Microservices-Based Software Architecture and Approaches

Kapil Bakshi
Cisco Systems, Inc.
13635 Dulles Technology Drive
Herndon, VA 20171
703 484 2057
kabakshi@cisco.com

***Abstract*—In the last few years, a revised software architecture style has been developed to design new software applications. This architecture style is particularly suited for use cases in the aerospace industry, from an independently deployable software service. The microservices architectural style develops a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. These services are built around business and mission capabilities and independently deployable by fully automated machinery. With microservices, some types of applications become easier to build and maintain when they are broken down into smaller, composable pieces that work together. Each component is then developed separately, and the application is then simply the sum of its constituent components. This is in contrast to a traditional, "monolithic" application which is all developed in one piece. This paper will discuss, several aspects of microservices-based architecture, including several potential use cases for the aerospace industry. The characteristics of microservice-based architecture such as componentization, organization, endpoints and messaging mechanisms. The technical implementation of microservices by reviewing containerization, services communication and related architectural components. Specific open source projects and components that can be utilized to build microservices-based architecture. A sample set of use cases.**

TABLE OF CONTENTS

1. INTRODUCTION

Today, there are several trends that are forcing application architectures to evolve. End users and missions expect a rich, interactive and dynamic experience via the next generation of clients, including mobile devices. Next generation applications must be highly scalable, highly available and be able to interface with cloud environments. Organizations often want to frequently roll out updates on a regular basis. Consequently, it is no longer adequate to develop simple, monolithic web applications that serve up HTML to desktop browsers. Over the years, most enterprise software applications have been designed and developed as large, complex, monolithic applications. In a monolithic approach, applications are decomposed into layered architectures such as model view controllers. However, applications are still optimized with regard to the level of functions that each of the applications are responsible for, therefore making them large and complex to manage. A monolithic application is an application where all of the logic runs in a single app server. Typical monolithic applications are large and built by multiple teams, requiring careful orchestration of deployment for every change. They usually have a large application code base, which often challenges new developers as they join a large project team. This then requires a significant amount of time to become familiar with the code base. With monolithic applications, even a small change might have a large ripple effect with regard to regression testing. Monolithic applications require a built up history of a large suite of regression tests. This adds to testing time cycles and also to the number of testers required, but missions and businesses want their teams to be able to respond to new requirements. Microservices allows for more frequent delivery and faster delivery times. This enables application owners to receive quicker feedback and make adjustments to their requirements. Large monolithic applications often have a large business scope and many infrastructure touch points. Any change to the application could result in multiple reviews and approvals, thus resulting in increased deployment cycle times. By adopting a microservices architectural style with a single platform enables service management teams to more easily support multiple product and service teams. Efficiencies can be realized by automating deployment, logging, and monitoring practices across multiple microservices project teams. Hence, there is a strong impetus to review a new architectural style for software applications.

2. ELEMENTS OF MICROSERVICES

Microservices Defined

Microservices are small, autonomous services that work together. [1] The key here is small and autonomous. Microservices is an architecture style, in which large complex software applications are composed of one or more services. Microservices can be deployed independently of one another and are loosely coupled and each of these microservices focuses on completing one task only. Microservices support multiple clients in a typical architecture. Additionally, there are several factors to be considered in a typical Microservices based application,

978-1-5090-1613-6/17/$31.00 ©2017 IEEE

called 12-factor app, which will be reviewed in following section.

Microservices Compared

Microservices is an independent unit of work, which is small enough to stand on its own in terms of functions. There must be zero coordination for the deployment with other microservices. Loose coupling enables frequent and rapid deployments of applications, therefore, getting much-needed features and capabilities to the consumers. An example of such comparison is portrayed in Table 1.

Microservices need to be built using the programming language that makes the most sense for the final outcome. They are composed together to form a complex application and do not need to be written using the same programming language. In some cases, Java may be the correct language, and in others it might be Python. Communication with microservices is through language-neutral APIs, typically a Hypertext Transfer Protocol (HTTP)-based resource Application Program Interface (API) such as REST.

