WHITE PAPER

Running Microservices on Pivotal Cloud Foundry

By Parag Doshi, Jared Ruckle, Peter Blum, Glenio Borges, and Cornelius Mendoza

Pivotal

Table of Contents

Introduction

Go to any software conference today, and you'll undoubtedly hear about "microservices"—an approach to software architecture that's gaining momentum with enterprise developers.

Why is this idea so popular? Consider the dominant software pattern of the last two decades: the monolithic application.

These systems are a paradox—crucial to business processes today, yet a painful obstacle to tomorrow's requirements. While the business demands speed and agility, these systems are slow and cumbersome.

With microservices, developers can carve up these monoliths, one piece at a time, and re-platform them into small components. Devs are having success with this approach.

For example, Java developers have adopted new open source tooling, Spring Cloud (based on NetflixOSS) to break down monolithic architectures. And they are making progress! Many big companies proudly tout their newfound agility at industry events, and showcase how Netflix tech has helped accelerate their shift to microservices. .NET developers are using tools like Steeltoe to reap these same benefits.

So what are microservices?

Microservices refer to an architectural approach that independent teams use to prioritize the continuous delivery of single-purpose services. Let's break the key terms in this definition down:

- An ***architectural approach***...microservices are not a language; nor are they language dependent.
- ***Independent teams***...indicates that there is an emphasis on organizational structure and for Microservices adoption to be successful, there will be organizational changes required.
- ***Prioritize continuous delivery***...hints at the promise that microservices can be prioritized independently and they help in delivering software faster.
- ***Single-purpose services***... defines the scale of a given microservice. A microservice does one thing, and does it well! Microservices are about scope, not size. Smaller services (irrespective of lines of code) help you deliver faster. Further, they enable independent scalability based on business need, and user load for those services.

A microservices model is the ***opposite*** of traditional monolithic software. Monoliths consist of tightly integrated modules that ship infrequently and scale as a single unit. Although monoliths work fine for some scenarios, microservices are used by companies that need greater agility and scalability.

When combined with continuous delivery and a DevOps culture, a microservices architecture delivers a competitive advantage. This whitepaper examines how Pivotal Cloud Foundry supports microservices for enterprise development teams.

Why Microservices?

Microservices deliver three benefits: speed to market, scalability, and flexibility.

Speed to Market

Microservices are small, modular pieces of software. They are built independently. As such, development teams can deliver code to market faster. Engineers iterate on features, and incrementally deliver functionality to production via an automated continuous delivery pipeline.

Scalability

At web-scale, it's common to have hundreds or thousands of microservices running in production. Each service can be scaled independently, offering tremendous flexibility. For example, let's say you're an insurance firm.

You may scale enrollment microservices during a month-long open enrollment period. Similarly, you may scale member inquiry microservices at a different time (i.e. during the first week of the coverage year), as you anticipate higher call volumes from subscribed members. This type of scalability is very appealing, as it directly helps a business boost revenue and support a growing customer base.

Flexibility

With microservices, developers can make simple changes easily. They no longer have to wrestle with millions of lines of code.

Microservices are smaller in scale. And because microservices interact via APIs, developers can choose the right tool (programming language, data store, and so on) for improving a service.

Consider a developer updating a security authorization microservice. The dev can choose to host the authorization data in a document store. This option offers more flexibility in adding and removing authorizations than a relational database.

If another developer wants to implement an enrollment service, they can choose a relational database its backing store. New open-source options appear daily. With microservices, developers are free to use new tech as they see fit.

Each service is small, independent, and follows a contract. This means development teams can choose to rewrite any given service, without affecting the other services, or requiring a full-fledged deployment of all services.

This is incredibly valuable in an era of fast-moving requirements.

PCF <3 Microservices

Microservices inject speed, agility, and flexibility into enterprise IT. But as the saying goes "nothing comes for free." Microservices also add complexity.

When you adopt a microservices architecture, you distribute business functionality into many smaller components. Instead of building one monolithic application, you implement several dozen services, sometimes many more.

Now consider the different environments that are part of your software lifecycle management (dev, test, uat, performance test, and production). Before you know it, you have hundreds of microservices running in all your environments.

This shift can frustrate operations teams. They must provision and manage numerous virtual machines (VMs) to run these microservices. Monitoring becomes more complicated; operators must now keep tabs on these services and find a new way to provide logs to developers.

Managing this complexity with traditional processes would be unsustainable. Enter Pivotal Cloud Foundry (PCF).

When you run microservices on PCF, you reap these benefits:

1. **PCF includes many features you need to run microservices successfully**. Logging, monitoring, and scaling are done for you; they "just work" as part of the platform. Further, infrastructure management is abstracted away. Developers no longer have to worry about these primitives. The paper describes many of these attributes in the following sections.
2. **PCF supports microservices-friendly frameworks you already use**. Many teams use Spring Boot and the Spring framework for Java development. Spring Boot makes dependency management effortless, embeds a web container (like Tomcat) for all web apps, and creates self-contained apps that "just run". Consequently, apps authored with Spring Boot can run on PCF with little-to-no modification.
3. **PCF supports multi-tenant environments**. With multi-tenancy, developers get environment parity (dev, test, uat, performance and production) for all aspects of the software lifecycle. This way, they know microservices will behave similarly across environments.
4. **PCF platform is managed by** BOSH. BOSH is an "infrastructure-as-code" tool chain for release engineering, deployment, and lifecycle management of distributed systems. This gives operators a highly flexible, autonomous way to manage their PCF installation, either on-premises or in the public cloud.
5. **PCF provides an ecosystem of pluggable services that can be integrated with microservices**. This marketplace has services for application monitoring, integrated logging, as well as polyglot database and language support. Pivotal also includes a selection of first-party services for metrics, single sign-on and data services.

Pivotal Cloud Foundry & the Microservices Ecosystem

In the following sections, we'll describe how Pivotal Cloud Foundry and related components help you achieve the benefits of microservices, while minimizing complexity. Figure 1 below summarizes many of the remaining topics. Keep this diagram in mind as you read the rest of the paper!

