Standing on the Shoulders of Giants

Supercharging Your Microservices with NetflixOSS and Spring Cloud

By Richard Seroter

Pivotal

WHITE PAPER

Table of Contents

Standing on the Shoulders of Giants

Supercharging Your Microservices with NetflixOSS and Spring Cloud

BACKGROUND

Back in 1869 when Washington Roebling started work on the Brooklyn Bridge, there wasn't a tradition of collaboration among fellow bridgebuilders. Engineers were encouraged to mind their own business and rarely offered up insight that would be deemed "valuable" to rivals. This meant that information about a mysterious illness crippling workers building a bridge in St. Louis was not socialized with the professional community. If Roebling had access to this information, or the findings from Europe about this affliction later labeled "the bends," the lives of many men would have been spared.

What does this have to do with microservices? As it turns out, a lot. We're now in a period of unprecedented openness and collaboration among technical colleagues. Instead of building systems made up of proprietary technology and closely-guarded local knowledge, companies are constructing modern microservices with the help of open source software and patterns evangelized by forward-thinking members of the community. This sharing of assets helps developers build more reliable software, faster. And almost no one shares their software assets quite like Netflix.

Netflix is a media giant, with over 80 million global customers who watch more than 125 million hours of streaming content everyday. During primetime hours, Netflix consumption constitutes 35% of all Internet traffic in North America.[1] In 2008, Netflix started their journey to the cloud and microservices for three reasons: availability, scale, and speed.[2] As a 24x7 service, Netflix needs to be "always on" and their monolithic code base made troubleshooting problems difficult. Back then, a single missing semicolon took the entire Netflix site down for hours! The company also faced scaling challenges that were hard to address within a single monolithic application. As Netflix added more types of user interfaces to their service and expanded into more geographies, they couldn't easily scale individual parts of their application. It was all or nothing. Finally, Netflix needed to optimize for speed. In a competitive market, companies like Netflix have to help teams deliver more software, faster, and that's not easily achievable when everyone steps all over themselves in a single monolithic system.

1 Netflix, Q2 2016 Letter to Shareholders, July 18, 2016, p. 6, http://files.shareholder.com/downloads/NFLX/2849147477x0x-900152/4D4F0167-4BE2-4DC1-ACC7-759F1561CD59/Q216LettertoShareholders_FINAL_w_Tables.pdf

2 Mooney, Gregory. "Why You Can't Talk About Microservices Without Mentioning Netflix," SmartBear Software (blog), SmartBear Software, Dec. 8, 2015, http://blog.smartbear.com/microservices/why-you-cant-talk-about-microservices-without-mentioning-netflix.

Netflix is a software-driven business, so their engineers constructed an impressive array of software that builds upon foundational components from Amazon Web Services, their chosen cloud provider. As an early adopter of microservices and highly-resilient cloud systems, Netflix solved a number of these complex problems that most companies are just starting to face. They've shared these hard-fought lessons in the form of open source software that anyone can use to build cloud-scale systems.

In this paper, we'll take a look at the architectural challenges Netflix faced, what they've released as open source software, why this software matters to you, and how Pivotal makes it easy to consume Netflix software as part of Spring Cloud.

MICROSERVICES: OPPORTUNITIES AND CHALLENGES

With a microservices architecture, you have many individual components, each associated with a specific responsibility, loosely coupled to each other, continuously delivered by independent teams through automation. Let's unpack that sentence, and compare these attributes to those of a monolithic software solution.

… many individual components …

Traditional business systems are often monolithic in nature. All of the business logic, user experience, and integration interfaces are part of a single code-base or software installation. It's often fairly self-contained, but not modular in a way that supports component upgrades. Conversely, a microservices architecture has lots of little pieces that may be dedicated to a given application, or shared by many systems. "Micro" is meant to refer to the scope of the service, so don't get caught up in "physical size" or "lines-of-code" as meaningful attributes.

… each associated with a specific responsibility …

In his O'Reilly book titled Migrating to Cloud-Native Application Architectures, Matt Stine says that it's about creating:

> *"… independently deployable services that do 'one thing well.' That one thing usually represents a business capability, or the smallest 'atomic' unit of a service that delivers business value."*[3]

A hallmark of microservices is building discrete services that don't encroach on the domain of others. As a reference, the principles of "domain driven design"[4] help teams successfully define their boundaries. Because services should "own" whatever they need to deliver a given business capability, experts recommend that microservices have their own data store. This concept requires a major shift in how we've traditionally design software, but offers a compelling approach to those have spent years building or maintaining dense, tangled systems.