Category	Monolithic architecture	Microservices architecture
Code	A single code base for the entire application.	Multiple code bases. Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled downtimes.	Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime.
Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.

Table 1: Comparing Monolithic and Microservice Architecture. [2]

Characteristics of Microservices

There are several characteristics that need to be considered in a microservices architecture. We will review and enumerate this below.

Microservices need to be designed around business capabilities. This approach takes a broad-stack implementation of software for that business area, including user-interface, persistent storage, and any external collaborations. As a result, teams are cross-functional, with a full range of skills required for the development: user-experience, database, and project management. [3]

The above characteristics imply that one has to follow the componentized model for services. Hence, a component is a unit of software that is independently replaceable and upgradeable. [3] Microservice architectures can use traditional libraries, but their primary way of componentizing their own software is by breaking down into services, which are components who communicate with a mechanism such as a web service request, or remote procedure call. The primary reason for using services as components (rather than libraries) is that services are independently deployable. An application, which consists of multiple libraries in a single process (perhaps a change to any single component), results in having to redeploy the entire application. However, if that application is decomposed into multiple services, many single service changes will be required for that service to be redeployed. Another reason for using services as components is a more explicit component interface. Most languages do not have a good mechanism for defining; therefore, componentization with interfaces makes sense for microservices architecture.

Monolithic application development efforts leverage the project model, where the goal is to deliver software, which is then considered to be complete on delivery. On completion, the software is handed over to a maintenance organization and the project team then moves to a new project. The microservice approach leverages the Continuous Development and Continuous Integration (CI/CD) process where software is continuously developed and features are also added. This process is constantly integrated with a development, test, staging and production environment.

Monolithic approaches require an intelligent communication system for application layers, for example, Enterprise Service Bus (ESB), where ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules. [2]

Microservices take an alternative approach: smart endpoints and dumb pipes. [3] Applications built from microservices decoupled their own logic, receiving a request, applying logic as appropriate, and producing a response. This communication is done using simple RESTish protocols rather than complex protocols such as WS or BPEL or orchestration by a central tool. Additionally, the common use is messaging over a lightweight message bus. The infrastructure chosen is typically dumb simple implementations such as RabbitMQ or ZeroMQ, which provide a reliable asynchronous fabric. The intelligence still exist in the endpoints that are producing and consuming messages in the services.

Microservices with CI/CD require infrastructure automation techniques. In this model, teams automate deployment to each new environment. Another area where development teams use infrastructure automation is for managing microservices in production.

Since microservices are designed as service components, applications need to be designed so that they can tolerate the failure of services. Any service call could fail due to unavailability of the infrastructure. This is a disadvantage compared to a monolithic design as it introduces additional complexity. The consequence is that microservice teams are

aware of how service failures affect the end-user experience. Netflix's Simian Army induces failures of services, including datacenter processes during the working day to test both the application's resilience and monitoring. [3] Services can fail at any time, therefore, it is important to detect failures quickly and automatically restore service. Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (such as how many requests per second is the database receiving) and business relevant metrics (such as how many orders per minute are received). Microservice teams would expect to see monitoring and logging approached for each individual service such as dashboards showing status and a variety of operational and business relevant metrics. Details on circuit breaker status, current throughput and latency are other examples of this characteristic.

The microservices approach works well with decentralization of logic and data. At an abstract level, the application is divided into separate components. Another way to consider decentralization is the Domain Driven Design (DDD) notion of Bounded Context. [3] DDD divides a complex domain into multiple bounded contexts and maps out the relationships between them. This process is useful for both monolithic and microservice architectures, but there is a natural correlation between service and context boundaries that helps clarify, and as we describe in the section on business capabilities, reinforce the separations. Microservices also decentralize data storage decisions. While monolithic applications prefer a single-logical database for persistant data, Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems, also called Polyglot Persistence. One can use Polyglot Persistence in a monolith, but it appears more frequently with microservices.

