Practical Microservices

A Pragmatic Approach

Rohit Kelapure

Table Of Contents

1 Preface
License
3 Motivation
- 3.1 Microservices
- 3.2 Platform As A Service (PaaS)

4 Microservice architectures
- 4.1 What Microservices Architecture Is Not ?

5 Terms
6 Cloud Foundry
7 Applicability
- 7.1 Social
- 7.2 Technical

8 Cloud Friendly Applications
9 Development Methodology
- 9.1 Local Development, Cloud Production
- 9.2 Samples
- 9.3 Build - Separate source code repository for each service
- 9.4 Packaging
- 9.5 Externalized Configuration - Comes from the environment

10 Design
- 10.1 Modeling
  - 10.1.1 Entity Modeling
  - 10.1.2 Event Modeling
  - 10.1.3 Modeling Tools
- 10.2 Granularity
- 10.3 Packaging
- 10.4 Partitioning
  - 10.4.1 Domain Driven Design (DDD)
  - 10.4.2 Breaking the monolith
  - 10.4.3 Implementing Microservices from scratch
  - 10.4.4 Strategies For Introduction of Microservices in Brownfield enterprise systems
    - 10.4.4.1 Co-existence & Integration
    - 10.4.4.2 Technical Transition
    - 10.4.4.3 Strangler Pattern
    - 10.4.4.4 Social Transition
- 10.5 Inter-service/Inter-process communication
  - 10.5.1 HTTP Synchronous Request/Response Style Interaction
  - 10.5.2 Asynchronous Integration of Services with Events
    - 10.5.2.1 Event Collaboration

- 10.5.2.2 Event Sourcing
- 10.5.2.3 Reactive Applications
- 10.5.3 Hybrid Mode
- 10.6 Binding Cloud Foundry Services to a Microservices Application
- 10.7 API Gateway aka Software Reverse Proxy for Microservices
- 10.8 Service discovery
- 10.8.1 User Managed Services
- 10.8.2 Spring Cloud - Netflix Eureka Integration
- 10.8.3 External Data Stores
- 10.9 Rest API
- 10.9.1 Granularity
- 10.9.2 Linked Data
- 10.9.3 Documentation
- 10.9.4 Design
- 10.10 Security
- 10.11 Persistence
- 11 Governance & Social Engineering
- 12 Testing
- 13 Deployment, Continuous Integration and Continuous Delivery
- Service Versioning
- Dark Launching Features
- Continuous Integration with Jenkins
- Continuous Integration with Wercker
- Cloud Foundry Application Lifecycle Management
- Rollback
- 14 Monitoring - Is my System Healthy ?
- Centralized Logging
- X-Ray Microservice Call Flow
- Distributed Tracing
- Synthetic Monitoring
- Cloud Foundry
- Metric Event Streams
- The following actuator endpoints are available for spring-boot: [learning-spring-boot]
- Customization of Spring Boot Actuator Endpoints
- JMX Monitoring
- End-To-End Request Flow Correlation
- Performance Testing
- Service Monitoring
- System Monitoring
- Automatic Baselines & Percentiles Based Alerting
- Next Generation Monitoring
- 15 Conclusion
- 16 Videos & Articles

17 Credits

1 Preface

The motivation for this book is to peel the microservices architecture and terms as invented by industry titans and understand what it means from a practical perspective. Understand what tool choices are necessary to become successful with such an architecture. The fundamental practices in terms of development and software modeling that will lead to the benefits of microservices. There is a lot of hype around microservices with a lot of players now jumping on the bandwagon. I am one of them. I have taken the time and effort to formulate an opinion which is contained as practical stratagems and advice in this book. In terms of libraries, frameworks, languages and deployment platform this book has a heavy slant towards JVM based technologies and Pivotal Software like Spring and Cloud Foundry. The reason for this is two fold 1. my own familiarity with Spring/Cloud Foundry and 2. a genuine belief that CF + Spring + Open Source is the right fit for your microservices architecture. Where possible I have tried to keep the vendor speak to a minimum. Overall this is a book everyone should get value out of, regardless of their programming language, starting from the novice developer to the software craftsman.

License

3 Motivation

3.1 Microservices

We sincerely believe in the premise that, “On the back of software programming tools and Internet-based services Software is Eating the World” [andreesen]. Software is disrupting and eating much of the value chain of traditional major businesses and industries ranging from book sellers, retail, defense, agriculture etc., Industry analyst Gartner predicts that in a Bimodal IT world that has become ripe for digital disruption, by 2017, a significant disruptive digital business will be launched that was conceived by a computer algorithm. Given these stressors it is critical for enterprises to enable agile IT to iterate, prototype, develop and rapidly deliver software. Implementing a rapid feedback and rapid release loop is critical for survival. Microservices is a style of architecture that enables such an outcome for the digital business.

Microservices facilitate autonomy with responsibility. Microservices engender systems that provide flexibility and composability in terms of architecture, scalability and operational semantics. Microservices when done right increase the productivity of product teams by reducing developer cognitive load and separating concerns. Microservices unlike monolith applications decouple change cycles and enable frequent deploys of small well-tested cohesive components to production leading to quicker turnaround of features and a data driven analytic application.

A microservices based architecture for an application will allow large-scale architecture changes with minimal external impact providing insurance against unforeseen problems.

The fundamental tenets of microservices applications enable the creation of data-driven analytics based smart apps. A triumvirate of transformative patterns i.e. 1. Open source eating the core enterprise IT infrastructure and proprietary software becoming the black sheep, 2. Form factor agnostic light weight applications developed based on cloud native models instead of heavyweight proprietary legacy programing models and 3. agile process transformative lean enterprise methodologies treating software, infrastructure and operations as code are creating a landscape where developing apps as microservices and automated deployment on a PaaS platform is the only way to survive the upcoming software apocalypse.

3.2 Platform As A Service (PaaS)

Any enterprise platform that serves the needs for developers and operators needs to provide the following qualities of service

Self-Service Provisioning: The ability to push applications and provision services for to production environments without dependence on other teams and gatekeepers.
Security and Control: Individual and team-based permissions, automated and enforced processes for new application instance rollouts, authentication system integration and intrusion monitoring.

High Availability: An auto-healing, automatically updated and redundant system that requires no intervention to restore an application instance failure, along with application monitoring for responsiveness, capacity and service-level maintenance.

Reliability: Support-tested components across all different runtimes and services.

Simplicity: Easy to use management consoles for developer to enjoy GUI management of application environments, scriptable APIs for DevOps teams, and turnkey installers for major private and public cloud platforms.

Data-Driven: One-click connection to all common data stores for instant access to data stores of any size. Provisioning data services through PaaS supports the creation of truly data-driven applications that can meet the needs of multiple customer segments by leveraging advanced analytics.

Click to Scale: Deploy to one instance or one hundred with a single click. Dynamically provision resources as usage increases.

Live Upgrades with Zero Downtime: “Blue/Green” update process allows application versions to run in parallel with instantaneous switchover when ready to bring the newer application into production.

Foundation, Vibrant Community and Access to Source Code: Buy-in from developers, access to active forum help, visibility into code, and access to platform abstractions below the surface providing insurance against vendor lock-in and collateral for vendor independence.

Customizability: Extend the platform for new runtimes and services without support or contract headaches.

Laws of Unix programming
Distributed system
network fallacies
remote objects - sun paper

4 Microservice architectures

Microservices is the first architectural style developed in a post continuous delivery world founded on the practices of continuous integration and continuous delivery. The key aspects of a microservices based architecture are listed below -

Componentization via Services: Services are independently deployable, encapsulated, out-of-process components that separate concerns and can be composed into a system of systems with their interactions defined via service contracts (APIs).

Organized around Business Capabilities: Services are vertical slices of technologies dedicated to particular business capability, ideally with coinciding service and team boundaries. Speed wins in the marketplace. Therefore leverage microservices for speed and availability.

Products not Projects: The team should own the service over its entire lifecycle, both development and support. Establish a high trust low process culture where developers are given freedom and responsibility. Remove friction from product development.

Smart endpoints and dumb protocols: Logic should reside within the services rather than the communications mechanisms that tie them together such as an HTTP request response style interaction or lightweight messaging.

Decentralized Governance: Because of the partitioning inherent in this style, teams are not forced into one set of standards for their internal implementations.

Decentralized Data Management: Because of the partitioning inherent in this style, data will be distributed across multiple services across availability zones and regions with a tendency towards eventual consistency.

Infrastructure and Test Automation: Automated testing, automated deployment and automation of infrastructure. Auto scaled capacity and deployment updates.
Design for failure: The distributed, partitioned nature of microservice architectures increases the need for system monitoring and resilience.

Evolutionary Design: Smaller, modular services enable agile, controllable change. Isolate business logic in stateless microservices. Microservices scale the development process by reducing cognitive load.

Hexagonal Architecture **(aka Ports and Adapters):** A way of structuring code for each service that separates the core domain model (What to do) from the ports and adaptors (How to do). Hexagonal architecture provides separation of concerns, ease of evolution and testability. Each of the hexagon's sides represents a different kind of port, for either input or output conversation with an external component. Sides consist of a port and an adaptor pair. By replacing the adaptors you can change how to the conversations happen. A hexagon handles 6 different types of conversations. Organize the methods on ports and adapters to reflect the conversations the core is trying to have. ports based on what conversation the core is trying to have with the outside world. Think of a Port as protocol endpoints like HTTP and the Adapter as a protocol handler like a Java Servlet or JAX-RS annotated class that receives method invocations from a Java EE container or framework (Spring, Jersey). [Skillcast & Implementing DDD].