3 Stine, Matt. Migrating to Cloud-Native Application Architectures. O'Reilly Media, 2015.
https://pivotal.io/platform/migrating-to-cloud-native-application-architectures-ebook

4 Avram, Abel, and Marinescu, Floyd. *Domain Driven Design Quickly*. InfoQ, 2006.
https://www.infoq.com/minibooks/domain-driven-design-quickly

… loosely coupled to each other …

One of the major challenges in a monolithic business system is the interconnectedness of all the components. Are you changing the database structure, swapping out a web server, or altering some business logic? Often this requires an update to numerous components because of the uncomfortable closeness of all the pieces. As a result, teams shy away from making (necessary) updates because of the many ways that things could go wrong.

Conversely, microservices encourage loose coupling. Services interact with each other through contracts or messaging buses, and have zero knowledge about the implementation details. The owner of a particular microservice can responsibly iterate on the service without fear of shattering a fragile relationship with its ecosystem. Thus, encouraging teams to make frequent changes.

… continuously delivered by independent teams ….

If one believes in Conway's Law[5]—the observation that systems reflect the organizational communication paths of those who build them—then a transition to microservices must be accompanied by a corresponding change in the team. The same project teams and line-of-business hierarchies that created a data center full of monolithic business systems cannot successfully create a microservices architecture.

Instead, microservices often bring about a major change in team structure and tooling. If the goal is to continuously deliver business logic in the form of microservices, then software teams need the autonomy and directive to build and deploy focused services all on their own. This means that short-lived project teams give way to long-lived, independent product teams. These teams don't file tickets to do deployments; they directly push to production early and often. The team makeup consists of all the skills needed to design, build, and deploy the service. Such teams have the personnel, permission, and technology needed to iterate constantly and get the results of each iteration into production immediately.

… through automation.

Pivotal's Casey West says that microservices are about "automating the path to production."[6] It's virtually impossible to ship software constantly if the delivery pipeline is manual, as it has historically been with monolithic software. Automation is a core part of the microservices experience, enabled via continuous integration, continuous delivery, and continuous deployment. **Continuous integration** refers to the practice of developers merging all their code multiple times per day. Instead of slogging through the "integration testing" phase of a project where countless unforeseen mismatches occur between code modules, continuous integration ensures that issues bubble up faster. There are a lot of logistics required to constantly execute tests, so modern teams rely on event-driven automation to retrieve source code, stand up infrastructure, and execute tests.

5 Brooks, Fred. *The Mythical Man-Month*. Addison-Wesley. 1975.

6 West, Casey. "The Five Stages of Cloud Native." Presentation at SpringOne Platform, Las Vegas, Nevada, August, 2016. Accessed September 2016.

If all the tests pass, code could be considered production ready. **Continuous delivery** happens when code is packaged and capable of being deployed to production. Capable being the key word here. There could be many reasons why the pipeline is completely automated, but code is purposely gated before actually getting deployed, usually related to business timing. **Continuous deployment** goes one step further and involves every change automatically moving to a production environment.

All of these techniques require a careful assessment of current deployment procedures, and a ruthless focus on eliminating waste and automating manual steps. The result? Repeatable, timely deployments that greatly reduce the amount of time needed to get valuable software into the hands of your customers.

CHALLENGES WITH A MICROSERVICES ARCHITECTURE

As one can imagine, the picture painted above opens up a host of challenges not typically faced when deploying monolithic software. Modern teams now need answers for ...

- **Where are my services located?** As mentioned above, a microservices approach typically results in more services in your environment. Because microservices are constantly being created and updated, the old way of hardcoding references to a service address in your code or load balancer doesn't work any longer!
- **How do I ensure consistent application configuration, at scale?** Automation drives deployment of microservices, and the servers (or containers) that host them may come and go. It's not feasible to log into all the microservices hosts and update code or configuration; teams have to figure out how to make sure that each instance of a microservice looks identical at all times.
- **Can I perform uniform deployments across environments?** A hallmark of modern microservices is the use of cloud technology at runtime. Software teams take advantage of the APIs and geographic reach offered by public clouds to ship services to the places that benefit their customers the most. But, is it really feasible to build unique deployment pipelines for private data centers, and a variety of public clouds? Is there a way to provide a unified interface for deployment, regardless of target?
- **What's the right way to secure such an architecture?** Gone are the days of a single entry point into an application. In years past, one could authenticate the user and then trust that user as they navigated the modules of a monolithic application. In a microservices world, services are composed to satisfy business need, and each service has to be impose its own authorization conditions.