The Microservice design stipulates that the application be broken into several service components: governance and management approaches also need to be considered. The monolothic approach is a centralized governance, which implies a tendency to standardize on single technology platforms. This approach can be too restricting. By splitting the monolith's components out into services, we have a choice when building each of them in languages and styles. Development teams can code each service in the language they are comfortable with — namely, Node.js, Python, Java or C++. One can apply the same approach to database platforms. Teams that are building microservices prefer a different approach to standards. Rather than use a set of defined standards, they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing. These tools are usually harvested from implementations and shared with a wider group; sometimes, but not exclusively using an internal open source model. Examples such as git and github have become the de facto version control system of choice.

12-Factor Application Method and Microservices

A common methodology, called the 12-Factor App methodology, has emerged with the purpose of providing an outline for building well-structured and scalable applications with the microservices approach. The application deployment process can be complicated and extensive. Virtualization, networking, and setting up runtime environments are just a few of the challenges involved. The 12-Factor App methodology helps to create a framework for organizing the process in order to maintain a healthy and scalable application. [4] The following paragraphs explain how to meet the standards of the 12-Factor App methodology, while addressing application infrastructure and deployment setup.

Codebase

A 12-Factor App has a version control system such as git. A copy of the revision-tracking database is known as a code repository. A codebase is any single repository or in any set of repository that shares a root commit. [4] A 12-Factor App requires one-to-one correlation between the codebase and the application. If there are multiple codebases, it is a distributed system with many applications. Each component in a distributed system is an application, and each can individually comply with 12-Factor. Multiple applications sharing the same code is a violation of 12-Factor. There is only one codebase per application, but there will be many deploys of the app. A deploy is a running instance of the application. This is typically a production site and one or more staging sites.

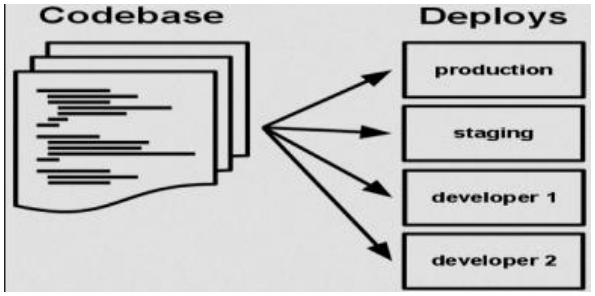

Figure 1. Code Base Repository [4]

Additionally, every developer has a copy of the app running in their local development environment, each of which also qualifies as a deploy.

Dependencies

A 12-Factor App declares all dependencies; it never relies on implicit existence of system-wide packages via a dependency declaration manifest. An explicit dependency specification is applied to both production and development. [4] The advantage of explicit dependency declaration is that it simplifies setup for developers new to the application. The new developer can check out the codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. They will be able to set up everything needed to run the app's

code with a deterministic build command. 12-Factor Apps also do not depend on the implicit existence of any system tools. These tools may already exist on deployment and test systems, but there is no guarantee that they will exist on all systems where the app may run.

Config

An app's config typically changes from different deployment environments like staging, production, and development. The config could include resource handles to databases, credentials to external services, host names, etc. 12-Factor requires strict separation of config from code, as config varies across deployments, while code does not. [4] One approach to config is the use of config files, which are not checked into revision control. However, there is a tendency for config files to be scattered in different places and different formats, making it hard to see and manage all in one place. Another approach is to store config in environment variables. Env vars are easy to change between deploys without changing any code. Yet, another approach, to config management is grouping. Sometimes apps batch configs into named groups (often called "environments") named after specific deploys such as the development, test, and production environments. In a 12-Factor application, env vars are granular controls, each disjointed to other env vars. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime.

Backing Services

A backing service is a service the application consumes over the network as part of its normal operation. [4] Examples include datastores, messaging/queueing systems, SMTP services for outbound email, and caching systems. This also includes third-party and non-traditional (like cloud) services. The code for a 12-Factor app makes no distinction between local and third-party services. A deployment of the 12-Factor app should be able to swap out a local database (MySQL) with one managed by a third party (such as Amazon S3) without any changes to the app's code. Each distinct backing service is a resource. For example, a MySQL database is a resource.