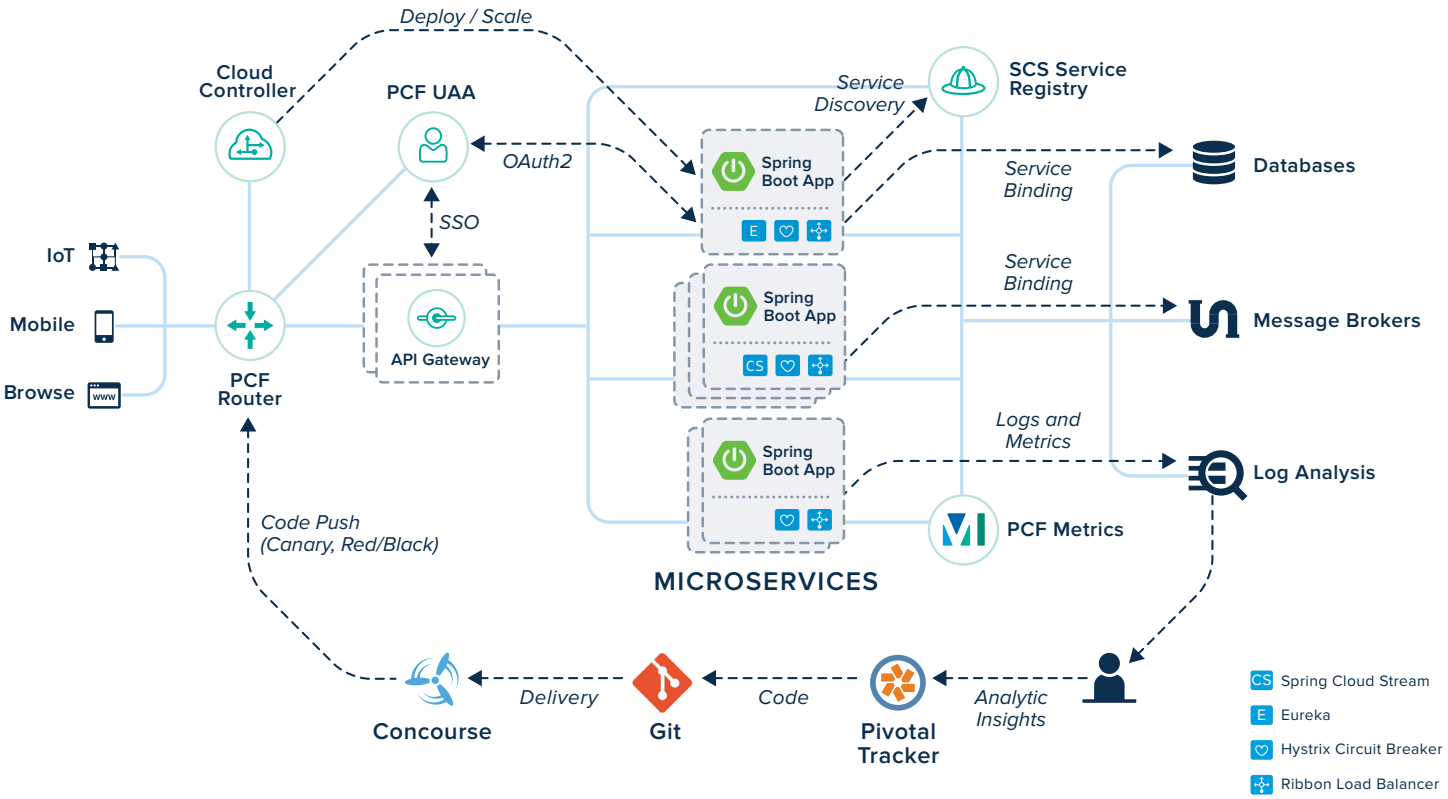

Figure 1: An overview of Pivotal Cloud Foundry and common add-on components used in microservices architectures.

Running Microservices on PCF

What really matters for microservices? And how does PCF help? We'll answer these questions (and more!) in this section.

Architecture: Use the Right Tools for the Job

As discussed above (and here in greater depth), microservices follow a different architecture than a monolith application. Microservices are similar to services based on SOAP that gained popularity in Service Orientated Architectures (SOA). However, microservices are loosely coupled, and don't follow a strict XML-based contract. Most microservices are written with a REST or Thrift API. Because these microservices use REST APIs (which run on HTTP transport), they can be written in any language.

The freedom to choose any language and any data store are one of the reasons to implement microservices. PCF supports multiple languages like Java, .NET, Ruby, Go, and Python. The platform also supports relational databases like Oracle, MySQL, and SQL Server. Similarly, developers are free to use NoSQL solutions like MongoDB and Cassandra. Engineers have the freedom to write microservices in any language, use any backing store, while still pushing them to PCF in the same way. Give your developers the power to choose the right tool for the job on a standard platform that's easy for operators!

Scalability: Speed & Automatic Routing

Each microservice, by definition, must be scaled independently. Two aspects of scalability that need to be considered: how fast can a service be scaled with all its dependencies and environment settings? And how quickly and transparently can it be discoverable for sharing the user load?

PCF handles both these requirements with aplomb. Here's how:

- **Rapid Scaling**. When a service or application is pushed in PCF, the platform creates an immutable artifact, a 'droplet'. This droplet contains all the service bits and any environment-specific dependencies that the service needs at runtime. When the service is scaled horizontally, the platform creates a container, puts the droplet in the container, and then starts the container. The service is then ready to be consumed. All this happens instantaneously.
- **Automatic Routing**. Microservices can be scaled up or down based on business needs. Because a given microservice instance is not bound to any specific IP or port, they cannot be statically bound. So service instances must be discoverable to other services or applications

  PCF does this for you—the platform manages the lifecycle of service instances. When a service instance is scaled up, PCF starts the new instances, registers them with the Gorouter, and binds them to the application route. When a service is scaled down, PCF will discard un-wanted instances, de-register them with Gorouter, and unbind them from the application route. All this is transparent to the user. PCF does all the heavy lifting of automatic routing!

Four Layers of HA: Resiliency Up and Down the Platform

Monolithic apps may only have a handful of instances running in production. When one instance crashes, it is easily noticeable. It is brought back online manually. In contrast, you will have numerous microservices running in production. Each microservice is scaled up to several instances. The chance of microservices crashing also increases. It is no longer realistic for operators to manually monitor, and subsequently bring up, failed instances.

PCF provides high resiliency by constantly monitoring the desired state and actual state of instances for all applications. If any microservice instance crashes, the actual state will not be in sync with the desired state. This discrepancy kicks off an self-healing process within PCF.

PCF will proceed to automatically create a container and place the droplet for that service. This way, PCF provides high resiliency at an instance level. PCF also mitigates total failure of all instances by spreading the instances across different availability zones. If any one availability zone goes down, then the instances in the other availability zone continue to handle requests and maintain availability. BOSH provides another layer of resiliency. It continuously monitors VM health. If any VM becomes unhealthy, BOSH will dispose of the VM, create a new VM, and redeploy all service instances that were running on the previous VM.

Security: Providing Authorization Support to Microservices

A single request to a microservice can trigger calls to multiple microservices. Most enterprises choose to secure their microservices such that there are no unauthorized calls to microservices within the system.

There are many common mechanisms that secure microservices to restrict unauthorized access. We will discuss one such mechanism within PCF: JWT with OAuth2.

JWT and OAuth2

Every microservice should be able to easily authenticate, and make authorization decisions. Many PCF customers choose JWT with OAuth2 for purpose. Here's why.