Types of architectures
http://iansrobinson.com/2009/04/27/temporal-and-behavioural-coupling/

4.1 What Microservices Architecture Is Not ?

Microservice architecture is not a layered architecture where presentation, business and data handling is realized as separate tiers to scale presentation and compute processing independently of the data tier. Microservices is also not an intelligently integrated architecture where dumb services are integrated via a smart ESB or a centrally integrated architecture where anemic services are integrated with a central database. [thoughtworks-lessons-frontline]

5 Terms

Service is an extremely overloaded term in the cloud vocabulary. In the context of this document a microservice is synonymous with service i.e. an application deployed as a war or a jar file/s that exposes a REST API to provide a specific business capability. Cloud Foundry (CF) has its own notion of managed and user based services. These services when referred to, are explicitly called out as Cloud Foundry services.

A distinction is also made between Cloud Foundry and its commercial on premise distribution by Pivotal called Pivotal CF. By default the treatment of microservices covered in this document applies to the open source CF. Where applicable we explicitly call out Pivotal CF.

Platform As A Service

First lets go back to the basics and review some definitions from the National Institute of Standards and Technology defines PaaS as -

td

Why Bother With a PaaS ?

Self Service

A PaaS provides a platform that empowers developers - allowing them to self-provision infrastructure and push applications and services to production. A PaaS built on top of open container format enables applications to be packaged in a deployment neutral format. PaaS provides a layer of abstraction over your cloud infrastructure preventing vendor lock-in allowing your app to migrate across IaaS's like Amazon, Vmware, Azure, OpenStack. Google Cloud Platform etc.,

Agility

A PaaS gives us an opportunity to radical rethink our existing approaches to software development and put in new pillars in place to out innovate the innovators. Move to practices you may have only tried a little like 3rd generation app architecture, CI, pair-programming , TDD, collocated teams, push-button deploys and other lean product development practices. The only way to effect this transformation of development practices to achieve fast predictable high sustained productivity is to go all out. You can't half ass the agile transformation. BE BOLD and go for it. This will feel like letting go. Release often and fail fast and perhaps even make mistakes. If all this sounds impossible then, there is a playbook for this social transformation. Carve out a real piece of business responsible for revenue such as a mobile application. Create a digital transformation center of excellence which will be a home for these practices. Once success is demonstrated with the canary, the rest of the enterprise will follow. [pivotal-labs-edward-hieatt-web-summit-2014]

12 Essential Elements of a Modern PaaS

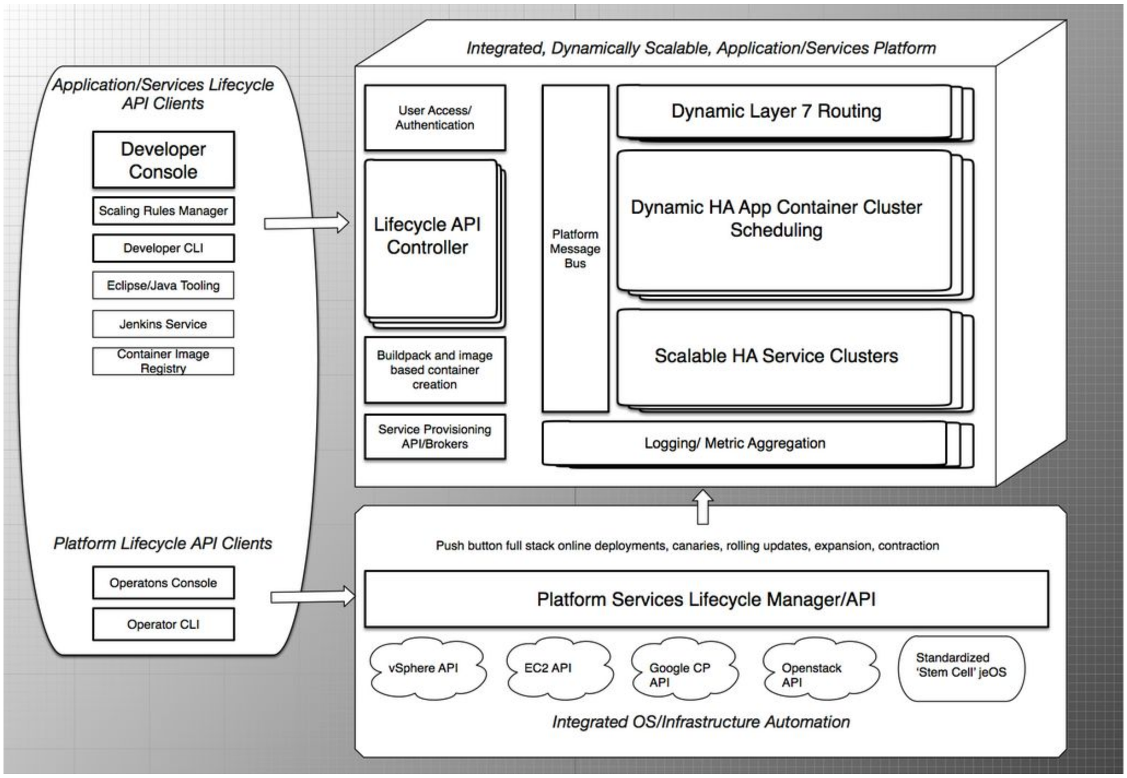
1. Focusing on developer experience and a flat learning curve is critical for wide adoption in the enterprise.
2. Consolidated technology stack to support multiple languages and frameworks on one platform.
3. Built-in and natively managed marketplace of services and ecosystem of service providers.
4. Enhanced dashboards showing application events, status, firehose of metrics and logs, resource consumption for app developers and system centric views & hooks for operators.

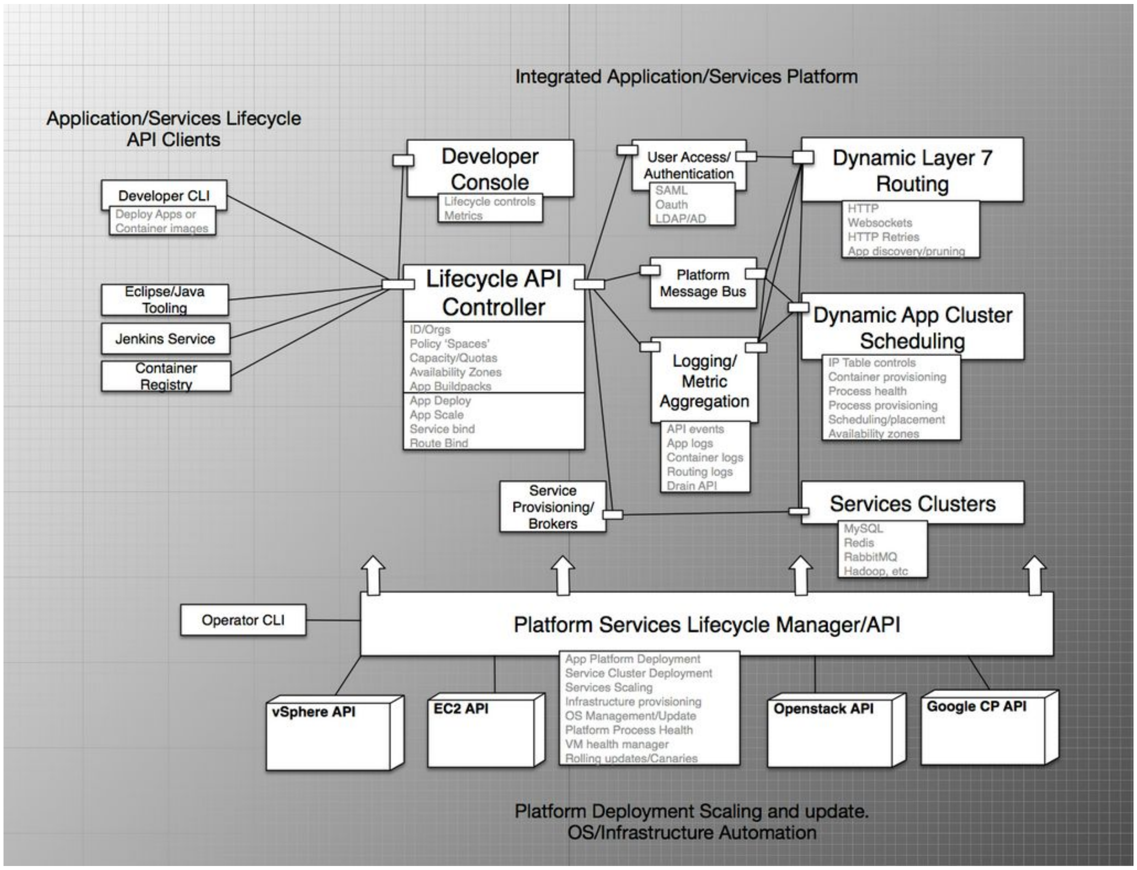
5. Ability to enact enterprise policy controls and governance.
6. Horizontal auto scaling to meet peak user demands while reducing hardware costs.
7. Designed for high application availability.
8. Built in access to Application monitoring and metrics and ability to snap in 3rd party monitoring tools.
9. Built for modern design and development practices. Deploy the same application “bundle” through Continuous Delivery pipelines with zero downtime blue-green deploys.
10. Flexibility to run and port your app on any infrastructure cloud.
11. Comprehensive security apparatus including built in features for Identity Management (Authentication), Role Based Access Control, Authorization, layered security model and network protection (hardware and software) for inbound and outbound traffic.
12. Open Source with an inclusive participation model allowing developers to contribute, influence and peel away at layers of abstraction.