- **How do I uncover where latency is occurring?** It's one thing to trace activity in a monolith where there are only a few paths that a user can take. It's another thing entirely to try and figure out the answer to "the app is slow" when the app is powered by dozens of microservices. Teams need a new approach for tracing and analyzing service traffic.
- **What can I do to ensure high uptime for an app relying on multiple microservices?** A 99.99% uptime for a service is pretty good, right? That's just four minutes per month of service interruption. However, if an application relies on 10 services that all have 99.99% uptime, then the cumulative downtime drops to 99.9%, which equals almost 44 minutes of downtime per month! How can teams cut off poorly behaving services before a cascading failure occurs?
- **What's a performant way to route traffic to a dynamic pool of services?** Classic load balancers are application unaware, which are insufficient for this new world where service locations are fluid, traffic patterns must be analyzed and reacted to in near real-time, and layer-7[7] routing decisions play a key role.
- **How should I run microservices in production?** The idea of hundreds of microservices running all over the place must leave IT operators and information security professionals in a cold sweat. What's the right host for these services that provides a centralized runtime, but doesn't sacrifice the agility or scale that are the hallmark of microservices? Teams will want to consider platforms that are optimized for these types of workloads.

The biggest question that needs answering: How can a team embrace a microservices architecture without absorbing a massive new technical burden that hinders their ability to actually ship anything? Just getting an understanding of the existing tools in this space is difficult, and full of noise, to say nothing of how it might change existing processes. If teams aren't using proven, maintainable patterns in their microservices journey, the journey will be over before it starts. Fortunately, Netflix volunteered to be the guide by sharing their best practices.

CATALOGUING NETFLIXOSS

The Netflix open source effort—located at netflix.github.io—represents the most comprehensive set of battle-tested software for distributed systems available today. This software runs the gamut from optimizations for media encoding, to a "Simian Army" that can initiate massive infrastructure failures as a way to test resilience. The focus of this paper is on specific NetflixOSS software that solves problems for microservices architecture. The table below lists out the ones to be most familiar with.

7 "The OSI Model's Seven Layers Defined and Functions Explained". *Microsoft Support*. Accessed September 2016. https://support.microsoft.com/kb/103884

NAME	PURPOSE
Eureka	Consider this the “phone book” for locating microservices. The results of this service registry are used as part of middle-tier load balancing, among other things. It solves many of the problems associated with dynamic service addresses, and helps clients only send requests to “good” instances of a microservice.
Archalus	This solves the problem of how to serve up shared configuration values in a reliable way. With Archaius, stateless microservices always have up-to-date settings that are centrally maintained and refreshed without requiring dependent apps to recompile or reboot. The problem of consistent configuration properties across instances disappears.
Ribbon	Reduce load on a centralized load balancer and use client side load balancing to make intelligent decisions about where to send a microservices request. Ribbon integrates with Eureka to ensure that clients have an accurate representation of the landscape.
Hystrix	The goal of Hystrix is to isolate latency and reduce the impact of misbehaving services. It implements the “circuit breaker” pattern, meaning that when a certain number of exceptions occur, the software “opens the circuit” and doesn’t allow additional calls to the offending service until it make a set of successful calls. Hystrix helps teams avoid cascading failures by quickly responding to faults and providing a fallback mechanism.
Zuul	Described as a “front door for all requests”, Zuul is an edge service that routes traffic based on a set of filters. Netflix uses this software for a variety of purposes, including authentication checks, stress testing, static response handling, and intelligent, dynamic routing to specific backend clusters. Like many of the NetflixOSS projects, Zuul integrates with sister software like Eureka and Ribbon.
Atlas	Operating modern systems relies on timely metrics that provide near real-time intelligence. Atlas captures telemetry data used to measure the current health of the system, and offers a query capability to uncover pain points.
Spinnaker	This is the Netflix tool for doing multi-cloud continuous delivery. The goal is to make it straightforward and safe to deploy code through well-defined pipelines. Spinnaker currently supports AWS, Microsoft Azure, Google Compute Engine, Kubernetes, and Cloud Foundry targets.

This software is extremely powerful, but advanced. It’s daunting to consider how to consume and operate such sophisticated components. That’s where Spring Cloud comes in.