JSON Web Token is a protocol independent, JSON-based token that can be passed in HTTP authorization headers. This is a lightweight solution that doesn't affect the payload size. It's also language independent.

With this option, developers need to answer two questions: What format should be used? And how do these microservices get the JWT token?

The first question is answered by JWT specification:

> *"JSON Web Token (JWT) is a **compact, URL-safe** means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a **JSON** object that is used as the payload of a **JSON Web Signature** (JWS) structure or as the plaintext of a **JSON Web Encryption (JWE) structure**, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted."*

If the request contains a valid JWT token, then the invoked microservice proceeds to execute the request.

Next up, how do these microservices get the JWT token? OAuth2 defines the way JWT tokens are passed to microservices.

The JWT tokens are created by any OAuth2/OpenId Connect Server implementation. PCF's UAA Server is an example of one such OAuth2 provider. The token is sent in the 'Authorization' HTTP header attribute along with HTTP request. Frameworks like Spring Security make it easy for Spring Boot apps to work with OAuth2 server implementations to create and authenticate JWT tokens.

The preferred way to implement this functionality:create a separate service that's responsible for creating JWT tokens by using an OAuth2 server. All microservices can invoke this JWT token creation service, prior to making a call to other microservices.

PCF has a feature named User Provided Service (UPS) where a custom service like JWT token creation can be configured as a UPS. Any microservice that needs a JWT token can then bind to this UPS, and access the JWT token. This provides a standard way for all microservices to get a JWT token and secure themselves from unauthorized access.

Reference Architecture: Command Query Responsibility Segregation

How can you achieve scalability and reliability in practice? Let's consider a real-world example.

Figure 2 shows the popular Command Query Responsibility Segregation (CQRS) approach. This divides the system in two distinct parts. CQRS separates the components used for *writing* (executing commands) from those for *querying*. As such, the Command Service and Query Service are independently scalable services. They are decoupled, and can be operated separately. This architecture is explained in more detail in this Appendix.

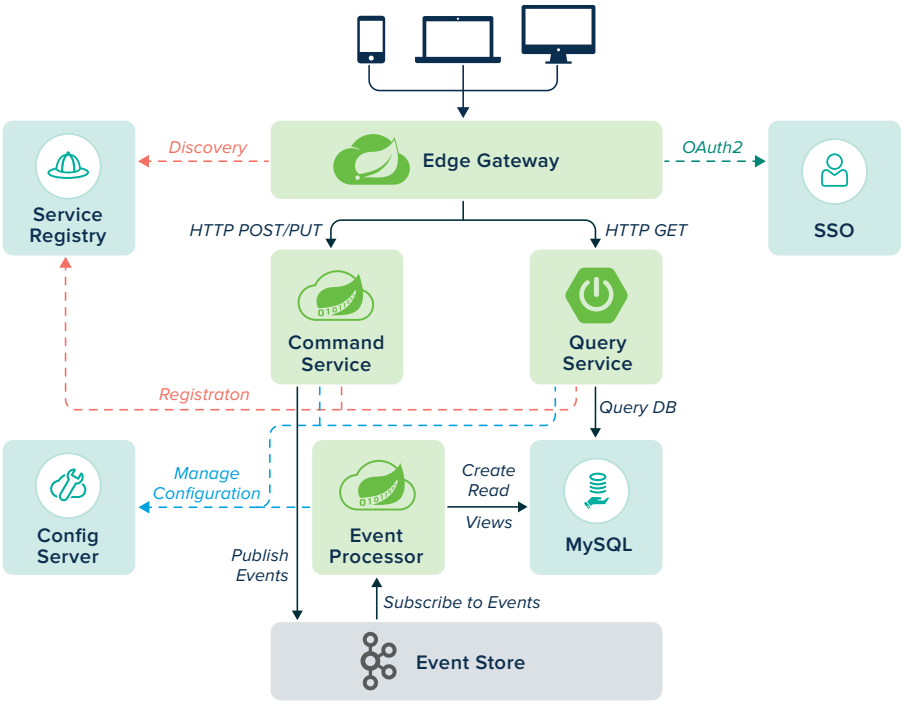

Figure 2: An event-driven, cloud-native application running Spring Cloud Services and Pivotal Cloud Foundry.

Development: Support for the Top Enterprise Frameworks

Microservices can be implemented in any programming language. Developers tend to choose the tools to help them build and integrate their microservices faster. With PCF, developers can opt for familiar languages for this new pattern. Here's how the platform supports two popular enterprise frameworks, Java and .NET.

Java: It's All About Spring Boot

Spring is the most popular framework for Java developers; Spring Boot takes it one step further.

Spring Boot makes it easy to create stand-alone, production-grade Spring apps that "just run." It takes an opinionated view of the Spring platform and third-party libraries. Consequently, developers can launch apps with minimum fuss. Spring Boot offers Java developers many other benefits:

- **Spring Boot makes dependency management effortless**. Spring Boot introduces spring-boot-starter-* dependencies that bring in all the required jar files for an app during compile/runtime.
- **Spring Boot auto-configures commonly registered beans (like DispatcherServlet and ResourceHandlers).** Developers get sensible defaults, so they don't have to spend time wiring the same settings time and time again.
- **Spring Boot embeds a web container (like Tomcat) with all web apps**. Developers don't have to deploy their app to external web containers. This step is done for them.
- **Spring Boot quickly creates self-contained apps**. Your app is ready to process web and RESTful requests immediately.

Spring Boot apps "just run" on PCF for a few reasons. First, the app is self-contained with all dependencies packaged in an executable jar file. Second, PCF adds environment properties, and starts the app in a container within seconds. Finally, a large number of deployment issues are eliminated, since Tomcat, Jetty, or Undertow is embedded.

This is why Spring adopters choose PCF for their microservices.

There are other benefits once your app is live.

PCF's support for Spring Boot Actuator helps developers monitor and manage their applications in production (Figure 3 and Figure 4). Aggregated logging, dynamic log level changes, and pluggable services allow developers to quickly integrate and test their services with minimal code changes.

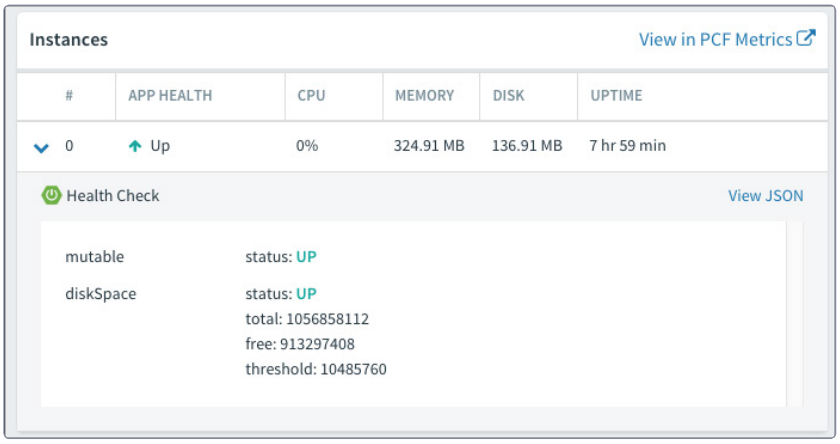

Figure 3: Actuator health endpoint details integrated in App Manager UI.