Before you pick a particular PaaS solution please evaluate its features against the checklist above. Furthermore enterprise architects building a platform should evaluate the features of a proprietary platform with a standard PaaS. There are very few reasons you are writing a enterprise platform from scratch.

Lexicon Of Required modern PaaS functionality

The couple of pictures below by James Watters VP Cloud Platforms @ Pivotal illustrate nicely the 12 elements of a modern PaaS. Please note the architecture diagrams are not CF specific and apply to any enterprise PaaS.


Integrated, Dynamically Scalable, Application/Services Platform
Application/Services Lifecycle API Clients
Developer Console
Scaling Rules Manager
Developer CLI
Eclipse/Java Tooling
Jenkins Service
Container Image Registry
Platform Lifecycle API Clients
Operatons Console
Operator CLI
User Access/ Authentication
Lifecycle API Controller
Buildpack and image based container creation
Service Provisioning API/Brokers
Platform Message Bus
Dynamic Layer 7 Routing
Dynamic HA App Container Cluster Scheduling
Scalable HA Service Clusters
Logging/ Metric Aggregation
Push button full stack online deployments, canaries, rolling updates, expansion, contraction
Platform Services Lifecycle Manager/API
vSphere API
EC2 API
Google CP API
Openstack API
Standardized 'Stem Cell' jeOS
Integrated OS/Infrastructure Automation



Differences between IaaS and PaaS

The differences between a PaaS and an Iaas when it comes to microservices based architectures is best illustrated by the following comparison done by Sergey Sverchkov, Solutions Architect at ALTOROS as part of my latest research paper, "Microservices Architecture: The Pros, Cons, and How It Compares to Monolithic Approaches (+ Cloud Foundry Examples)."

Capabilities of microservices solutions	IaaS implementation	PaaS implementation (Cloud Foundry)
1.One service for one job	Every service is deployed on an IaaS instance (a physical or virtual machine) by a	A service (or an application) is deployed by a developer. Scalability can be controlled by a developer. Communication endpoints are served by

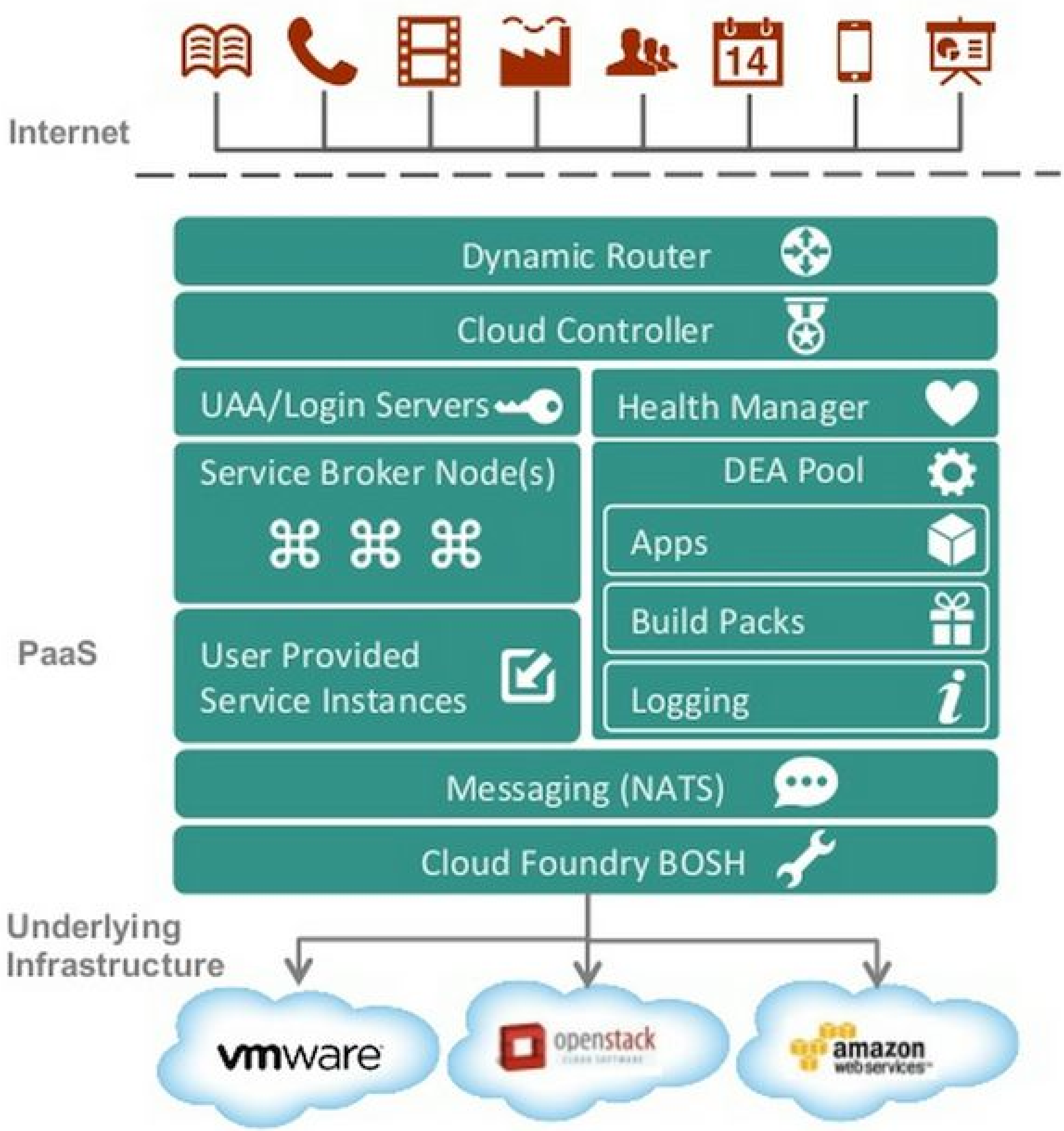
	QA/DevOps team. DevOps are responsible for configuring valid communication interfaces. Scalability is provided by the DevOps team.	the PaaS. You just need to assign a unique name to the service in the root PaaS domain. You do not have to think about IaaS. Instead, you will be able to focus on implementing business logic for each of the services.
2.Using different tools to implement different services	The DevOps team needs to configure an application runtime on IaaS instances.	An application runtime is automatically deployed in a PaaS container.
3. Loose coupling	The DevOps team manages IaaS instances used for service deployment.	PaaS containers are isolated elements for application deployment. Container lifecycle is managed by the PaaS.
4. Independency of developers	DevOps may need to create multiple IaaS environments for each of the development groups.	When a PaaS is used, development groups can be managed as “organization” units. Deployments for development and testing can be arranged as “spaces”. You can have multiple “organizations” and “spaces” in a single PaaS deployment.
5. Continuous delivery	DevOps engineers need to install and configure build-automation tools and	Build-automation solution can be deployed in Cloud Foundry as a regular application. This reduces the

	integrate them with a project repository to provide continuous delivery.	time necessary to provide continuous delivery for a project—compared to IaaS.
6. Integration with external services	The DevOps team deploys external services. Applications connect to external services using properties.	A service broker of the PaaS can be used to deploy and publish some external services. Service binding makes it easier to connect an application instance to external services.

The Cloud Foundry PaaS

Cloud Foundry has been written ground up for the development of 3rd generation web application motivated by the twelve-factor methodology for building apps. Cloud Foundry combines operating system, load balancing, middleware and software development frameworks into a single platform to deliver turnkey automation for developers and operators. Developers can deploy faster with more choices in terms of programming language and persistence options. Operators can focus on streamlining operations and telemetry instead of fire fighting and maintaining complex IaaS deployment scripts.

Cloud Foundry Components



Cloud Foundry is itself an orchestration of microservices and as such makes an excellent case study for a microservice architecture. A description of the components of cloud foundry informs us about the concerns of building a large scale distributed system and also imparts a better understanding of how to use the platform. The coverage of Cloud Foundry here is to provide one concrete example of how a PaaS platform can speed up the deployment of a microservices system.