THE EMERGENCE OF SPRING CLOUD AND SPRING CLOUD NETFLIX

In 2014, the Spring team revolutionized Java development with the release of Spring Boot 1.0[8]. Later that year, Pivotal teamed up with Netflix on what would become Spring Cloud, the only microservice stack native to Spring Boot. The goal of Spring Cloud was to give developers an easy-to-use set of tools for building distributed systems. Instead of consisting of entirely novel software, Spring Cloud was about implementing other mature software stacks—including the work of Netflix—and exposing them through the annotation and template-based configuration that Spring developers were familiar with.

March of 2015 saw the 1.0 release of Spring Cloud[9], including the Spring Cloud Netflix services. Spring Cloud has grown since then, and now contains an unparalleled set of developer accelerators, including:

- **Spring Cloud Netflix (Eureka, Hystrix, Zuul, Ribbon, Archaius)** delivers distributed systems functionality based on NetflixOSS.
- **Spring Cloud Config** is a high-performing, Git-backed configuration server.
- **Spring Cloud Consul** permits service discovery with the popular Consul library.
- **Spring Cloud Security** makes it simple to implement OAuth 2.0 authorization flows with microservices.
- **Spring Cloud Sleuth** offers distributed tracing capabilities and integration with Twitter-born Zipkin.
- **Spring Cloud Stream** provides abstractions on top of engines like RabbitMQ and Kafka for messaging-based solutions.
- **Spring Cloud Dataflow** represents the modern approach to data microservices by offering data processing pipelines.
- **Spring Cloud Task** lets developers build short-lived, single-task microservices.
- **Spring Cloud Zookeeper** gives Spring developers access to the established Zookeeper ecosystem for service discovery and distributed configuration.
- **Spring Cloud for AWS** exposes AWS services like messaging, caching, data storage, and databases to Spring developers.
- **Spring Cloud Spinnaker** makes it possible to deploy all the various parts of Spinnaker (multi-cloud deployment) as a set of Spring Boot apps atop Pivotal Cloud Foundry.
- **Spring Cloud Contract** helps teams set up a service contracts and stubs to facilitate testing and collaboration.

8 Webb, Phil. “Spring Boot 1.0 GA Released,” *Spring by Pivotal* (blog), Spring, Apr. 1, 2014, https://spring.io/blog/2014/04/01/spring-boot-1-0-ga-released.

9 Syer, Dave. “Spring Cloud 1.0.0 Available Now,” *Spring by Pivotal* (blog), Spring, Mar. 4, 2015, https://spring.io/blog/2015/03/04/spring-cloud-1-0-0-available-now.

All of these projects are open source, and actively maintained by Pivotal and an engaged community of contributors. Usage of Spring Cloud is skyrocketing, with developers downloading it over 500,000 per month. This message of modern microservices development is clearly resonating, but what's the easiest way to consume and run this software?

HOW TO CONSUME SPRING CLOUD NETFLIX

The syntax of Spring Boot makes it simple to consume Spring Cloud Netflix software, and tools like Spring Initializr make it incredibly easy to get started For instance, for a microservice that needs circuit breaker behavior, the developer must:

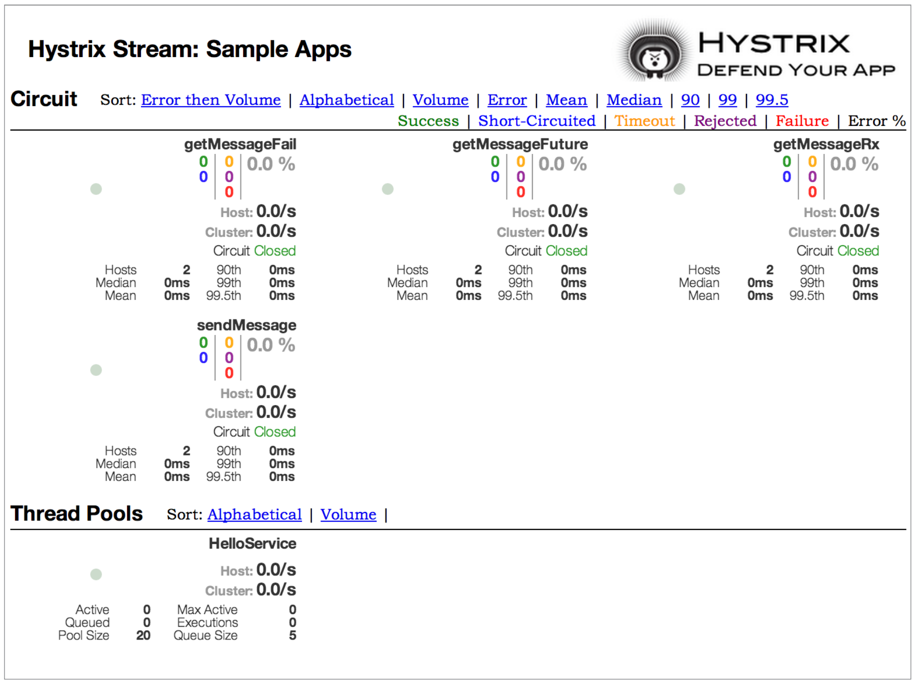
1. Add a dependency for spring-cloud-starter-hystrix. For developers using the Maven build system, it's a simple addition to the pom.xml file.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

2. Then within the Spring code itself, the developer adds a pair of annotations and defines a “fallback” method to use if the primary operation trips the circuit.

