is to test applications, services and components for which functionality depends on other services.		
Open Source	WireMock (N/A)	Testing
WireMock is a flexible library for stubbing and mocking web services. Unlike general-purpose mocking tools, it works by creating an actual HTTP server that your code, under test, can connect to as it would a real web service.		
Open Source	Zabbix (Zabbix)	Monitoring and Debugging
Zabbix is an open source, enterprise-class monitoring solution.		
Open Source	ZeroMQ (ØMQ)	Messaging
ZeroMQ is a brokerless, intelligent transport layer.		

Product/Project (Company or Supporting Org.)		Capability Type
Open Source	ZooKeeper (Apache Software Foundation) ZooKeeper is an open source server that enables highly reliable, distributed coordination.	Configuration and Discovery
Open Source	Zuul (Netflix) Zuul is an edge service that provides dynamic routing, monitoring, resiliency, security and more.	API Gateways / Edge Services

DEPLOYMENT AND CONTINUOUS INTEGRATION

Product/Project (Company or Supporting Org.)	
Open Source	Active Deploy (IBM) Active Deploy allows you to release a new version of your software with no downtime. If at any time during the release a problem occurs, Active Deploy allows you to quickly revert back to the original version. You finalize the new version only when it has shown to work properly.
Open Source	AppVeyor (AppVeyor) AppVeyor is a continuous delivery service for Windows, focusing on .NET developers.
	Artifactory (JFrog) JFrog provides software developers with a binary repository management solution that integrates into CI/CD pipelines.
	Atlas (HashiCorp) Atlas unites Hashicorp development and infrastructure management tools to create a version control system for infrastructure.
	AWS CodeDeploy (Amazon Web Services) AWS CodeDeploy is a deployment service that enables developers to automate the deployment of applications to instances and to update the applications as required.
	AWS OpsWorks (Amazon Web Services) AWS OpsWorks provides a simple and flexible way to create and manage stacks and applications.
	Bamboo (Atlassian) Bamboo is a continuous integration and delivery tool that ties automated builds, tests and releases together in a single workflow.
	Bitnami (Bitnami) Bitnami is a library of server applications and development environments that can be installed with one click. Bitnami is beta testing functionality that will allow users to create container images.

Product/Project (Company or Supporting Org.)	
	<p>Bluemix (IBM)</p> <p>Bluemix is IBM's digital application platform, providing all you need to develop, deploy, manage and run cloud applications. Select your preferred managed platform for your applications from Containers, Virtual Machines and Cloud Foundry for instant run times. Freedom to develop microservices-based applications within a platform or with microservices deployed across multiple platforms, using Bluemix services for deployment, operations, logging and monitoring.</p>
Open Source	<p>Brooklyn (Apache Software Foundation)</p> <p>Apache Brooklyn is a library and control plane for deploying and managing distributed applications.</p>
	<p>Calm.io (Calm.io)</p> <p>Calm.io is a platform to deploy, manage and maintain distributed applications in private and public cloud environments. It claims to offer full-stack orchestration, run item lifecycle management, and policy-based governance.</p>
Open Source	<p>Capsule (N/A)</p> <p>Capsule is a packaging and deployment tool for JVM applications.</p>
Open Source	<p>Centurion (New Relic)</p> <p>Centurion is a deployment tool for Docker. It ships containers to and from a registry, running them on a fleet of hosts.</p>
Open Source	<p>Chef Delivery (Chef)</p> <p>Chef Delivery provides a workflow and visualization tool to help manage the continuous delivery pipeline.</p>
Open Source	<p>CircleCI (CircleCI)</p> <p>CircleCI provides a continuous delivery process for Docker applications.</p>
Open Source	<p>Lattice (Cloud Foundry Foundation)</p> <p>Lattice is an open source project for running containerized workloads on a cluster. Lattice bundles up HTTP load balancing, a cluster scheduler, log aggregation, and streaming and health management into an easy-to-deploy and easy-to-use package.</p>
	<p>CloudBees Jenkins Platform (CloudBees)</p> <p>CloudBees Jenkins Platform is a PaaS that supports continuous integration and delivery of mobile and web applications. It's the enterprise version of Jenkins that has multiple plugins to support Docker.</p>
Open Source	<p>Cloudbreak (Hortonworks)</p> <p>Cloudbreak helps users launch on-demand Hadoop clusters in the cloud or to any environment that supports Docker containers.</p>