Component (Language)	Description

Dynamic Router (Go)	Routes incoming traffic to the appropriate component; generally the CloudController or application
Cloud Controller (Rails)	Exposes a REST API to the system. Manages a database of apps, services, service instances, etc
UAA (Spring Tomcat Java)	Identity management service for the platform. Acts as an OAuth2 SAML, OpenID and SCIM provider
Health Manager (Go)	Monitors the state of applications and ensures that the correct number of instances are running
DEA Pool (Go)	A collection of code that is responsible for transforming pushed app artifacts into a ready to run droplet
Apps (Polyglot)	End user provided code that is “pushed” to the cloud and packaged to run in the warden container
Buildpacks (Polyglot)	A collection of code that is responsible for transforming pushed app artifacts into a ready to run droplet
Logging (Go)	Provides a stream of log output from your application and from CF system components that interact with your app during updates and execution
Service Brokers (Polyglot)	Advertises service offerings to the Cloud Controller and handles requests to create, bind, unbind, and delete managed service instances.
User Provided Service Instances	Service instances managed outside of Cloud Foundry. Don’t adhere to the Service Broker API
NATS (gnatsd)	High performance publish-subscribe open source messaging system for cross-component communication
BOSH (Ruby)	A distributed deployment install and management tool that abstracts the details of the IaaS layer

Cloud Foundry’s fundamental unit of scaling, the droplet promotes one codebase tracked in revision control and multiple deploys. Cloud Foundry’s emphasis on deployable units (i.e. war, jar) along with the CF buildpack wiring of runtime dependencies allows applications to explicitly declare, isolate and inject external dependencies. Backing services are treated as attached resources via the cf create-service and cf bind-service commands.

Droplets generated by CF buildpack are run in immutable warden containers enforcing strict separation between build and run stages. The encapsulation of the application code and the runtime in a droplet allows CF to execute the app as one or more stateless processes. There is no clustered memory or shared file system. CF scales quickly; however can only move as fast as your app can bootstrap. CF spaces provide for separation of concerns and controls without technical differences.

Cloud foundry provides a highly available platform for application deployment. Cloud Foundry provides 4 levels of HA via application instances spread across availability zones, failed application instance restart by the Health Manager, VM resurrection by BOSH, recovery of processes on the VM by monit.

Cloud foundry treats logs as event streams providing an unified log stream for applications in cloud foundry. This allows developers to continually drain their application logs to 3rd party log archive and analysis services. Operators and administrators can access the firehose, which includes the combined stream of logs from all apps, plus metrics data from CF components.

Every application deployed to the platform is automatically wired to roles management and policy enforcement and audit events and resource usage dashboards. Cloud Foundry obviates infrastructure concerns for developers, abstracting out underlying platform differences to provide an uniform deployment, monitoring and logging mechanism.

Cloud Foundry mitigates polyglot language and environment provisioning complexity via CF Buildpacks. Cloud Foundry mitigates routing/load balancing and plumbing concerns via CF Router and CF Services. CF provides buildpack agent based tooling for monitoring as well as metric streams via the firehose. CF BOSH and Pivotal Operations Manager provide high quality operations infrastructure and release management to not only recreate a CF deployment consistently but also to keep it running in production. With BOSH, it is NOT difficult to recreate CF environments in a consistent way for either manual or automated testing. Robust release/deployment automation can be performed via the scriptable GO language based CF API and the Maven/Gradle CF plugins.

CF platform features help tackle some of the complexity of microservices running on distributed systems like latent or unreliable networks, fault tolerance and load variability. Thanks to the choices made by the platform, microservices designed with the 12 factor app methodology are compatible and natively optimized in Cloud Foundry.

Applicability

7.1 Social

Before you embark on a microservices architecture for your enterprise you should establish a DevOps culture of production monitoring, rapid application deployment and provisioning implying close collaboration between developers and operations. This will often require a reorganization of teams. DevOps cannot be added like salt and pepper to a product. The CloudFoundry PaaS facilitates these baseline competencies enabling organizations to embrace microservices; however without the organizational shift to product centered teams that invert Conway’s law, the long term benefits of microservices are negated. [References]

7.2 Technical

There is a certain threshold beyond which microservice style applications make sense. Cloud Foundry provides the baseline competencies to be successful with microservices i.e. 1. Rapid Provisioning, 2. Basic Monitoring and 3. Rapid Application Development. For your average enterprise application, a proven monolith architecture with clean component boundaries and the ability to scale in a cookie-cutter fashion using existing transactional paradigms like 2-phase commit makes sense. It is critical to balance speed of development/deployment while minimizing the risk of introducing change to critical systems. Microservices is not a panacea for architectural and scalability problems. In fact the same principles of SOLID design i.e. single responsibility, loose coupling, high cohesion, DRY, rigid interfaces, tolerant readers apply to both monolithic and microservices based systems. But a microservices architecture makes the process separation explicit, making it easier to separate bounded contexts.

8 Cloud Friendly Applications

- Shared State is evil
    - Statelessness leads to scalability
    - Coordination of shared state across peers impacts performance
- No filesystem access
    - Local file system storage is short-lived
    - Instances of the same application do not share a local file system
- Configuration via environment or via an external configuration server (github/Zookeeper/etcd)
- Inject external dependency connection and credential information via services
- Console based logging
- Monitoring via services and frameworks
- Local development, cloud production

- All persistent data and state including HTTP sessions, caches etc., to be persisted and replicated by services like MySQL, REDIS cloud, etc., in external data stores.
- Execute application as one or more stateless processes enabling scale-out via a process model
- Explicitly declare and isolate dependencies
- One codebase tracked in revision control, multiple deploys
- Fast startup and graceful shutdown
- Make no distinction between local and third party services
- Treat logs as event streams
- Immutable code with instant rollback

9 Development Methodology

9.1 Local Development, Cloud Production

Develop microservices applications using Spring Boot and Spring Cloud projects. Spring Boot provides a nice abstraction layer for portability between local and cloud development. Spring Boot takes an opinionated - convention over configuration view of the Spring platform and third-party libraries so you can get started and deploy apps to production with minimum fuss. Spring Boot enables development in java, groovy or scala and packages apps as self contained jar and container deployable war files. Spring Boot tackles dependency hell via pre-packaging and the smart use of spring-boot-starter projects. A detailed treatment of spring cloud can be found in the following section.

1. Leverage the use of Spring programmatic configuration to configure the cloud and default profiles. Spring Boot automatically extracts the Cloud Foundry services connection and credential information from the VCAP_SERVICES environment variable and flattens the data into properties that can be accessed through Spring's Environment abstraction. Once the application is running in the cloud it is a best practice to switch to running with the "cloud" profile. Take a look at the SampleWebApplicationInitializer code from the rapwikiapp that dynamically switches the profile at runtime to "cloud" when running in CF.
2. Use the Spring Cloud local connector for local development and testing without mocking up `VCAP_SERVICES`. The connector provides the ability to configure Spring Cloud services locally for development or testing.
3. A less desirable approach is to ALWAYS set the `cloud` profile and mock the VCAP_SERVICES. see Externalized Configuration for more details. One way to do this is to configure the local tomcat or jetty server with `-DVCAP_SERVICES={mock JSON Service connection and credential information}`. Choice of IDE is left to the development team. STS, IntelliJ and Eclipse have plugins for deploying to cloud foundry. see [CF Eclipse plugin] and [Run/Debug config for CF server]

Spring Boot certainly makes it easy and convenient to develop for the cloud; however it is not mandatory since any spring application that has been designed natively for the cloud will work on CF. Features like Spring's support for Environment and profiles are applicable across not projects, not just restricted to Spring Boot. To sum it up, proper use of Spring profiles encapsulates conditional application configuration and ensure that Applications **run** locally and in the cloud **without** modification.

9.2 Samples

Sample spring projects to understand and build a microservices architecture. These samples leverage spring-data, spring-data-rest, spring-cloud, spring-cloud-* and spring-boot to build an application based on microservice architecture

- https://github.com/spring-cloud-samples … Applications for Spring Cloud with Netflix use cases
- https://github.com/spring-cloud-samples/scripts … Deploying and updating the sample spring cloud applications to cloudfoundry
- https://github.com/spring-cloud-samples/customers-stores … Microservices with Netflix integration. Table below shows the location of spring netflix annotations in code
- https://github.com/cloudfoundry-samples/spring-music … Provides documentation on how to run the same application locally and on cloud foundry and showcases spring-cloud.
- https://github.com/4finance/micro-infra-spring … Non cloud foundry repository for groovy code that sets up the Spring infrastructure stack for microservices. Provides Spring modules for service discovery using ZooKeeper, Spring environment setup, Health check, Metrics publishing, CorrelationId setting. request logging, customized rest template for service discovery
- https://github.com/cf-platform-eng/spring-boot-cities … Script for building a cloud ready microservice with Spring Boot, Spring Data, and Spring Cloud.
- https://github.com/olivergierke/spring-restbucks … Showcase for integrating Spring Data JPA, Spring Data REST and Spring HATEOAS to implement a hypermedia REST web service
- https://github.com/bijukunjummen/rapwikiapp … Explores the integration of GemFireXD and RabbitMQ with Spring Cloud, Spring Boot and Spring Securit

9.3 Build - Separate source code repository for each service

Each microservice represents a business capability and should have its own git source code repository. By extension each microservice should then generate one atomic deployable unit i.e. a collection of one or more artifacts that is pushed together to the cloud. A tool that helps with updating multiple git repos all at once is the MR tool (http://myrepos.branchable.com/). If each module is cloned into a separate folder underneath one parent folder, then multiple edits can be made quicker across multiple repositories if desired. This is very useful when updating identical files across repositories with a single command.