Figure 4: Dynamically configure Log levels at package or individual class level without app re-start.

Spring Cloud Services

When developers bind their apps to Spring Cloud Services, they gain out-of-the-box support for Netflix OSS components like Eureka (Service Registry), Hystrix (Circuit Breaker) and Config Server (for externally managed configuration). All three are best-in-class tools for microservices patterns. Refer to the whitepaper "Standing on the Shoulders of Giants: Supercharging Your Microservices With NetflixOSS and Spring Cloud" for a deeper look.

Cloud-Native Data with Spring Cloud Data Flow

Use PCF and Spring Cloud Data Flow (SCDF) for data microservices. With SCDF, you can build data integrations and real-time data processing pipelines.

Pipelines consist of Spring Boot apps built with the Spring Cloud Stream or Spring Cloud Task microservice frameworks, as shown in Figure 5. The diagram below features RabbitMQ as the messaging broker; the engine itself is broker agnostic and supports other systems such as Kafka.

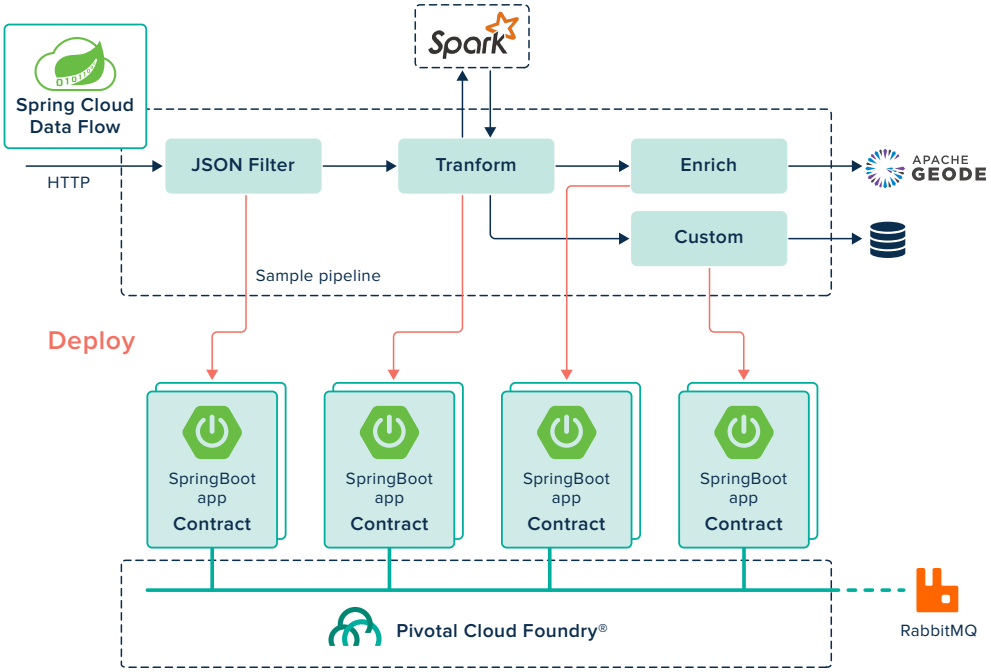

Figure 5: A common data microservices pipeline with Spring Cloud Data Flow.

All of this makes Spring Cloud Data Flow suitable for a range of data processing use cases: import/export, event streaming, and predictive analytics. The Spring Cloud Data Flow server uses Spring Cloud Deployer to deploy pipelines onto modern runtimes such as Pivotal Cloud Foundry.

Spring Cloud Data Flow also offers a collection of patterns and best practices for data microservices running as streaming and batch data pipelines in PCF. PCF allows developers to create, unit-test, troubleshoot and manage data microservices in isolation. So, developers can build data microservices using DSL, Shell, REST-APIs, Dashboard, and Flo. From there, they can perform Blue/Green deployments on data microservices, just like any other app running on PCF.

And since Spring Cloud Data Flow microservices are implemented as Spring Boot apps, they enjoy all the benefits of PCF (metrics, logging, health checks, and remote management) at the data microservice level!

.NET Apps: Now a First-Class Citizen in PCF

PCF is popular with .NET developers for the same reason it's popular with Java teams:

> *PCF helps developers rapidly deliver new experiences to customers by building and deploying new applications quickly. Engineers can focus on their code, while deferring much of the operational complexity to the platform.*

PCF allows .NET developers to push applications or microservices with a simple **`cf push`** command.

PCF Supports Full-Featured .NET and .NET Core

The future of .NET apps is becoming clear.

Most of your existing .NET web apps today are written in C#, and run on IIS, powered by Windows Server. Meanwhile, Microsoft is encouraging developers to build modern microservices with .NET Core.

The good news: PCF supports microservices written in .NET Framework (leveraging the Hosted Web Core buildpack) and .NET Core! This offers maximum flexibility and utility for .NET teams.

Pivotal recommends creating new microservices with the .NET Core framework, and targeting either the .NET Core runtime or the .NET Framework for the service. Follow this guidance, and you can deploy to either a Linux or Windows environment.

Either way, when you push .NET apps to PCF, they enjoy all the platform benefits discussed in this paper. And with support for Steeltoe, .NET microservices integrate with Spring Cloud Services. Developers can bind their apps with three Netflix OSS services – Eureka, Hystrix and Config Server.

Accelerate CI/CD for .NET Teams with PCF-Visual Studio Integration

Microsoft and Pivotal have teamed up to build a slick integration between Visual Studio Team Services / Team Foundation Server and PCF. Use Microsoft's Cloud Foundry plugin to integrate existing CI/CD pipelines with PCF. Blue/Green deployments via CI/CD pipelines can be developed in minutes. And they can be tested without additional servers or network support!

Manage Windows Servers at Scale with PCF Runtime for Windows

PCF also supports BOSH-managed Windows Server. The feature—PCF Runtime for Windows offers an extra level of resiliency for .NET microservices. What's more, Windows admins gain a new way to run their server fleets according to "immutable infrastructure" principles.

Monitoring/APM Tools: A New Approach for a New Age

Application performance monitoring (APM) tools are traditionally agent-based systems.