Product/Project (Company or Supporting Org.)	
	<p>CloudSlang (Hewlett-Packard Enterprise)</p> <p>CloudSlang is a flow-based orchestration tool for managing deployed applications. It aims to automate DevOps workflows and offers Docker capabilities and integrations.</p>
Open Source	<p>Codefresh (Codefresh)</p> <p>Codefresh combines Docker tools with a web IDE based on Eclipse's Orion. The result is what they call a complete development environment for Node.js with DevOps and QA teams.</p>
Open Source	<p>Codenvy (Codenvy)</p> <p>Codenvy provisions, shares and scales developer environments. Users can use a Dockerfile to launch a custom stack.</p>
	<p>Codeship (Codeship)</p> <p>Codeship provides a hosted continuous delivery platform for web applications. It can be used to test Docker apps on different operating systems.</p>
	<p>ContainerShip (ContainerShip)</p> <p>ContainerShip is a self-hosted container management platform, capable of running on any cloud, and used to manage containers from development to production.</p>
	<p>DCHQ (DCHQ)</p> <p>DCHQ is a deployment automation and governance platform for Docker-based application development. The solution provides self-service access to Docker-based applications using an agent-based architecture for orchestration.</p>
	<p>Decking (Full Fat Finch)</p> <p>Decking aims to simplify three core areas: building images based on local Dockerfiles, creating containers and orchestration of containers.</p>
	<p>Delivery Pipeline (IBM)</p> <p>As you develop an app in the cloud, you can choose from several build types. You provide the build script, and IBM Bluemix DevOps Services runs it. With one click, you can automatically deploy your app to one or many Bluemix spaces, public Cloud Foundry servers, or Docker containers on IBM Containers for Bluemix.</p>
Open Source	<p>Distelli (Distelli)</p> <p>Distelli manages your applications as they progress through the development lifecycle with complete visibility of who did what and when along the way.</p>
	<p>Docker Compose (Docker)</p> <p>Compose is a tool for defining and running multi-container applications with Docker.</p>

Product/Project (Company or Supporting Org.)	
	<p>Docker Machine (Docker)</p> <p>Docker Machine lets you create Docker hosts on your computer.</p>
Open Source	<p>dockersh (Yelp)</p> <p>Dockersh is a login shell for machines with multiple users; it gives access to multiple users but allows for isolation between them.</p>
Open Source	<p>Drone (Drone.io)</p> <p>Drone is a continuous integration system built on top of Docker. Drone uses Docker containers to provision isolated testing environments.</p>
Open Source	<p>Dusty (GameChanger Media)</p> <p>Dusty is a development environment that provides OS X support for containers. Their documentation compares the tool to Vagrant and says it uses Docker Compose to orchestrate containers.</p>
	<p>Fabric8 (Red Hat)</p> <p>Fabric8 is an open source DevOps and integration platform that is built as a set of microservices that run on top of Kubernetes and OpenShift V3. Its continuous delivery is based on Jenkins, Nexus and SonarQube. Its iPaaS integrates with Apache ActiveMQ, Camel and CXF.</p>
Open Source	<p>Ferry (N/A)</p> <p>Ferry is a big data development environment. It lets you define, run and deploy big data applications on AWS, OpenStack and your local machine using Docker.</p>
Open Source	<p>Flockport Apps (Flockport)</p> <p>Flockport is a Linux container-sharing website. It also provides tools that make it easier to install and use LXC containers.</p>
Open Source	<p>Go Continuous Delivery (ThoughtWorks)</p> <p>Go Continuous Delivery helps you automate and streamline the build-test-release cycle for worry-free, continuous delivery of your product. This is a Java/JRuby on Rails project.</p>
	<p>Gradle (Gradle)</p> <p>Gradle is a build automation system. Gradle has been designed to support build automation across multiple languages and platforms — including Java, Scala, Android, C/C++ and Groovy — and is closely integrated with development tools and continuous integration servers, including Eclipse, IntelliJ, and Jenkins.</p>
Open Source	<p>Harbormaster (Crane Software)</p> <p>Crane's main product is Harbormaster, which is a Docker Release Management platform. It focuses on helping DevOps teams build, deploy and manage containers in production.</p>