9.4 Packaging

When developing locally the application can be packaged as a fat jar file that includes an embedded Apache Tomcat server and when deploying on the cloud it is **recommended** to package the same application as a war file which can be pushed to the java buildpack. Packaging as a war file only bundles the application and the dependent framework bits without dragging in the Apache Tomcat Server binaries. You can use Maven or Gradle to build spring projects with their corresponding plugins cloud foundry gradle plugin and cloud foundry maven plugin to deploy and manage apps to Cloud Foundry. Each microservice should have its own git repository, manifest, dependencies and continuous integration (CI) pipeline to avoid branch conflicts and allow each service to evolve and scale independently of other services.

9.5 Externalized Configuration - Comes from the environment

ALL configuration should come from from the environment in the form of environment variables.

- **Recommended:** The Spring Cloud Config project provides a centralized external configuration management backed optionally by a git repository. It exposes a HTTP REST API for external configuration. The configuration resources map directly to Spring Environment. The Spring Cloud Configuration Server supports encryption of properties and yml files. Consumers of the config server use the client library as a Spring Boot Plugin to bootstrap the environment from the server and POST to `/env /refresh` or `/restart` management endpoints to reload the configuration, rebind configuration properties, reset the refresh scope and restart application context. The `@RefreshScope` annotation enables `@Beans` to be refreshed when the configuration changes. To run your own server use the `spring-cloud-config-server` dependency and `@EnableConfigServer.` On the client side to consume the configuration declare a dependency on the `spring-cloud-config-client and @EnableConfigurationProperties.`
- The Netflix Archaius project is an alternate approach to injecting configuration into the application. Spring Cloud has a Spring Environment Bridge so Archaius can read properties from the Spring Environment. Spring applications should generally not use Archaius directly; however if you do want to use Netflix's Archaius as the single source of all property information then look into the Archaius Spring Adapter ASL 2.0 licensed project.
- Do not package property files with the application. If you *have* to bundle property files in your application then do it in a way that can be overridden by the same property specified as a system or environment property. Understand the order in which Spring Boot loads property sources.
- Do not build the application differently for each environment. This is also one of the edicts of 12 factor app design.
- Environment variables can be set in the application using the command `cf se app_name test.property=test.value`. Configuration can then be retrieved in the application with `System.getenv("TEST_PROPERTY")` or `@Value("${test.property}")String testProperty` using Spring Boot's relaxed binding capability. A less desirable option is to use java system properties to set configuration properties `cf set-env app_name JAVA_OPTS "-Dtest.property=test-value"` as setting a bunch of system properties in one big `JAVA_OPTS` env var is hard to maintain, and the java buildpack hasn't historically handled overlaying `JAVA_OPTS` very well.

10 Design Of Microservices

10.1 Modeling

Service boundaries should be modeled around business capabilities. A good service has a single responsibility, loose coupling and strong cohesion. These characteristics allow the service to change and evolve independently from other services. All the changes to the service are related as a single unit. The key to modeling services is to identify coarse grained bounded contexts aligned around business capabilities. Technical boundaries should NOT be directly converted to service boundaries. A useful guideline when deciding the scope of a service is that each microservice should have its own persistence layer also known as Polyglot Persistence. Multiple microservices all sharing the same backend repository is an anti-pattern since all the services are now coupled to the fragile database schema. Ideally each microservice should persist to its own denormalized single function SQL/NoSQL store.

10.1.1 Entity Modeling

DDD helps discover the top-level architecture and inform about the mechanics and dynamics of the domain that the software needs to replicate [ddd-theGoodAndChallenging]. DDD has number of tactical patterns made of artifacts such as ubiquitous language, bounded contexts, and context mapping; and strategic patterns made up of aspects strictly related to the supporting software architecture and implementation. In terms of return on investment, DDD works best by applying the tactical patterns on the complex areas of your business. Use of sound software development principles based on DDD should be considered in less demanding areas.

Employ the techniques of domain driven design (DDD & iDDD) such as knowledge crunching, making implicit concepts explicit, supple design, applying analysis patterns and refactoring to create a hexagonal architecture consisting of aggregates, entities, value objects and services delimited by a bounded context per domain. Event-Driven Architecture can be derived from this style with each bounded context as a hexagon, publishing and subscribing to events. DDD.

Replacing the entire legacy system with a microservices based architecture is generally a bad idea. Carving out new features for important strategic initiatives as microservices and integrating with the legacy system just enough to enable the new system is the more sensible approach to integration. There are two ways of integrating DDD with a legacy system i.e. 1. Have the legacy system publish some key Domain Events indicating what has occurred in that system. The new microservice Bounded Context listens for and react to these significant Domain Events by carrying out some system behavior. 2. Establish an Open Host Service and a Synchronizing Anti-Corruption layer [strategies-for-dealing-with-legacy , intro-to-ddd, glossary-of-ddd].

10.1.2 Event Modeling

Every state transition within the domain should be modeled as a domain event. Current state is transient. It is a first derivative of the facts. State can be rebuilt by replaying the events. Events are append only and immutable. Events and projections can never be updated or deleted. Event Sourcing is the only model that does not lose information. Event sourcing (ES) allows time travel in software allowing the exploration of alternate reality in terms of business scenarios. Its a model that is very easy to scale up since events can be cached, copied and distributed without any problems. This kind of modeling is very helpful in regulated industries like financials, healthcare where long term audit logs needs to be maintained. In terms of domain modeling we generate an event stream per aggregate. State is determined by replaying only a select number of events within the aggregate. Using rolling snapshots is another optimization in ES systems. Replay events from snapshot forward to determine current state. ES is a functional data storage mechanism requiring a mindshift similar to the shift from object to functional programming. Event sourcing cannot be applied retroactively. It requires the provision of of modeling domain events from the outset and their storage in an event store.

Combined with the command query and responsibility enabling pattern, append only and immutable logs provide a foundation for transactional systems. Scaling queries is a necessity since reads exceed writes generally by an order of magnitude. Queries can also be eventually consistent. Most systems require different types of read models UI, OLAP, lucene - different projections to work well. A new model can be added if needed. [CQRS-Event-Sourcing DDD-CQRS].

Entity modeling is a natural fit for certain classes of functionality like search, reporting and audit logging. The ability to build features retroactively makes this modeling paradigm very powerful albeit a less explored area of microservices architecture. Event driven eventual consistency architecture is not suitable for applications especially those that need exact information at all times.

10.1.3 Modeling Tools

Tools like Component Responsibility Collaborator Cards (CRC) cards or good old sticky notes can be used to write the job descriptions for each service and design the dynamics of service collaboration. Give each person a CRC card and role play out the conversations between the hexagonal microservices to flush out any issues with component interaction. Using CRC cards in the ubiquitous language of your domain helps business users and other domain experts make sense of the system. For extra credit some modelers have known to directly write JSON file and then visualize service models and relationships using Neo4J.

10.2 Granularity

A tricky question when it comes to microservices is deciding the granularity of the microservice. This is a delicate tradeoff that is affected by various concerns. If the service API is consumed directly by a client UI then the service APIs have to be necessarily fine grained. As services become more fine grained there is increased chattiness and higher communication, integration and operational overhead. If microservices are teased out and integrated in a legacy environment then it makes sense to slowly evolve each microservice and integrate with the bigger system. This is a choice between micro and macro that every organization will need to make on its own based on its appetite for risk and desired velocity of feature delivery.

10.3 Packaging

A service should be packaged as a war if

- It has dependency on external systems, legacy systems, mainframe systems, SMTP services, messaging queues, sys log drains etc.,
- It acts as a data provider for external systems
- It has scalability needs different to that of its consumers
- It is reused across multiple business capabilities. This can introduce potential performance and XA transaction issues leading to design for compensations. Decision to be based on carefully weighing the pros & cons against jar deployment

A service should be packaged as a shared library or an utility jar if

- It provides capabilities which are non business specific that serve as building blocks and are atomic in nature
- There is potential to reuse across business services exists
- It deals with basic capabilities like string manipulation, date manipulation, logging etc.

- Addresses cross cutting concerns like auditing, notification, string manipulation, date manipulation etc.

10.4 Partitioning

Partitioning your domain and problem space into microservices is more an art than a science. The approaches below provide different ways of slicing and dicing the application space into microservices. It is always easier to merge services instead of splitting them apart later. So when in doubt create a new service and merge and disintegrate later based on usage. The heuristics for determining service boundaries depend on whether a project is Brownfield or Greenfield.

10.4.1 Domain Driven Design (DDD)

Domain driven design and its influence on the implementation of a complex core domain forms the foundation of building microservices. DDD divides up a large system into Bounded Contexts, each of which can have a unified model and are explicit about their interrelationships. A model acts as the foundation for the design of the software itself i.e. how it's broken down into services. It provides a way to set logical and physical boundaries in terms of team organization, operations, usage within specific parts of the application and physical manifestations such as code bases and database schemas. To delineate bounded contexts pay attention to where the definitions of the words and language changes across contexts i.e. an order in the product design domain means something totally different in the product shipping domain. The seminal book on deriving bounded contexts is Domain Driven Design by Eric Evans. An excellent companion book to the DDD is Implementing DDD that implements many of the concepts described by Eric and updates them for even driven architecture.