Language-specific monitoring agents are installed on web and application servers. When the apps run on those servers, the agents instrument the code, and send performance data back to the monitoring servers. Monitoring clients and web-based dashboards are then used to diagnose performance issues and usage trends. This concept is manageable with handful of monolith apps deployed in known servers (aka "pets"). Often, a team of monitoring specialists would own the installation and management of these agents. This team worked with system admins to install the agents on each server manually. Monitoring becomes the responsibility of this specialist team and developers get further removed from configuring and monitoring their applications.

Why won't this work in the world of microservices?

In a microservices architecture, there are numerous service instances running in their own containers on different VMs. Also, technologies like Spring Boot bundle the web server along with the application jar file—so there's no need for external web servers. In this environment, pre-installing the agent won't work.

PCF solves this problem by bundling the APM agent binaries as part of the application's buildpack. The agent (or runtime binaries) become part of the droplet. When the application starts up—and environment

variables are configured with APM properties—PCF will run the agent processes and send performance data to the APM server. In this way, PCF takes care of installing the agent binaries transparently. This also remove the manual agent installation task and the need for a specialist team to manage the agents.

PCF supports APM tools like Dynatrace and New Relic. APM tools may be installed on-premises or consumed in a SaaS model.

Logging: Easy Real-Time Streaming to a Centralized Location

Traditional logging depends on operators manually accessing servers one by one, and subsequently providing log files to developers. This cumbersome process prolongs issue triage and troubleshooting. With microservices, this problem is magnified, because there are numerous service instances running in their own containers, invoking other microservices.

Once again, PCF offers a solution for microservices adopters.

PCF aggregates the logs, and streams them for easy access. Developers get web server, platform, and application logs in a single place. This helps engineers correlate logs and events. As a result, application issues can be solved faster.

PCF also supports external Log Management Systems (LMS) like Splunk. Operators can configure firehose nozzles to send application and platform logs directly to a LMS. Developers and operators can then filter, search, and run analysis on these logs using their preferred tools. Refer to the Metrics section below for more details.

Distributed Tracing: Understand the Interactions Between Your Microservices

When monoliths run into performance issues in production, it is easy to conduct an investigation. All logic is encapsulated in that one app.

When you move to microservices, performance issues can occur at any of the nodes of communication. A single call to a microservice can lead to multiple calls to different microservices, which in turn could invoke other microservices. Before long, you're dealing with a web of interconnected calls. This make triaging issues very hard.

How do you solve this problem? Distributed tracing to the rescue!

Distributed tracing works by tagging every request (a 'trace') and sending data for each operation/service ('span') the request invokes during its processing. For any given request, a trace is created that also contains one or more spans.

Spring Cloud Sleuth provides distributed tracing for Spring Boot apps. Spring Cloud Sleuth adds an abstraction layer on top of distributed tracing models like HTrace and Zipkin (Dapper). Spring Boot apps will add trace information automatically if Spring Cloud Sleuth is added as a dependency in your project classpath. If spring-cloud-sleuth-zipkin is available as a Maven dependency, then the app will generate Zipkin compatible traces via HTTP. PCF will also stream the trace information in the loggregator and PCF Metrics for these apps.

PCF Metrics provides its own implementation of Spring Cloud Sleuth, shown in Figure 6. It features a UI and search capabilities for investigating latency and performance issues. Trace and span ids are streamed as part of the logs. Users can visualize the flow of requests among different microservices.

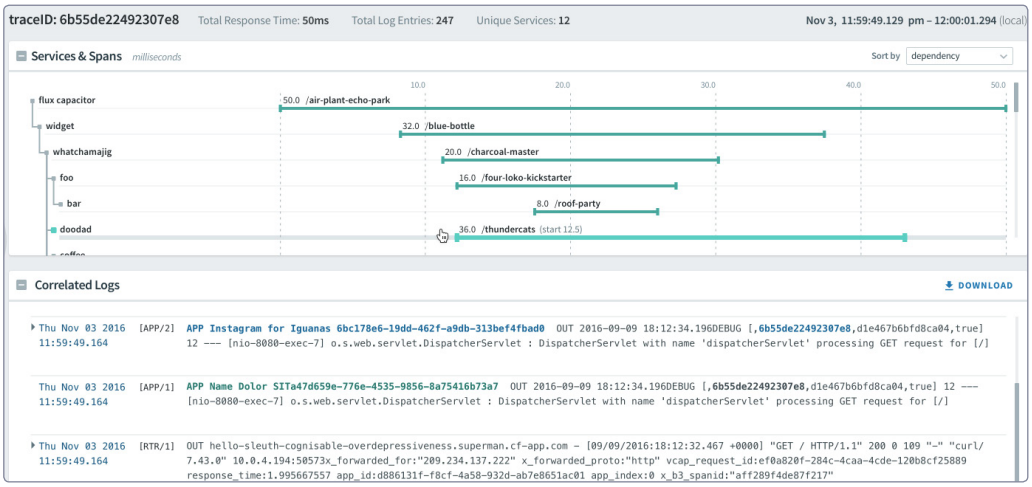

Figure 6: Distributed Tracing with trace and span info in PCF Metrics.

Operations

As you build microservices, think about how you push and monitor your microservices in production. Let's review how PCF helps you in three operational areas—CI/CD, Blue/Green Deploys and Metrics.

CI/CD

Continuous Integration and Continuous Delivery pipelines are prerequisites for microservices. Use automation to release incremental to production frequently (daily if possible!). CI and CD helps you visualize the flow of your code through various environments and test cycles.

PCF supports CI/CD in two ways:

First, PCF uses multi-tenancy. Orgs and Spaces map to your software development lifecycle environments. PCF provides environment parity as your code moves through the different Spaces. This gives the developer peace of mind: code will behave the same way in different environments.

Why does this matter? As you push incremental code changes to different environments (dev, test, uat, performance), the new changes are validated through an automated test harness in each environment. As your code moves to higher environments, the focus of your tests will change from unit test to integration tests to performance tests. Without guarantees on consistency, the whole process falls apart.

Second, PCF integrates with popular CI/CD tools like Jenkins and Concourse. PCF provides a set of APIs that can be invoked from these CI/CD packages. Developers quickly script a CI/CD pipeline by invoking the Cloud Foundry API. Then, they can integrate it with their favorite IDE, or run it as part of their automated build process.

For a detailed view of CI and CD—and how to get started with it—review this companion whitepaper.

Blue-Green Deployments (Zero Downtime Deployments)

Teams choose microservices architectures to accelerate time to market. Pushing incremental changes to production results in rapid user feedback, which in turn yields more incremental changes that improve the user experience. This is a virtuous cycle!

There are cases when you may want to rollout a new feature based on geographical location, customer type, or other criteria. Ideally, you make the changes without affecting existing functionality. Blue-Green deployments are the pattern to handle this scenario.

But what happens if your users don't like a particular feature? Blue-Green deployments help there too!