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {

  public static void main(String[] args) {
    new SpringApplicationBuilder(Application.class)
     .web(true).run(args);
  }
}

@Component
public class StoreIntegration {

  @HystrixCommand(fallbackMethod = “defaultStores”)
  public Object getStores(Map<String, Object> params)
  {
    //do stuff that might fail
  }

  public Object defaultStores(Map<String, Object> params) {
    return /* something useful */;
  }
}
```

3. Optionally, create an application (with the corresponding Spring Cloud Netflix dependency) for the Hystrix dashboard that delivers a live look at protected services.[10]



```
Hystrix Stream: Sample Apps

Circuit   Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5
                    Success | Short-Circuited | Timeout | Rejected | Failure | Error %

              getMessageFail                 getMessageFuture                 getMessageRx
              0 | 0 | 0.0 %                  0 | 0 | 0.0 %                    0 | 0 | 0.0 %
              0 | 0                          0 | 0                            0 | 0
                  0                              0                                0
          Host: 0.0/s                    Host: 0.0/s                      Host: 0.0/s
       Cluster: 0.0/s                 Cluster: 0.0/s                   Cluster: 0.0/s
        Circuit Closed                 Circuit Closed                   Circuit Closed
Hosts    2    90th   0ms     Hosts    2    90th   0ms       Hosts    2    90th   0ms
Median 0ms    99th   0ms     Median 0ms    99th   0ms       Median 0ms    99th   0ms
Mean   0ms  99.5th   0ms     Mean   0ms  99.5th   0ms       Mean   0ms  99.5th   0ms

              sendMessage
              0 | 0 | 0.0 %
              0 | 0
                  0
          Host: 0.0/s
       Cluster: 0.0/s
        Circuit Closed
Hosts    2    90th   0ms
Median 0ms    99th   0ms
Mean   0ms  99.5th   0ms