10.4.2 Breaking the monolith

Strategies for carving out microservices and evolving a microservices based system architecture are articulated below -

1. **Databases/Data Stores -** Separate databases before separating services. Refactor existing databases using database schema visualization tools like SchemaSpy. Denormalize into one data source per table or materialized view. Break foreign key relationships and shared data across tables by introducing new APIs or services that encapsulate the data access to that particular service domain. Recognize seams of the system with database smells like multi-purpose table/column, redundant data, tables with large number of columns & rows, smart columns, lack of constraints and chiefly fear of change.
2. **Existing Afferent (Inbound) and Efferent (Outbound) Coupling -** Partition microservices by inspecting code with static code analysis tools like JDepend, XRay, D3.js, Structure 101, etc., to discover god objects and coupling between packages. Examine the most complex code in the application and determine if its accidental or essential complexity using metrics like cyclomatic complexity and start refactoring the most complex methods and classes. Carve out behaviors studying the interaction between god objects and coupling between packages/classes. Start with parts that hurt or change the most. For instance if the monolith talks to a third party legacy application or service build a facade microservice to help with the boundary or change of language in the domain. Employ evolutionary design to break monolith into microservices one at a time with A/B testing in production and validation before removing the equivalent

function from the monolith. When starting from an existing domain be cautious of the tight coupling and tight feedback loop that exists between entities in the domain.

3. **Transaction Boundaries** - Separate transaction boundaries into their own services. Use transaction less coordination between services. Decompose data into functional groups, relax consistency constraints across groups and partition the busiest groups across multiple databases [base-over-acid]. Build services for eventual consistency and have a reversal process to deal with mistakes. [banks-are-base-not-acid]. Embrace independent transaction scopes across services, avoid distributed transactions by refactoring the persistence model in such a way that state changes happen inside the boundaries of a single aggregate entity [ddd_aggregate] [patHellandLifeBeyondDistribTrans].
4. **Modes of communication** - Synchronous and Asynchronous is a good way of organizing microservices based on the inter service communication protocol stack.
5. **Team Organization and Structure -** Take advantage of Conway's law to carve out boundaries for new services based on existing team structure.

10.4.3 Implementing Microservices from scratch

Patterns for genesis of a microservices in a greenfield environment in in order of importance -

6. **Use Cases/User Journeys/Business Processes** - Derive services from existing user flows in consultation with domain experts. For example a login service that implements login or a partitioned e-commerce application that has a Shipping service that's responsible for shipping complete orders. This naturally leads to further partitioning based on verbs and nouns.
7. **Verbs/Operations** - For example a login service that implements the login use case. One "verb" per single function micro-service.
8. **Nouns/Resources** - For example an e-commerce system that has an Inventory service that keeps tracks whether products are in stock and responsible for all operations on inventory items. This kind of service is responsible for all operations that operate on entities/resources of a given type.
9. **Separated models for reading and writing**- For example a microservice system built following the CQRS pattern. CQRS allows read and write services to be scaled independently if they are operating on different data stores. The service that reads the data can be separated from the service that is writing the data. This separation forces a break of the mental retardation that because the two use the same data they should also use the same data model. CQRS and event sourcing go hand-in-hand.
10. **Stability and Point of Evolution** - Group and develop microservices at a similar point in their evolution. New code should be separated stable business critical microservices. Have two classes of microservices and once the dust settles consolidate. Decouple parts of your application that change at different rates.

10.4.4 Strategies For Introduction of Microservices in Brownfield enterprise systems

10.4.4.1 Co-existence & Integration

If interaction is needed with legacy apps/systems then build a facade/anti-corruption layer that talks to the new apps on the Paas and translates to the the language of the legacy app. An AnticorruptionLayer translates to/from that model and yours. The legacy systems can be deployed using traditional IaaS templating pattern engine technologies including tools like chef and puppet whereas the cloud native apps can be setup using automation (BOSH or PCF).

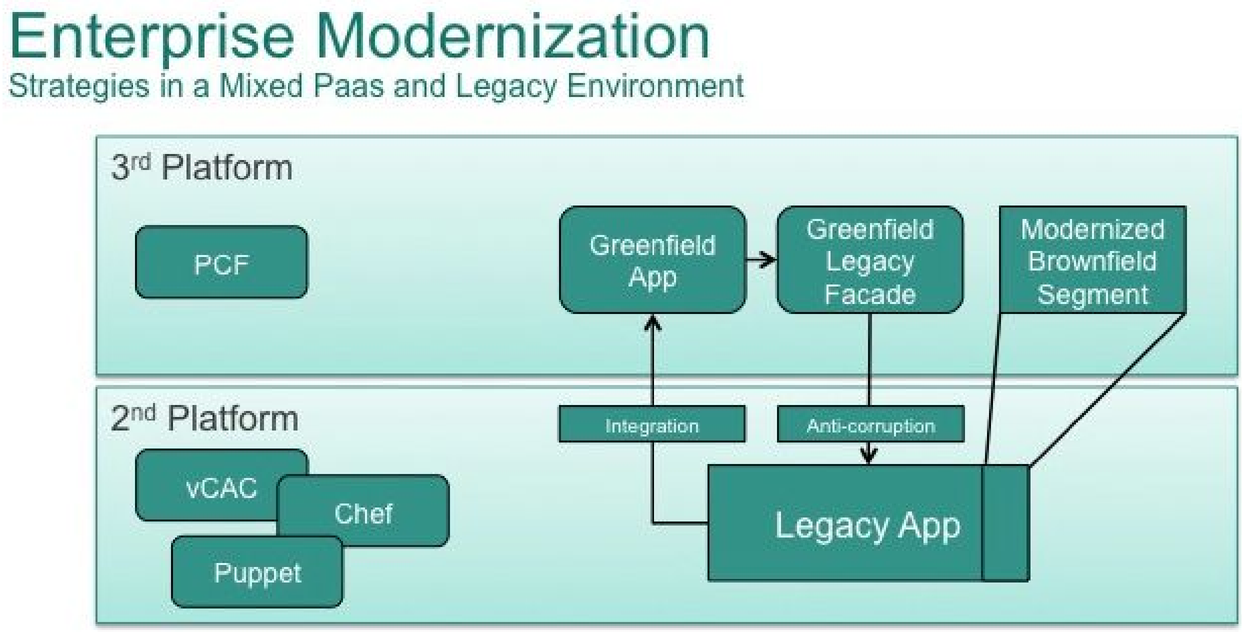
10.4.4.2 Technical Transition

Gradually migrate existing function to newer cloud style native microservices based apps. The reasons for breaking the monolith and the specific function to cherry pick for the separation should be well understood. Reasons being 1. scale (a particular part of monolith has different scaling characteristics than the whole) 2. encapsulation (a particular part of the megalith undergoes very frequent change) 3. Cost (Moving code off the mainframe consuming MIPS to the tomcat java stack) 4. Better Team Organization (Use Conway's law to assign code ownership) 5. Technical Debt (choose to attack and separate code that smells the most). More strategies on how-to break a monolith are [2]. Thereafter reduce the risk of deployment, by dark launching, using feature flags to turn the function on/off in the monolith while introducing the same feature concurrently via the cloud native app. The goal is to iterate and ultimately entirely move off the older monolithic code base.

10.4.4.3 Strangler Pattern

This pattern involves gradual replacement of a system by implementing new features in a new application that is loosely coupled to the existing system, porting existing functionality from the original application only where necessary. Over time, the old application is "strangled"— just like a tree enveloped by a tropical strangler fig. The key to implementing the strangler pattern is not attempting port existing functionality unless it is to support a business process change [agile-approach-legacy-system] and delivering something fast designed for testability and deployability, architected to run on a PaaS.

It takes longer to do a replacement incrementally compared to a hypothetical big bang re-architecture delivering the same functionality. However, since a strangler application delivers customer value from early on, evolves in response to changing customer needs, and can advance at its own rate, it is almost always to be preferred [Lean-Enterprise]. Incremental change eventually wins.



Enterprise Modernization

Strategies in a Mixed Paas and Legacy Environment

1. Left side is infrastructure tooling view
2. Right-hand is app view, with migration-related components
3. Greenfield apps, of course, are cloud native
4. For greenfield dependencies on legacy app (valuable services to leverage), try to use a modern, cloud-native facade/gateway/proxy. This facade is the modern API for the service that new apps use to access legacy functionality. The legacy app is encapsulated beneath this facade and can be refactored/replaced over time.
5. For segments in legacy app identified for modernization (delineated by domain and strive to decouple volatility of domains), move the modernized segments into PaaS.
6. Integration from legacy to greenfield should prefer dedicated "integration apps". These small components in the diagram are apps that exist only to implement integration patterns, and are as external as possible to legacy app. An example of external such app would be a spring integration app deployed along-side legacy. A not-really-external example would be a set of SQL views acting as an anti-corruption layer for legacy facade.