To conduct a phased rollout to a new version (or rollback to a previous version), you need a way to divert user traffic easily and dynamically. PCF supports this with 'routes'.

A route is a unique URL associated with an application. An application can have one or more routes, and a route can be associated with one or more apps. If you want to rollout a new version of an app, you may create a temporary route to validate it with a smaller subset of users. After the validation is successful, and you want to do a complete rollout, you can map the new version of the app with the same route as the previous version. You can then slowly phase out the previous version of the app by un-mapping the route. Figure 7 illustrates this workflow.

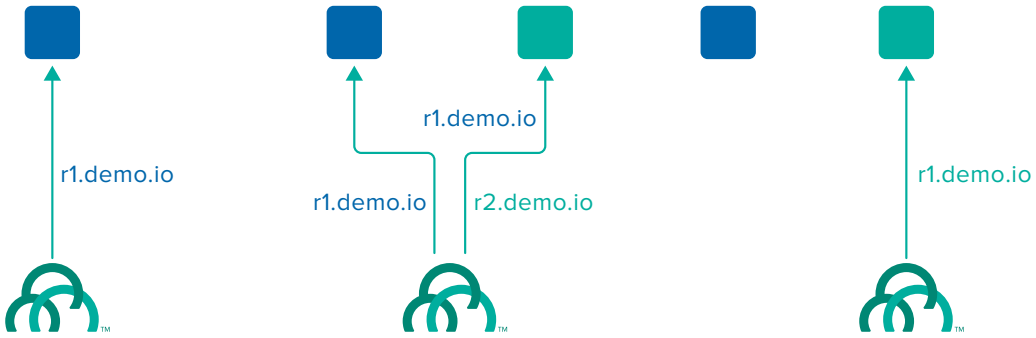

Figure 7: A Blue-Green deployment with Pivotal Cloud Foundry.

All this can be done dynamically, while users interact with the previous version of the app. In case of rolling back updates, you can easily un-map the route from the new version such that no one can access the new version of app. You can then remap the previous version of the app.

Blue-Green deployments reduce the risk associated with new changes in production. This is a useful technique to employ when making small changes to numerous microservices. You can elicit faster feedback from your customers, without downtime!

And remember: Blue-Green deployments are most effective with stateless microservices. Life becomes more difficult if you have microservices talking to backend databases.

Metrics

Operators are interested in the health of all three layers of their cloud-native apps: microservices, containers, and VMs. PCF boosts operational efficiency by showing platform and application metrics in a single place.

PCF's Loggregator is a logging system that collects logs and metrics from apps and platform components. It streams them to a single endpoint, the Firehose. A nozzle is a component dedicated to reading and processing data that streams from the Firehose. Customers use custom nozzles for popular tools like Splunk, LogSearch and DataDog that help customers view and analyze metrics in their preferred systems.

For developers, the aforementioned PCF Metrics module displays application logs and metrics on a dashboard. This add-on provides a graphical representation of application data for a given window of time, along with the corresponding application logs.

Conclusion

Companies committed to becoming a software-defined business are opting for microservices. This pattern delivers speed to market, flexibility, scalability, and developer efficiency.

Many traditional tools and processes—geared towards monolithic applications—are no longer applicable to microservices. Thankfully, platforms like Pivotal Cloud Foundry resolve much of the complexity associated with microservices.

A mature, distributed microservices architecture demands support for popular frameworks like Spring and .NET, continuous integration and continuous delivery tooling, and easy access to logs and metrics.

Pivotal Cloud Foundry is a microservices platform that supports all these requirements and much more. Further, many enterprise organizations are building and operating new microservices architectures atop PCF, and having success. Recommended Reading and Appendix A offer a deeper look at microservices in Pivotal Cloud Foundry and customer success stories. Appendix B offers a look at cultural considerations for microservices, and Appendix C describe useful reference architectures for this pattern.

Recommended Reading

- Whitepaper: Standing on the Shoulders of Giants: Supercharging Your Microservices With NetflixOSS and Spring Cloud
- Topic Page: Microservices
- Deploying Microservice Architectures with Pivotal Cloud Foundry A Pivotal Perspective
- Speed Thrills: How to Harness the Power of CI/CD for Your Development Team
- Whitepaper: Secure Hybrid Banking Reference Architectures for Cloud-Native Applications
- E-Book: Cloud Native Java

Appendix A: Learning from Your Peers—How the Best Companies Use Microservices

Because Pivotal works side-by-side with customers, we think it's important to share best-practices knowledge and lessons learned from real-world, large customer projects and hope you find these case studies informative.

Citi

Brad Miller, Head of Global Digital and Cloud Technology at Citi, says, [Pivotal helped] develop the skills that we needed to start building microservices and thinking about transitioning our existing architecture. [Learn More]

Comcast

Thousand of applications and services, built by hundreds of developers, supporting tens of millions of transactions—Pivotal solutions help Comcast go from idea to feature in days not weeks, driving competitive differentiation. [Learn More]

Allstate

Committed to using technology to drive business forward, Allstate engaged with Pivotal to innovate and scale services at a speed and level of effectiveness that would have once been unimaginable. [Learn More]

Appendix B: Microservices and Your Culture

Organizational: Get Ready for Microservices

In monolithic application architecture, different teams own the phases of software development life cycle. There would be a development, functional testing, quality assurance and performance testing teams. After the application is deployed to production, the application is handed over to the production support teams. The teams become siloed and the organization relies heavily on processes over joint ownership. Ownership is fragmented and every team has ownership of only one aspect of the software development life cycle. No one has end-to-end responsibilities of the software product.

Organizations that adopt microservices architecture have to change this model. If they want to gain benefits from higher developer productivity and faster speed to market, they have to remove these process bottlenecks and empower the teams to own the entire software development life cycle. This requires changes in culture, team structure and roles within these teams.

Culture: Aligning Dev and Ops Around Common Goals

Organizational culture is a system of shared assumptions, values, and beliefs, which governs how people behave in organizations. The goal is to have uniform culture across departments and teams. But that seldom happens.

Organizations that want to adopting microservices architecture need to tackle a common this difference in culture cultural conflict. Ask yourself these two questions:

1. Are your developers rewarded for higher velocity and faster delivery of value to customers?
2. Are your operators rewarded for maintaining stability and adherence to processes?

If you answered yes to both, you may need to tie these goals to a common goal.

Organizational leaders need a transformative approach, and bring the two teams together and tie them to a common goal. The teams should have shared responsibility and shared understanding of success.

The developers need to ensure that the code changes that they deploy to production are highly reliable, regression tested and fault tolerant. One piece of code should not bring the entire system down!.