Product/Project (Company or Supporting Org.)	
Open Source	Hook.io (Hook.io) Hook.io is an open source hosting platform for microservices.
	Hortonworks Data Platform (Hortonworks) Hortonworks Data Platform is a commercial Hadoop offering. With its 2015 purchase of SequenceIQ, Hortonworks acquired Cloudbreak's ability to launch on-demand Hadoop clusters in the cloud or to any environment that supports Docker containers.
	ION-Roller (Gilt) ION-Roller is an AWS immutable deployment framework for web services.
Open Source	Janky (GitHub) Janky is a continuous integration server built on top of Jenkins, controlled by Hubot, and designed for GitHub.
Open Source	Jenkins (N/A) Jenkins is an extensible open source continuous integration server.
Open Source	Kafka Deploy (N/A) Kafka Deploy enables automated deployment for a Kafka cluster on AWS.
Open Source	Lorry (CenturyLink Labs) Lorry.io is a CenturyLink Cloud utility and open source project that provides a graphical user interface for Docker Compose YAML validation and composition.
Open Source	Mantl (Cisco) Mantl is an open source platform for rapidly deploying globally distributed services. It works with tools such as Marathon, Mesos, Docker and Consul.
Open Source	nscale (nearForm) nscale is an open toolkit which supports configuration, build and deployment of connected container sets.
Open Source	Otto (HashiCorp) Otto is a development and deployment tool that is the successor to Vagrant.
Open Source	Packer (HashiCorp) Packer is a tool for creating machine and container images for multiple platforms from a single source configuration.
Open Source	Panamax (CenturyLink Labs) Panamax is a containerized app creator with an open source app marketplace hosted on GitHub. Panamax provides an interface for users of Docker, Fleet and CoreOS.

Product/Project (Company or Supporting Org.)	
Open Source	Powerstrip (ClusterHQ) Powerstrip is a tool for prototyping Docker extensions.
Open Source	Project Shipped (Cisco) Cisco's Project Shipped is a model for deploying microservices to the cloud. To reduce environmental inconsistencies, Project Shipped also emulates the cloud environment on your development workstation. Vagrant, Consul and Cisco Microservices Infrastructure are components.
Open Source	PureApplication (IBM) PureApplication offers solutions that use automated patterns to rapidly deploy applications, reduce cost and complexity, and ease management. Docker containers can be included in patterns along with non-container components to take advantage of the PureApplication enterprise-grade lifecycle management. Support includes a private Docker registry pattern deployable as a shared service.
	runC (Linux Foundation) runC is a CLI tool for creating and running containers according to the Open Container Initiative's specification. It is the result of a multi-company coalition and is based on Docker's libcontainer project.
	Runnable (Runnable) Runnable works with GitHub and other tools to automatically deploy commits and launch containers in your sandbox when branches are created.
Open Source	Shippable CI/CD (Shippable) Shippable is a continuous integration platform built natively on Docker and using Docker Hub to deploy.
	Shippable for OpenShift (Shippable) Shippable is a continuous integration platform built natively on Docker and using Docker Hub to deploy.
	ShutIt (N/A) ShutIt is a tool for managing the Docker image building process; it expands on some of the capabilities of Dockerfiles.
	Spinnaker (Netflix) Spinnaker is an open source, multi-cloud, continuous delivery platform for releasing software changes with high velocity and confidence.
Open Source	Spoon (Spoon) Spoon is a platform for building, testing and deploying Windows applications and services in isolated containers.