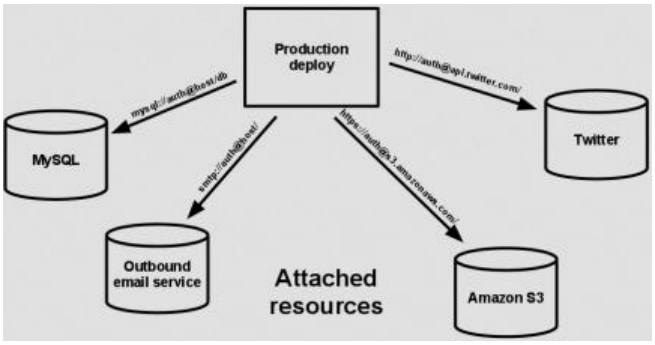

Figure 2. Backing Services [4]

The 12-Factor treats these databases as attached resources, which implies loose coupling. Resources can be attached and detached to deploy at will. For example, if the app's database is misbehaving due to a hardware issue, the app's administrator might spin up a new database server restored from a recent backup.

Build, Release and Run

A codebase is delivered to deployment through three stages — Build, Release and Run. First stage is Build, which converts a code repo into an executable bundle. Using a version of the code at a commitment specified by the deployment process, the build stage gathers dependencies and compiles binaries and assets. [4] Second stage is Release, which takes the build produced by the build stage and combines it with the deploy's current config. The release contains both the build and the config and is ready for execution in the deployment environment. The third stage is Run, which runs the app in the execution environment by launching some set of the applications against a selected release. A 12-Factor App uses a strict separation between the build, release, and run stages. Builds are initiated by the application's developers whenever new code is deployed. Runtime execution happens automatically in cases such as a server reboot or a crashed process being restarted by the process manager. Hence, the run stages should be simple and constant, since there are problems that prevent an app from running. The build stage can be more complex since errors are always in the foreground for developers who are managing the deployment.

Process

The application is executed in an environment as one or more processes. On one end of the continuum, the code is a stand-alone, and the execution environment is a developer's local system with an installed language runtime. [4] On the other end of the spectrum, a production deployment of a sophisticated app may use many processes. 12-Factor processes are stateless and share-nothing, hence any persistent data needs to be in a backing service, typically a database. The memory space or file system of the process can be used for a single-transaction cache; for example, downloading a large file, operations, and storing the results of the operation in the database. The 12-Factor App assumes that any data cached in memory or on disk will be unavailable for job request. Some web-based systems rely on caching user session data in memory of the process and expect future requests from the same visitor to be routed to the same process. Sticky sessions are a violation of 12-Factor and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration.

Port Binding

12-Factor Apps are sometimes executed inside a webserver container and completely self-contained and do not rely on runtime injection of a webserver into the execution environment to create a web-facing service. [4] The web app exports HTTP, as a service by binding to a port and listening to requests coming in on that port. In a local development environment, a service URL, for example,

http://localhost:5000/, is used to access the service exported app. In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes. This is implemented by using dependency declaration to add a webserver library to the app. This happens entirely in the user space, that is, within the app's code. The contract with the execution environment is binding to a port to serve requests.

Concurrency

In the 12-Factor App, processes are first-class citizens [4]. Processes in the 12-Factor App are built on a unix process model. Using this model, the developer can architect the application to handle diverse workloads by assigning each type of work to a process type. For example, HTTP requests may be handled by a web process, and background tasks handled by a worker process. This process model is a scale-out approach. The share-nothing, partitioned process architecture provides a simple approach to concurrency. 12-Factor App processes should leverage the operating system's process manager to manage task output streams and handle user-initiated restarts and shutdowns.

Disposability

The 12-Factor Apps processes are disposable, as they can be started or stopped easily and instantaneously. [4] This enables fast, elastic scaling, rapid deployments of code and configs. Short process startup time provides agility scaling up and movement of process to different deployment setups. Typically, a process takes a few seconds to launch, until the process is available to receive requests or jobs. Consequently, a proper process shutdown is done by returning the current job to the work queue. A recommended approach is to use a robust queueing backend that returns jobs to the queue when clients disconnect or time out. Processes should also be robust against sudden death, in the case of a failure in the underlying hardware. A 12-Factor App is architected to handle unexpected, non-graceful terminations.