Operators should also have good monitoring, metrics and logging tools in place. They should be able to proactively identify issues and work with developers to get it resolved before customers are affected. Developers and Operators should pair together to create automated pipeline for code deployment so that both teams have visibility on how the code moves through different software development lifecycle phases. It is a shared responsibility for both teams and should not be handed over to a separate ‘DevOps’ team.

Team Structure/Composition

In a monolithic architecture, organizations have big teams that work on different aspects of the monolith. There are different development teams, tester teams and production support teams. The teams are tightly coupled, not only in code, but also how they move to production.

The level of dependency amongst the team is very high and no single team can innovate or move faster than any other team. Deploying to production is a big ceremony where all the teams come together one weekend and try to push all their code to production at the same time. If code from one team breaks, than the entire production deployment comes to a standstill.

This team structure will not work for microservices architecture as the emphasis is on independent, smaller teams delivering changes to production faster.

Organizations that adopt microservices need to re-organize their teams around product features or bounded contexts. Each team will own a small piece/feature of the entire software. For example, for an e-commerce website, there would be a Search feature team, Shopping Cart feature team, Checkout feature team and so on.

These feature teams will own the entire development and maintenance of the product feature they own and adhere to the RESTful contracts with other teams. The teams will have developers, business analysts, and functional testers. Production support would be rotated amongst the developers in the same team. Teams are loosely coupled, just like the software they build!

Smaller, independent, multi-disciplinary, self-sustaining teams are needed to make microservices software development successful.

Roles

In monolithic architecture, the organization roles are clearly defined—developers will develop code, testers will test code and maintenance support team will support the code in production. When organizations move to microservices architecture, the focus is towards independent self-contained teams. All team members wear multiple hats and can fill in the role of other team members, if needed. The same developers would perform development and production support role.

In monolithic architecture, there is a separate integrator role. The developers in this role are part of a central integration team who are tasked to integrate the different monolithic software components together, usually around a enterprise service bus product. In a microservices architecture, the integration is done through RESTful contracts. The developers in the microservices team are responsible for their integration with other teams. There is no need for a special integration team in this organization design.

The same goes for ‘DevOps’ teams that are usually created to bridge the gap between developers and operators. These team members are configuration specialists that are responsible for taking the code from development environments to production environment. Again, this is another centralized team that is created specifically to manage environment configurations for monolithic applications. In a microservices based environment, the developers and operators work together to create their automated pipeline and manage the different environment configurations as part of their pipeline. Configurations are ideally managed independently in git-backed config servers. In this manner, the specialist team roles that exist in monolithic team structure are no longer required in microservices based team structures.

Appendix C: Reference Architectures

Applications & Microservices Running on Pivotal Cloud Foundry

The architecture in Figure 8 features common cloud-native attributes.

- A modern approach to real-time data processing
- A highly distributed, scale-out pattern
- The Command Query Responsibility Segregation (CQRS) approach. This divides the system in two distinct parts. CQRS separates the components used for writing (executing commands) from those for querying. As such, the **Command Service** and **Query Service** are independently scalable services. They are decoupled, and can be operated separately.
- An event store for processing a high-volume of transactions

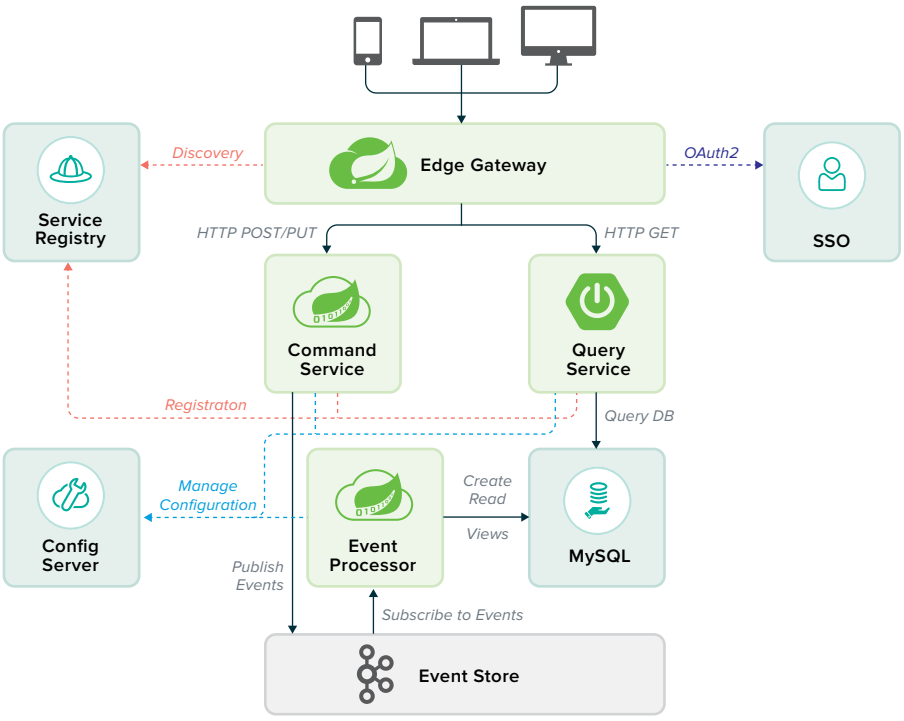

Figure 8: An event-driven, cloud-native application running Spring Cloud Services and Pivotal Cloud Foundry.

Reference Architecture Components

- **Edge Gateway, implemented with Spring Cloud's Zuul, is a reverse proxy**. This gateway service provides dynamic routing, monitoring, resiliency, security, and more. In this architecture, it executes specified dynamic routing rules, and accommodates diverse inbound clients.

- **Service Registry, implemented as Spring Cloud Services Service Registry, tracks the services.** The Command Service and the Query Service both register with this module; the Edge Gateway uses the Service Registry to discover service locations and where requests should be routed.

- **SSO (Single Sign-On) for PCF**. The Single Sign-On service is an all-in-one solution for securing access to the application and its services. It improves security and productivity since users do not have to log in to individual components.

- **Command Service, written with Spring Cloud Data Flow, orders the system to do something or change its state**. This represents part of the business logic of the application.

- **Query Service, a Spring Boot app, searches the relational database and returns relevant results**. In contrast to the Command Service, this does not change the state of the system. This service is likely to experience variable load, and can be scaled accordingly.

- **Config Server, implemented as Spring Cloud Services Config Server, stores all environmental variables**. No configuration is stored in any of the services. The Config Server stores this information and provides it on-demand. This is a cloud-native best practice, in keeping with "12 factor app" principles.

- **Event Processor, written in Spring Cloud Data Flow, works with the Command Service**. This is the other part of the application's business logic.

- **MySQL for PCF serves as the "system of truth."** It returns results requested by the Query Service.

- **Event Store, based on Apache Kafka, is the high-volume event processing system.** This pattern is often referred to as "Event Sourcing". It is a superior option for this scenario, since event streams in the past may need to be "replayed." In contrast, systems like RabbitMQ immediately delete messages upon successful receipt.