Thread Pools   Sort: Alphabetical | Volume |

              HelloService
          Host: 0.0/s
       Cluster: 0.0/s
Active      0   Max Active   0
Queued      0   Executions   0
Pool Size  20   Queue Size   5
```

Many things happen behind the scenes when including references to these Spring Cloud projects, but developers shouldn't have to spend time configuring boilerplate components from NetflixOSS. Rather, developers should spend all their time building innovative distributed applications that can easily take advantage of this essential software!

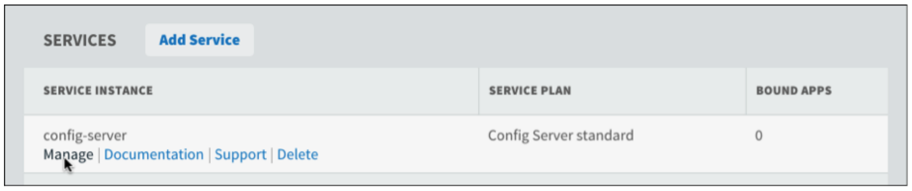
No doubt that Spring developers can build Spring Cloud-powered apps and run them virtually anywhere: on-premises data centers, container clusters, public clouds like AWS, or in a cloud-native platform like Cloud Foundry.

Pivotal wants to make it as easy as possible to install, configure, deploy, manage and secure NetflixOSS software that your microservices depend on. Enter Spring Cloud Services.[11] Spring Cloud Services packages a pair of core NetflixOSS projects—the Eureka service registry and Hystrix circuit breaker dashboard, plus Spring Cloud Config—into turnkey services that developers can deploy in Pivotal Cloud Foundry with a single click. Instead of building and maintaining a service registry, configuration server, or circuit breaker dashboard, teams can quickly deploy managed instances into their Cloud Foundry space.

10 Screenshots, code taken from official Spring Cloud Netflix documentation http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.1.5.RELEASE/

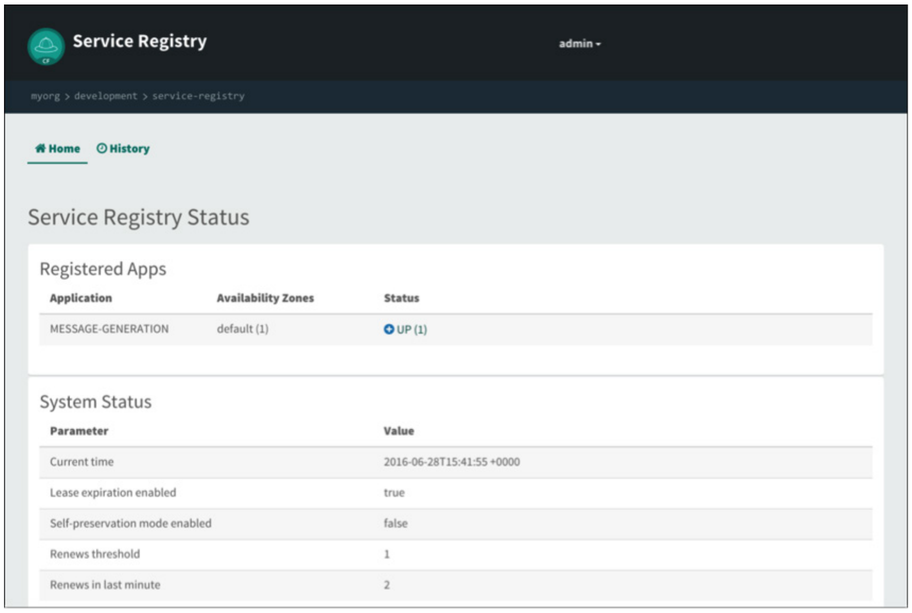
11 For Spring Cloud Services documentation, read http://docs.pivotal.io/spring-cloud-services/. You'll find a helpful explanation of these services at https://blog.pivotal.io/pivotal-cloud-foundry/products/now-available-spring-cloud-services-for-pivotal-cloud-foundry

Once the operator team installs the tile into a Pivotal Cloud Foundry environment, creating an instance is easy. For the Spring Cloud Config server, the developer can use either the command line interface, or Apps Manager web user interface to spin up an instance. The integration with Pivotal Cloud Foundry makes it simple to see what instances are deployed, and which applications are connected.



SERVICE INSTANCE	SERVICE PLAN	BOUND APPS
config-server Manage | Documentation | Support | Delete	Config Server standard	0

The Eureka Service Registry helps teams locate their numerous microservices. With the Spring Cloud Services tile installed, deploying a registry is straightforward: just use the command line interface or Apps Manager web user interface to ask the Service Broker for an instance. The Apps Manager-integrated dashboard gives you a live look at active services.



Service Registry Status

Registered Apps

Application	Availability Zones	Status
MESSAGE-GENERATION	default (1)	UP (1)

System Status

Parameter	Value
Current time	2016-06-28T15:41:55 +0000
Lease expiration enabled	true
Self-preservation mode enabled	false
Renews threshold	1
Renews in last minute	2

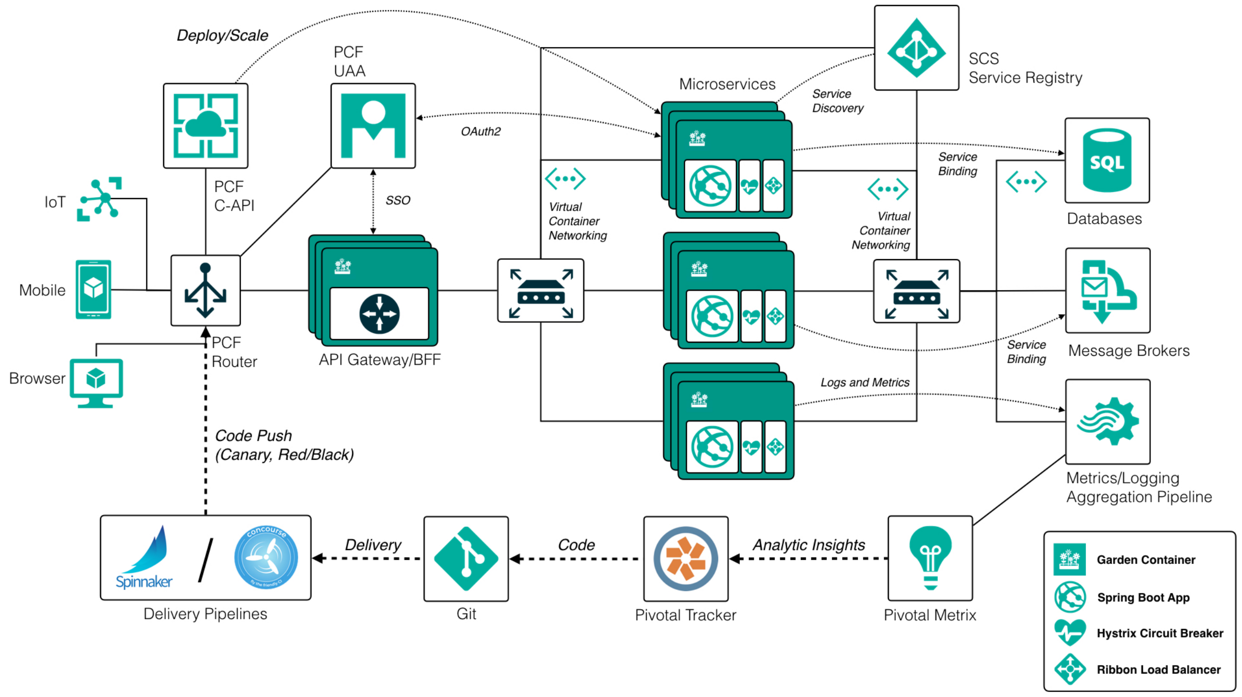
Finally, deploying the Hystrix circuit breaker dashboard into Pivotal Cloud Foundry ensures that you have a real-time view into any services that have tripped their circuit, and how the healthy services are performing.

Pivotal continues to invest heavily in Spring Cloud Services and will continue bringing more and more of the Spring Cloud family of software into this turnkey package. Spring Cloud Services can be found for download on the PivNet site.

WHERE TO RUN APPS POWERED BY SPRING CLOUD NETFLIX