Development and Production Parity

The gaps between development and production environments are typically due to either time gap, as development releases code and operations deploy it, or a tool gap between developers and operations team as they may use different tools. A 12-Factor App is designed for continuous development with minimal gap in environments. [4] Additionally, the 12-Factor App should use the same backing services between development and production, even when adapters abstract away any differences in backing services. Differences between backing services could mean incompatibilities, defeating the purpose of continuous deployment. Adapters to different backing services are still useful because they make porting to new backing services relatively simple. But all deploys of the app (developer environments, staging, production) should be using the same type and version of each of the backing services.

Logs

Logs provide a time-ordered status of processes and backing services of an application and are typically written to disk files. A 12-Factor App should not attempt to write to or manage logfiles. During local development, the developer views streams in the foreground of their terminal to observe the process behavior. [4] In staging or production deploys, each process stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival. These archival destinations are not visible to or configurable by the application, and instead, are completely managed by the execution environment.

Administration Process

In a 12-Factor App, admin tasks should be run as a one-off process. Developers often utilize one-off administrative or maintenance tasks for the app such as, database migrations, console or to inspect the app's models against the live database. [4] One-off admin processes should be run in an identical environment as the regular long process of the app, and against a release, using the same codebase and config. The admin code must ship with the application code to avoid synchronization issues.

Additional Considered factors

There could be additional factors to consider include Scaling services, via mechanisms like auto scaling and load balancers. Also integration of desperate services with external service also needs to be considered. Additionally, deugging techniques also need to be considered. Finally, security requirements would also need to be reviewed in the microservices architecture.

3. MICROSERVICES ARCHITECTURE

Microservices architecture consists of several components. These components work together to provide distributed services that have been discussed in the previous sections. The key components are as follows.

Containers

Containers have been in deployments for some time now in the form of Linux containers and several commercial implementations (like Dockers). Linux containers are self-contained execution environments with their own isolated CPU, memory, block I/O, and network resources that share the kernel of the host-operating system. [5] The experience is like a virtual machine, but without the weight and startup overhead of a guest operating system.

Two key elements of containers are Linux cgroups and namespace, the Linux kernel features that isolate containers and other processes running on the host. [5] Linux namespaces, originally developed by IBM, wrap a set of system resources and present them to a process to make it look like they are dedicated to that process. Linux cgroups,

originally developed by Google, govern the isolation and usage of system resources such as CPU and memory for a group of processes. For example, if you have an application that takes up a lot of CPU cycles and memory such as a scientific computing application, you can put the application in a cgroup to limit its CPU and memory usage. Namespaces deal with resource isolation for a single process, while cgroups manage resources for a group of processes.

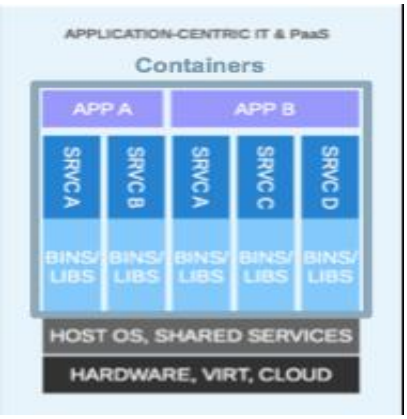

Figure 3. Container Architecture for Applications

Image-based containers package applications with individual runtime stacks, making the resultant containers independent from the host-operating system. Thus making it possible to run several instances of an application, each developed for a different platform. [5] This is possible because the container run time and the application run time are deployed in the form of an image.

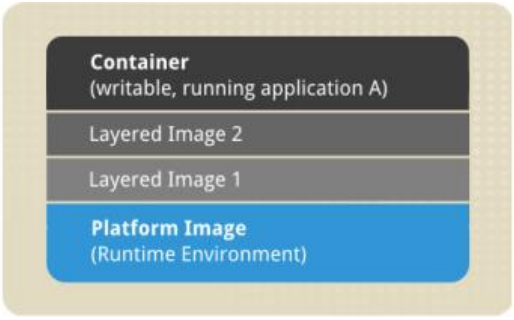

Figure 4. Image-Based Container

Figure 4 describes a typical image-based container.