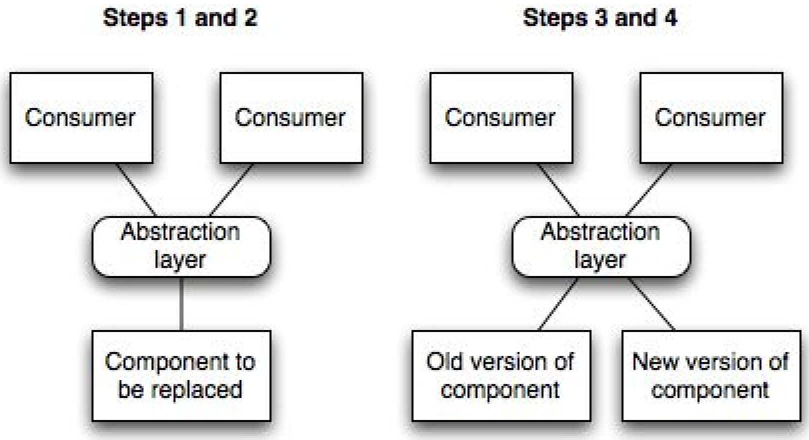
Branch Abstraction Pattern

Branch by abstraction enables rapid deployment with feature development that requires large changes to the codebase. For example, consider the delicate issue of migrating data from an existing store to a new one. This can be broken down as follows: [facebook-development]

1. Encapsulate access to the data in an appropriate data type.
2. Modify the implementation to store data in both the old and the new stores.
3. Bulk migrate existing data from the old store to the new store. This is done in the background in parallel to writing new data to both stores.
4. Modify the implementation to read from both stores and compare the obtained data.
5. When convinced that the new store is operating as intended, switch to using the new store exclusively (the old store may be maintained for some time to safeg against unforeseen problems).

Branch by abstraction involves making large-scale changes to your system incrementally as follows:

1. Create an abstraction over the part of the system that you need to change.
2. Refactor the rest of the system to use the abstraction layer.
3. Create new classes in your new implementation, and have your abstraction layer delegate to the old or the new classes as required.
4. Remove the old implementation.
5. Rinse and repeat the previous two steps, shipping your system in the meantime if desired.
6. Once the old implementation has been completely replaced, you can remove the abstraction layer if you like.



The strangler application pattern operates at a higher level of abstraction than branch by abstraction, which is for incrementally changing the implementation of a component of your system instead of completely replacing the older implementation.

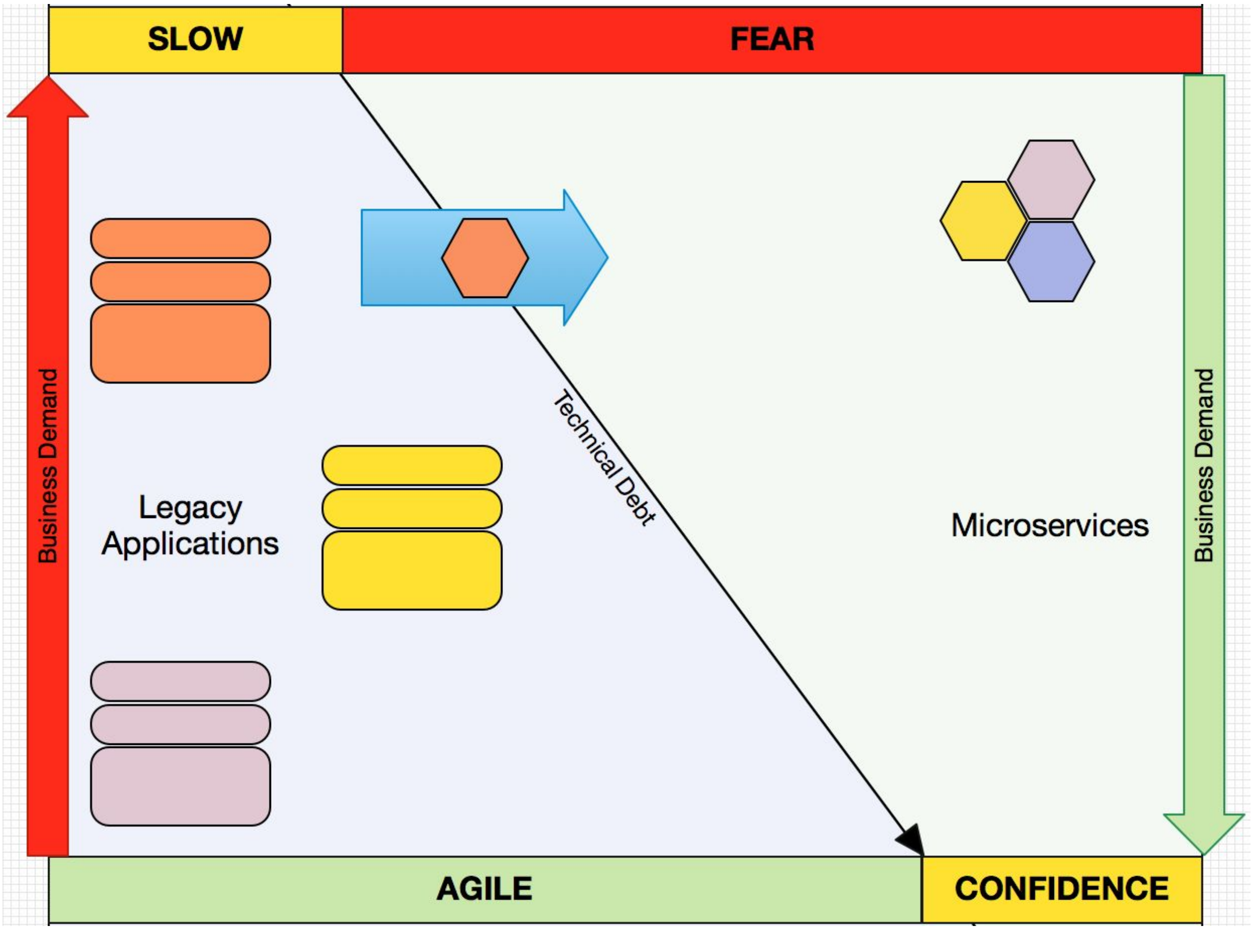
10.4.4.4 Social Transition

The biggest challenge isn't technical, but organizational. Business demand drives application development while technical debt impedes it. Further, as demand mounts, debt compounds. It takes an active, concerted effort to eliminate debt. The explicit focus should be on eliminating debt as enterprise systems are migrated to the PaaS.

Traditional IT is a victim of their own success; they have active development on many applications, and they support considerable traffic. In addition, they have typical symptoms of a debt problem: incomplete/absent test harness, fragile and incohesive components and dependencies, etc. leaving them with a lack of confidence in their ability to make needed changes. Enterprise architects have little confidence in their ability to isolate service boundaries and successfully re-implement portions of an application while keeping the lights on -- let alone supporting continued business demand.

If you approach this problem with a purely technical solution, it might miss the mark. It is critical that the solution and process supports existing operations while supporting business demand

AND also working down debt. Fall back to a roadmap pitch based on concrete behaviors and drill down a little on what a given application migration might look like. Lead with a successful migration/implementation of one/many API services to develop proficiency and demonstrate what is possible. One of the biggest concerns is existing applications not being a good fit for PCF -- this is the area for transformation and development.



10.5 Inter-service/Inter-process communication

Due to the inherent instability of the network and the fallacies of distributed computing any non trivial microservices system has fragmented status (all up, all down, mostly one up, all up apart from 1, etc.,). Building resiliency and stability into microservices is an essential requirement for operating a microservices system. Fortunately the resiliency patterns have been documented well in ReleaseIT and implemented in some form by open source projects like Hystrix and Akka.

There are in general, three approaches for communications between microservices. The first-line choice is often to use HTTP to make RESTful synchronous or asynchronous calls. A second more complicated choice is using an event driven architecture with messaging. A third approach is a hybrid combining REST synchronous request/response interaction with event driven architecture. It is **recommended** to start with REST over HTTP and as the needs of the system evolve migrate to an asynchronous event driven model. If possible avoid RPC-mechanisms or shared serialization protocols to avoid coupling. Microservices architectures promotes coupling from application to integration architecture.

Consuming the REST JSON API via Javascript makes sense for edge and public services; however for internal and highly performant consumption of microservices an alternative approach is to use binary transport protocols like google protocol buffers, Thrift, Avro and Simple Binary Encoding. Use of binary protocols leads to the emergence of client libraries for services. These client side reference libraries should be owned & published by same team that owns the microservice.

10.5.1 HTTP Synchronous Request/Response Style Interaction

When using JSON payloads with REST APIs over HTTP it is a best practice to build resilience into the service interaction by employing design patterns like fail-fast, connection pools, thread pools, service-groups, timeouts, circuit breaker and bulkheads chronicled in Release It. Netflix Hystrix technology is an implementation of the command design pattern that bakes in timeouts, circuit breakers and bulkheads to avoid cascading failures.

- The **recommended** way for building resilience into your REST service API interaction is to use the spring cloud project. Spring Cloud has historically contained Platform-as-a-Service connectors that let you consume services - databases, message queues, etc. - from within a PaaS environment. Spring Cloud has expanded scope to define and provide software to better enable modern, cloud-y architectures, like the microservices that Netflix builds atop Spring Boot and their own, open-source stack. Spring Cloud aims to provide solutions for emergent patterns in large scale, often cloud-based applications. Spring Cloud offers the following features.