We've established that a microservices architecture involves many moving parts. To achieve agility and flexibility, we invite some complexity. Operations is often a source of unexpected complexity with software. With only a handful of microservices in production, companies may not see any major operational issues. But, microservices at scale uncover challenges that have potential to derail your entire effort. How do we avoid this?

Look at a modern microservices architecture. There are dozens (or hundreds!) of continuously delivered services, accessed by diverse clients, discovered through a service registry, secured via OAuth 2.0, protected by circuit breakers, sharing configuration values, and logging data to a centralized collector. This is an extremely powerful topology that gives companies immense control over how they update and scale individual services. However, there are now more components to deploy, manage, monitor, troubleshoot, scale, and evolve.



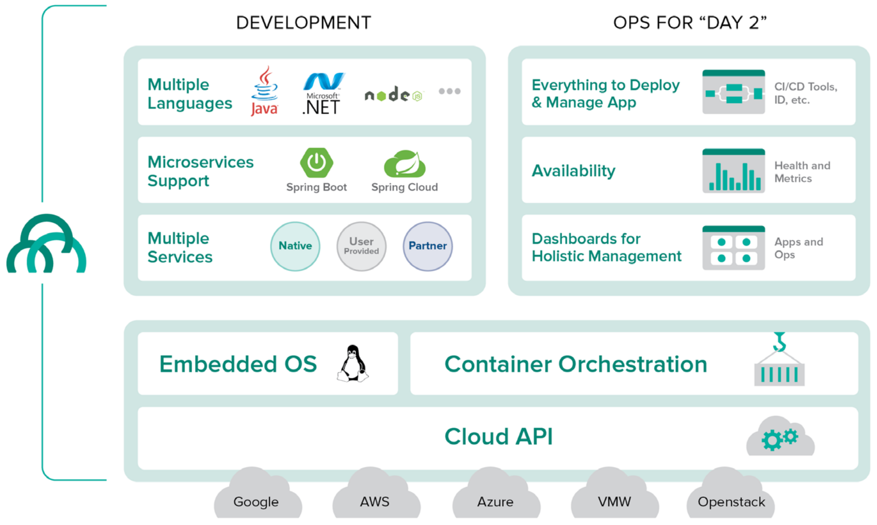
In this paper, we have looked primarily at the software that makes up the NetflixOSS stack and Spring Cloud. However, it's absolutely critical to find the right place run this software so that it doesn't dramatically increase your operational cost.

What matters most when assessing the right home for NetflixOSS-powered software?

- **Supportive of CI/CD pipelines**. A microservices-friendly platform makes it easy to quickly and continuously deliver software. That means that it exposes APIs for publishing code, and ideally, handles the configuration tasks that are extremely error prone when done manually: configuring containers & operating systems, middleware, load balancers, firewall rules, and logging infrastructure. Similarly, keep an eye out for platforms that support blue/green deployment—a release pattern that reduces downtime by deploying to a parallel environment and switching traffic over once the new version of the application is ready to serve traffic.
- **Offers reliable, scalable infrastructure**. One of Netflix's key reasons for moving to microservices was "scale." Any platform hosting microservices must be able to rapidly scale up and down to meet demand. The components that comprise the platform have to be robust and reliable so that there are no single points of failure that can make the microservice unavailable. Look for multiple layers of high availability, first at the infrastructure tier, and then through the platform components.
- **Provides uniform capabilities for services, regardless of language or runtime**. Some teams forego the economy of scale in using a single language & toolchain to develop services, working instead with their available talent pool and developing in multitude of programming languages. Flexibility is king! The team independence that comes with microservices means freedom to choose the frameworks that make the most sense for their business problem and skill set. But what you don't want, is unique deployment or management procedures for each type of application! Rather, look for platforms that, regardless of the type of application, provide uniform behavior for deployment, logging, monitoring, and scaling.
- **Runs atop multiple IaaS platforms**. There's no debate that the adoption of public cloud is skyrocketing. At the same time, companies do have legitimate worries about coupling to one infrastructure provider over another. Try to identify platforms that provide an IaaS-agnostic feature set that works with your preferred cloud provider.