A container is an active component in which an application runs. Each container is based on an image that holds necessary configuration data. When a container is launched from an image, a writable layer is added on top of this image. An image is a static snapshot of the container's configuration and is a read-only layer. All changes are made in a top-most writable layer and can be saved only by creating a new image. Each image depends on one or more parent images. A Platform Image is an image that has no parent. Platform images define the runtime environment, packages and utilities necessary for a containerized application to run.

Services Communication

In a monolithic application, components are invoked via function calls. In contrast, microservices-based services interact using an Inter-Process Communication (IPC) mechanism. When selecting an IPC mechanism for a service, it is important to consider services interaction. One dimension is whether the interaction is synchronous or asynchronous. With synchronous interaction, the client expects a timely response from the service and might even block while it waits. With asynchronous interaction, the client does not block while waiting for a response. Additionally, there are other types of one-to-one interactions — namely, request/response where a client makes a request to a service and waits for a response. Notification occurs where the client sends a request to a service, but no reply is expected or sent. Request/async response is where a client sends a request to a service, which replies asynchronously. There are different types of one-to-many interactions: Publish/subscribe, where client publishes a notification message, which is consumed by zero or more interested services; Publish/async responses, where a client publishes a request message, and then waits a certain amount of time for responses from interested services. Each service typically uses a combination of these interaction styles. Let us review the IPC mechanisms that a developer can implement in microservice architecture.

IPC Technologies

There are several IPC mechanisms that can be applied to microservices. Services can use synchronous request/ response-based communication mechanisms such as HTTP-based REST or Thrift. Alternatively, they can use asynchronous, message-based communication mechanisms such as Advanced Message Queuing Protocol (AMQP). [6]

In an asynchronous system, client makes a request to a service by sending it a message. If the service replies, it sends a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. Instead, the client is written assuming that the reply will not be received immediately. There are a large number of open source messaging systems to choose from, including RabbitMQ, Apache Kafka, Apache ActiveMQ.

When using a synchronous, request/response-based IPC mechanism, a client sends a request to a service. The service processes the request and sends back a response. In many clients, the thread that makes the request blocks while waiting for a response. There are numerous protocols to choose from. Two popular protocols are REST and Thrift. Let us first take a look at REST.

Service Registry and Discovery

In a microservices application, the set of running service instances changes dynamically, including network locations. Consequently, in order for a client to make a request to a service, it must use a service-discovery mechanism. A key part of service discovery is the service registry. The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and deregistered from the service registry using the management API. The query API is used by system components to discover available service instances. There are two service-discovery patterns: client-side discovery and service-side discovery. In systems that leverage client-side service delivery, clients query the service registry, select an available instance, and make a request. In systems that use server-side discovery, clients make requests via a router, which queries the service registry and forwards the request to an available instance. There are two methods where service instances are registered with and deregistered from the service registry. One method is for service instances to register themselves with the service registry, the self-registration. The other method is a system component to handle the registration and deregistration on behalf of the service. In deployment environments one needs to setup a service-discovery infrastructure using a service registry such as etcd and Apache Zookeeper. In other deployment environments, service discovery is built in. For example, Kubernetes will service instance-registration and deregistration.

4. SAMPLE AEROSPACE USE CASES

There are several aerospace-specific use cases that can be implemented by adopting a microservices-based approach and architecture. This section provides a selection of use cases that could be leveraged by the aerospace industry, as well as other industries.

Mission Data Analytics

Data Analytics is traditionally real time, batch, and archived in nature. Technically, the solution involves accessing aerospace sensor data from a digital source. If the sensors are analog, there needs to be some component within the architecture that converts the analog data to digital, for example, with an Analog-to-Digital (AD) device in the field. The analytics services can be developed to perform small tasks, which can collectively give the outcome of mission. The key here is that different services can be running on a variety of systems and invoked independently. For example, the edge analytics service can be run on a network edge device (like a router), and batch service on a Hadoop cluster at the data center.