Product/Project (Company or Supporting Org.)	
	<p>StackEngine (Oracle)</p> <p>StackEngine is an end-to-end container application management system that provides a way for dev and enterprise IT teams to deploy Docker applications.</p>
	<p>StackHut (StackHut)</p> <p>StackHut is a platform to develop and deploy microservices without writing any server-logic. It takes a regular class (in Python or JavaScript for now), a YAML file describing your stack, and deploys a microservice whose functions can be called natively in other languages or through REST.</p>
	<p>TeamCity (JetBrains)</p> <p>TeamCity is a continuous integration and build server for developers and DevOps.</p>
Open Source	<p>Terraform (HashiCorp)</p> <p>Terraform is a tool to build and launch infrastructure, including containers.</p>
	<p>Totem (N/A)</p> <p>Totem is a continuous delivery pipeline tool designed for microservices.</p>
Open Source	<p>Travis CI (Travis CI)</p> <p>Travis CI is an open source continuous deployment platform; it is able to run on Docker-based infrastructures.</p>
Open Source	<p>UrbanCode Build (IBM)</p> <p>UrbanCode Build is a continuous integration and build management server optimized for the enterprise. It is designed to make it easy to scale and seamlessly plug into development, testing and release tooling. It supports Docker build and integration with Docker registries.</p>
	<p>UrbanCode Deploy (IBM)</p> <p>UrbanCode Deploy orchestrates the deployment of applications across environments, coordinating the deployment of many individual components with inventory tracking. This includes support for Docker Containers, Docker Registries, and IBM Container Service on Bluemix via a community plugin.</p>
	<p>UrbanCode Release (IBM)</p> <p>UrbanCode Release is a collaborative release management tool that handles the growing number and complexity of releases.</p>
	<p>Vagrant (HashiCorp)</p> <p>Vagrant is a tool for building and distributing development environments. Development environments managed by Vagrant can run on containers.</p>

Product/Project (Company or Supporting Org.)	
	<p>Vamp (Magnetic.io)</p> <p>Vamp stands for Very Awesome Microservices Platform. It helps developers build, deploy and manage microservices. Vamp's core features are a platform-agnostic microservices DSL, powerful A/B testing, canary releasing, autoscaling and an integrated metrics/event engine.</p>
Open Source	<p>Vault (HashiCorp)</p> <p>Vault secures, stores and tightly controls access to tokens, passwords, certificates, API keys and other secrets in modern computing.</p>
Open Source	<p>Vessel (N/A)</p> <p>Vessel automates the setup and use of dockerized development environments. It requires both OS X and Vagrant to work properly.</p>
	<p>Virtuozzo (Odin)</p> <p>Odin's Virtuozzo is a container virtualization platform sold to providers of cloud services.</p>
	<p>Weave Run (Weaveworks)</p> <p>Weave Run provides routing and control for microservices applications.</p>
	<p>Wercker (Wercker)</p> <p>Wercker is a platform for automating the creation and deployment of applications and microservices.</p>
	<p>xDock (Xervmon)</p> <p>Xervmon is a cloud management platform. Its xDock lets users deploy, manage and monitor Docker images in the cloud.</p>
	<p>XL Deploy (Xebia Labs)</p> <p>XL Deploy provides application release automation to standardize complex deployments, speed up development time, and gain visibility into the pipeline.</p>
	<p>XL Release (Xebia Labs)</p> <p>XL Release automates, orchestrates and provides visibility into release pipelines. It identifies bottlenecks, reduces errors and lowers the risk of release failures.</p>
Open Source	<p>Zodiac (CenturyLink Labs)</p> <p>Zodiac is a lightweight tool, built on top of Docker Compose, for easy deployment and rollback of "Dockerized" applications.</p>

DISCLOSURES

The following companies mentioned in this ebook are sponsors of The New Stack: ActiveState, Apprenda, Cloudsoft, Codenvy, Hewlett-Packard, Intel, New Relic, SAP, Teridion and Weaveworks.

[Horse](#) and [Elephant](#) icons, by [Joao Santos](#) from [thenounproject.com](#), is licensed under [CC BY 3.0 US](#).

[Whale](#), [Office Worker](#), [Female Avatar](#), [Office Building](#) and [Coding](#) icons designed by [Freepik](#), from [flaticon.com](#)