- **Simplifies "day 2" operations for services**. One of the biggest risks with microservices involves the operational upkeep of this more complex environment. Make sure to review platforms that simplify activities like patching services beneath the platform, scaling individual microservices, isolating "tenants" in the environment, and more. Look for platforms that maximize the productivity of the entire team, not just developers.

Pivotal Cloud Foundry is the world's premier cloud-native platform. It's well-suited for existing applications or completely new ones. Whether hosted in a public or private cloud environment, Pivotal Cloud Foundry provides a consist experience for developers and operators. It's purpose built for software teams that want to constantly deliver custom-built, scalable applications. As mentioned above, Spring Cloud Services are built into Pivotal Cloud Foundry and ensure that NetflixOSS technology is easy to use, and fully managed by the platform.



WORKING WITH THE BEST ON YOUR DIGITAL TRANSFORMATION

A "not invented here" stance is a dangerous position for companies that expect to compete in the digital world. Time is of the essence, and companies can't afford to invest years of time and untold dollars building platforms to run their modern microservices workloads. Fortunately, there's help available.

Netflix built an arsenal of technology to run one of the most sophisticated distributed systems in the world. They did so, in part, because the existing technology wasn't suitable for their then-uncommon needs. Few companies dealt with the challenges that Netflix did.

Fast-forward to today where developers at companies of all sizes are delivering global services with traffic that would have been unheard of just a few short years ago. Instead of reinventing the wheel and absorbing unsustainable complexity to deliver these services, smart teams have embraced the NetflixOSS stack, as delivered through Spring Cloud. The Netflix engineers themselves find significant added value[12] in Spring Cloud; frameworks focus on more complete enterprise readiness versus the raw innovation in the components. Those workloads run best on the Spring-optimized Pivotal Cloud Foundry platform where developers can focus on their apps, not the underlying infrastructure. At the recent SpringOne Platform conference, former Netflix chief architect Adrian Cockcroft said[13]:

> *"We have Cloud Foundry and we have the Spring Cloud, Spring Boot stuff, and it's just there. And if you're doing Java, there really isn't anything else out there that's got this level of support. That's just going to give you all of this stuff … you don't need to go build it yourself anymore. You need to build on top of it. You need to build something interesting that is your product that's going to be the next generation thing that adds your business value.*
>
> …
>
> *Let's just do Spring Boot on Cloud Foundry and use a bunch of those Netflix projects that got bundled into it, and move on to the discussion. It's scaffolding."*

Visit us at pivotal.io to get started today with a free trial of Pivotal Web Services. You can also download the free, local edition of Pivotal Cloud Foundry—with Spring Cloud Services baked in—called PCF Dev. Let Pivotal help you transform how you deliver microservices powered by NetflixOSS and Spring Cloud!

12 Schneider, Jon and Wicksell, Taylor. "Spring Cloud at Netflix." Presentation at SpringOne2GX, Washington, D.C., September, 2015. Accessed September 2016.

13 Cockcroft, Adrian. "Simplifying the Future." Presentation at Spring One Platform, Las Vegas, Nevada, August 2016. Accessed September 2016.