Equipment Health Monitoring

The deployment of equipment health monitoring and predictive analytics technologies promises to deliver benefits to aerospace and defense companies. Microservice architecture can be implemented for this use case. For example, imagine that a space flight experiences an intermittent hydraulic fault code. The equipment health monitoring system performs a local analytics service and sends a notification message to the ground station where maintenance control personnel run the fault detail service. After identifying the likely problem, maintenance control orders a mitigation process. When the aircraft lands, the ground crew replaces the pressure transmitter during the scheduled cleaning and refueling time. In this use case, the microservices performed the predictive analytics locally and only transmitted the notification to the ground crew. Consequently, the equipment health data can be locally collected and then transferred to a core data center for batch historical and trend services.

Manufacturing Visibility and Management

A microservices-based architecture for manufacturing can provide aerospace production line information to manufacturing plant staff and improves factory efficiency. For example, a plant manager on the production floor could also use connected floor and visibility tools to access the efficiency of each machine, view production from any location, and reduce the time to decision and action. The manufacturing facility and equipment assembly line can be connected to a service that will collect real-time data on the status of the plant. So, instead of managing a control room, facilities managers and production personnel can have easy access to real-time information and collaborate more effectively. For example, if there is a quality control problem on a production line, they can shut down the line before it continues to create products where possibly all products could be wasted. The benefits of this architecture can extend beyond the enterprise to a wide range of suppliers and third-party providers of services, consumables, and capital goods.

5. SUMMARY

Whether or not microservices architecture becomes the preferred style of developers in the future, it clearly has strong benefits for designing and implementing applications. However, as with gaining adoption of any new technology, there are a few challenges, for example, incorrect requirements. If we start with requirements that are wrong, it will not matter how fast we can develop, deploy, or scale a system. There is also a lack of frameworks to support this from an operational perspective. Additionally, substantial DevOps skills are required to keep microservices running and available. Finally, when breaking an existing monolithic system into collaborating components, you are introducing interfaces between them. Each interface requires maintenance across releases, despite some capabilities available using leading practices such as compatibility with earlier versions. In the future, we expect to see standards for describing microservices. Currently, one can use any technology to create a microservice, if it exposes JavaScript Object Notation (JSON) or Extensible Markup Language (XML) over the HTTP to provide a REST API. Thus the future of the microservices approach is promising and

applications would overtime migrate to this revised architecture.

REFERENCES

[1] Newman, S. (February 2015). Building Microservices. USA: O'Reilly Media Inc.

[2] Daya, S., Duy, N., Eati, K., Ferreira, C., Glozic, D., Gucer, V., Gupta M., Joshi, S., Lampkin, V., Martins, M., Narain, S., Vennam, R. (August 2015) Microservices from Theory to Practice. IBM Redbook.

[3] Fowler, M., Lewis, J. (March 2014). Microservices. http://www.martinfowler.com/articles/microservices.html #CharacteristicsOfAMicroserviceArchitecture

[4] Wiggins, A. (Jan 2012). The twelve factor app. https://12factor.net/concurrency

[5] Mouat, A. (December 2015). Using Dockers. USA: O'Reilly Media, Inc.

[6] Morris, K. (June 2016). Infrastructure as a Code. USA: O'Reilly Media, Inc.

BIOGRAPHY

Kapil Bakshi *is a Distinguished Systems Engineer for Cisco Systems, Inc. He is responsible for leading and driving Analytics, data center and cloud computing strategy and initiatives in the U.S. public sector. Kapil has extensive experience in strategizing, architecting, managing, and delivering data center solutions. During his career he has held several architectural, consulting, and managerial positions within the industry. Prior to Cisco, Kapil worked for Sun Microsystems where he spent a decade working with the U.S. Government and service provider customers. Prior to Sun Microsystems, he worked for Hewlett-Packard and several government system integrators in consulting and product development roles. Kapil is a native of Washington, D.C., and holds both a BS in electrical engineering and a BS in computer science from the University of Maryland, College Park, as well as an MS in computer engineering from Johns Hopkins University, and an MBA from the University of Maryland, College Park. He also holds U.S. patents for data center and related solution sets.*