Sample Customer Banking App

Now, let's consider a Sample Application. This application architecture was the result of a deep collaboration and co-engineering between Pivotal and a strategic banking customer, JPMorgan Chase (Figure 9).

This application is an enterprise data platform. It acts as data source for all other services to get reference data.

We can see how the platform enables the developers to focus on their custom code. PCF and Spring Cloud Services take care of ongoing management and operation.

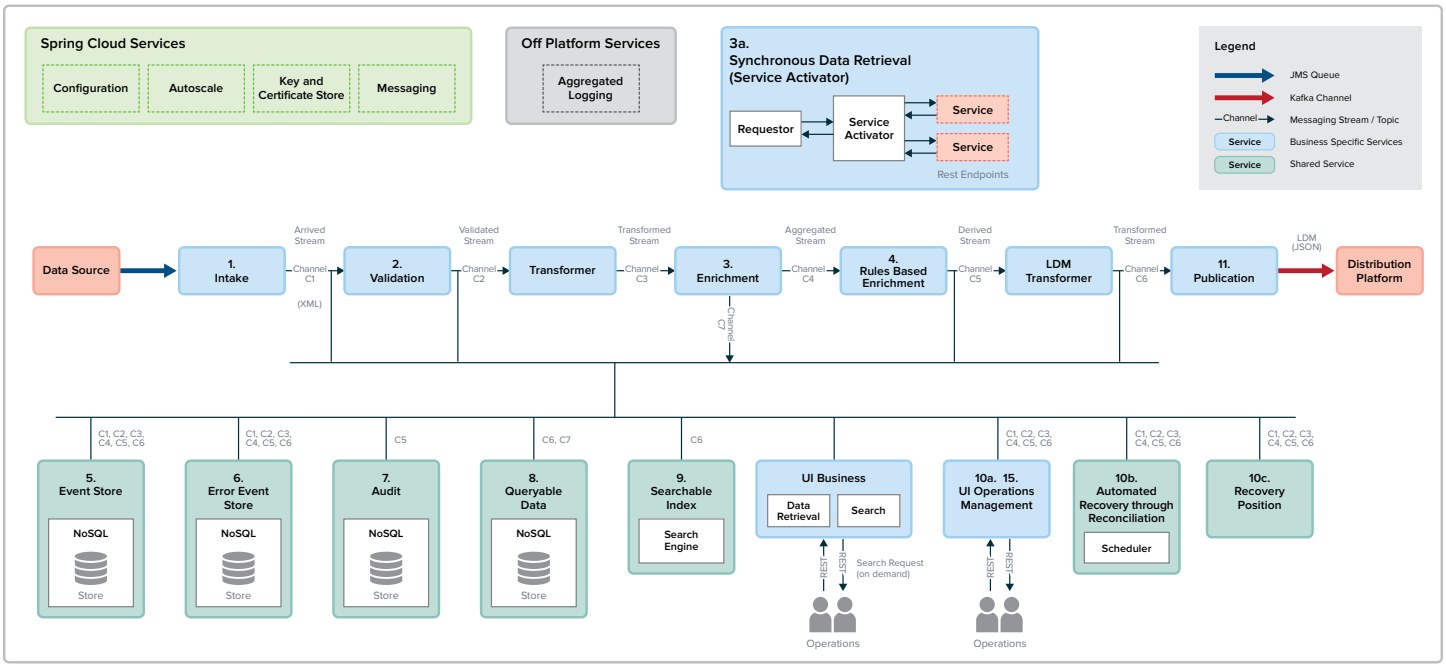

Figure 9: Reference Architecture for Data Ingestion based on Spring Cloud Stream.
An production banking application architecture from a Pivotal customer.

Sample App Components

This co-architected application features these custom services.

1. **Intake**. These services collect data from upstream sources. Data is then registered within some form of store. Data is subsequently transformed into the shape expected by the downstream pipeline.

2. **Validation**. Here, the data is validated according to its adherence to the required schema and quality controls.

3. **Enrichment**. Enrichment of the data message is dependent on data sourced from external resources. The external sources are accessed through an asynchronous or synchronous interface. The patterns used include:

    a. *Synchronous Data Retrieval*. The reference to a number of external services (that will typically be synchronous in nature and not dependent upon each other) are referenced through the service activator pattern. For performance benefits, this will be a composite of the splitter—aggregator pattern.

    b. *Asynchronous Data Retrieval*. Data that is required from multiple sources must be retrieved through an asynchronous / messaging system. This system adopts the reactive approach.

    c. *Gateway*. If requests are made to resources “outside the boundary of control”, then they engage through the gateway.

4. **Rules-Based Enrichment**. A rules engine, with custom rule definitions, is needed to apply complex business rules (such as derivation and validation). Typically, this is a RETE-based engine.

5. **Event Store**. All notable events that occur within the system are captured and recorded for later processing. These events are also discoverable.

6. **Error Event Store**. Similar to the Event Store described above, but for error events that occur within the system. This requires that each message be tagged with its recovery point. Usually, the recovery point is the channel from where the message can be re-submitted into the flow. This also includes the data payload that can be re-sent through the system.

7. **Audit**. All changes that occur (within the flow) to the original data message are recorded. Additional metadata (such as when the change occurred and the cause of the change) is also recorded, subject to audit requirements.

8. **Queryable Data**. The “quality” data of the system will be queryable and therefore retrievable.

9. **Searchable Index**. All “quality” data (i.e. enriched data) within the system is searchable through an efficient search service. Typically, the service is provided by a search engine backed by a NoSQL data store.

10. **Recovery**. The recovery point is the output channel for the last known successful component. This requires that a record of the successful processing within the system be applied to the message.

    a. *Manual Recovery*. Some error events require inspection and manual intervention for recovery. All information is based on the error condition at time of fault. The last successful point in the system is made available.

    b. Automated Recovery through Reconciliation. Failed (but recoverable) messages are automatically placed onto the last known channel. A record of the number of attempts that a message has been played is also maintained within the message's metadata. This allows the reconciliation process to be stateless.

    c. *Recovery Position*. This is any point within the flow where a message can be replayed from.

11. **Publication**. Data that has reached the quality threshold, as determined by the system, is published to the downstream consumer. This is typically a data reservoir or equivalent distribution platform.

12. **Circuit Breaker**. This allows for the isolation of external dependencies within the system. When such dependencies are unavailable or broken, the system is fault tolerant, and offers a path to recoverability, when handling said dependencies.

13. **Edge Gateway**. Façade for the integration of multiple services or data sources. This allow for centralized security (authentication and authorisation) management through tokens and entitlements.

14. **Operations Management**. Controls the operation of the platform. This includes both monitoring operations and responding to events, including errors and security events.