  Spring Cloud Netflix

  - Dynamic Routing and API gateway capabilities with Zuul. Zuul is covered in detail in section 10.7 API Gateway.
  - Microservice registration and microservice discovery with Eureka. Eureka is covered in detail in section 10.8.2 Spring Cloud - Netflix Eureka Integration.
  - Client side load balancing with Ribbon. Ribbon is integrated with Hystrix and uses Eureka to find service instances based on configured policies like round-robin, best-available and random.
  - Failures can be managed in a declarative way with the @HystrixCommand annotation on a method. For a spring boot application integrating Hystrix is as easy as adding the `spring-cloud-starter-hystrix` dependency and @EnableHystrix. Hystrix commands can be executed in synchronous (returns immediately), asynchronous (returns Future) and reactive (returns Observable) modes. Circuit breaker metrics for Hystrix service API calls can be tracked with with the `/hystrix.stream` event stream. The Turbine application aggregates /hystrix.stream endpoints into a combined /turbine.stream for use by the Hystrix dashboard. Running Turbine is as simple as annotating your main class with the @EnableTurbine annotation.
  - Archaius Integration with Spring Environment. Again see 9 Configuration - Comes from the environment.

- Feign a declarative REST client. Feign creates a dynamic implementation of an interface decorated with JAX-RS or @RequestMapping Spring MVC annotations thereby allowing rest systems to be addressed as java interfaces. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

Spring Cloud Config

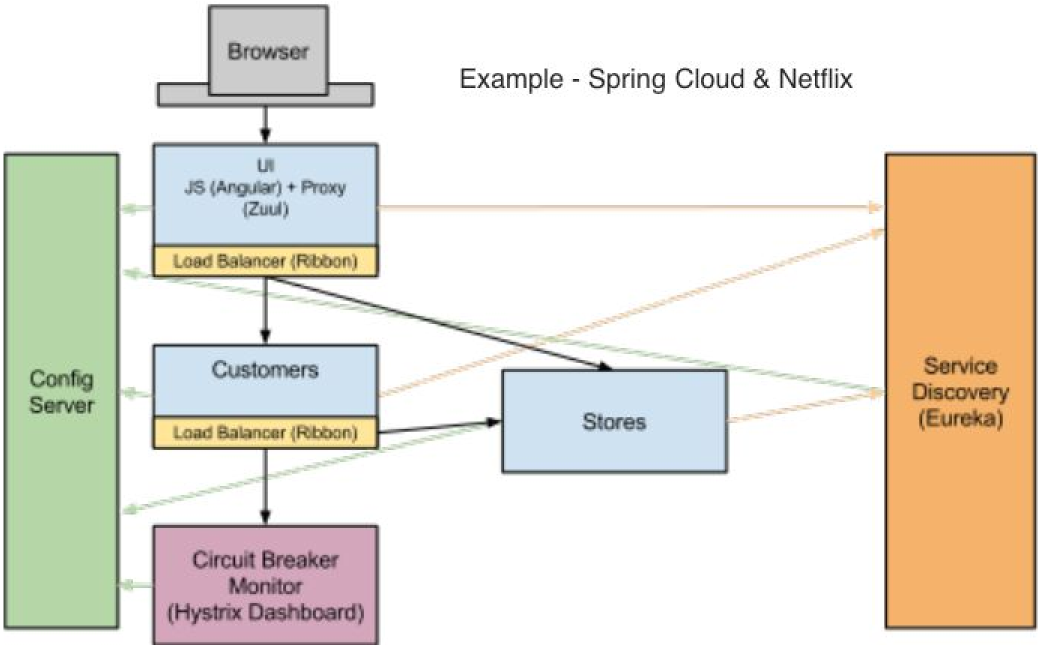
- External configuration management backed by a git repository
- Refreshable configuration. Covered extensively in 9 Configuration - Comes from the environment

Spring Cloud Bus

- Event bus implemented as a lightweight AMQP message broker used to broadcast state changes or other management instructions. The bus currently supports sending messages to all nodes listening or all nodes for a particular service that satisfy a particular criteria. There are currently two endpoints implemented `/bus/env` and `/bus/refresh`.

Spring Cloud Security

- Set of primitives for building secure applications. If you deploy apps on Cloud Foundry that use HTTP Basic Security, then the best way to configure security is through service credentials, e. g. in the URI of the user provided service, since then it doesn't even need to be in a config file. If you use another form of security you might need to provide a RestTemplate to the ConfigServicePropertySourceLocator.
- It is not feasible to pass the username and password of the user from one microservice to another. OAuth2 comes to the rescue wherein the user authenticates once with the authentication server and can delegate authorization to resources. Integrating an application with cloud foundry using OAuth2 is covered in 10.10 Security.



- Leverage Spring Integration’s RequestHandlerRetryAdvice and RequestHandlerCircuitBreaker advice showcased here or
- Embed Netflix Hystrix library into a Spring application. Use the Netflix OSS library Hystrix to build fault tolerance into service calls. Hystrix should also be used to encapsulate interaction with cloud foundry services like REDIS and MySQL.
- Flux Capacitor - Flux Capacitor is a Java-based, cloud-native, reference architecture using many Netflix Open Source projects

Spring Session a recently announced project provides a common infrastructure for managing sessions. The primary benefit is uniform access to sessions from any environments like web, messaging, infrastructure etc,. Other benefits include a pluggable strategy for determining session id and a way to keep HTTP sessions alive when a WebSocket is active. Instead of configuring and relying on session management by the container and the java buildpack Spring session enables the application to create and access the session life cycle.

10.5.2 Asynchronous Integration of Services with Events

Microservices emit events conveying the history of the domain. These events are subscribed to and processed by all the other microservices that need to take action based on the event. This kind of architecture leads to decoupled services that prefer choreography over explicit orchestration. New features can be added easily without impacting existing domain model or business logic. Implementing microservice system with events requires the use of event sourcing and event collaboration enterprise application integration patterns :

10.5.2.1 Event Collaboration

Instead of components making requests when they need something, components raise events when things change. Other components then listen to events and react appropriately. Event Collaboration leads to some very different ways of thinking about parts need to think about their interaction with other parts. With Event Collaboration you can easily add new components to a system without existing components needing to know anything about the new arrivals. As long as the newcomers listen to the events, they can collaborate.

10.5.2.2 Event Sourcing

Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not only can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes. Complementary to the capture of business meaningful events, the technique has positive implications for analytics in driving greater customer insight.

There are other lesser known patterns as well that build on event sourcing like Retroactive Event that allows a system to automatically correct the results of many incorrect events and Parallel Model that captures all the nuances of the application in a past or imagined state by allowing us to easily take our entire information store and represent it as it was in the past or in some alternative past, present, or future. To build a Parallel Model the system has to be designed with Event Sourcing, a way to process events in a model and project it into an alternate state.

Recommendations - Designing a system entirely based on events as the single source of truth is difficult. A couple of frameworks that can help are the Axon framework which is a am implementation of CQRS available in Java and the EventStore framework. A honorable mention goes out to the lambda architecture which provides a generic, scalable and fault-tolerant big data processing architecture.

10.5.2.3 Reactive Applications

A new architectural style called Reactive applications has emerged to allow developers to build systems that are event-driven, scalable, responsive and resilient. The key building blocks for event driven reactive applications are asynchronous sending of events, non-blocking, decoupling of event generation and processing, isolation, observable models, event streams and stateful clients. The entire solution needs to be asynchronous from the top(browser) to the bottom (web layer & service components). For a full treatment of this topic please read the Reactive Manifesto.

Reactive Streams is an initiative to provide a **standard** for asynchronous stream processing with non-blocking back pressure on the JVM.The main goal of Reactive Streams is to govern the exchange of stream data across an asynchronous boundary—think passing elements on to another thread or thread-pool—while ensuring that the receiving side is not forced to buffer arbitrary amounts of data. In other words, backpressure is an integral part of this model in order to allow the queues which mediate between threads to be bounded. The benefits of asynchronous processing would be negated if the communication of backpressure were synchronous.

A number of frameworks implement Reactive Streams -

1. **Recommended:** The Spring Reactor project provides a foundational framework for applications that need high throughput when performing reasonably small chunks of stateless, asynchronous processing is very promising.
2. Typesafe platform for reactive application development built on top of the Play web framework, the Akka event-driven runtime, and the Scala programming language.
3. RxJava - Reactive Extensions for the JVM by Netflix - a library for composing asynchronous and event based programs using observable sequences. *Note* - Spring Reactor also provides support for RxJava.

The fundamental underpinning of reactive event-driven applications is a reliable messaging passing protocol like AMQP. Other technologies can be used for event generation and processing ranging from Spring in-memory application events, to in-memory data grids like Gemfire or Redis. A combination of the Spring JMS, Spring AMQP and Spring Integration spring projects is typically used to implement EAI patterns.

10.5.3 Hybrid Mode

Service communication can also be modeled with both events and synchronous HTTP. One way to decide on the mode of communication is to use HTTP request/response style with internal components and services while relying on events to interact with external dependencies. The messaging channel then serves as a shock absorber for any network events or service outages.

In most enterprise systems there is mixture of synchronous and asynchronous model of interaction between services. Synchronous interaction with a website (content router and downstream services) leads to generation of behavioral data such as clicks, purchases, recommendations, wish lists etc., which is later aggregated and analyzed by asynchronous batch jobs running mapreduce analytics, distributing the results to the back-end tier further influencing browsing and purchase habits.

A single model does not work well for everything - reporting, searching and transactional behaviours. This illustrates the need for a harmonious relationship between the sync and async parts of the microservices system.

10.6 Binding Cloud Foundry Services to a Microservices Application

- Microservice applications should declare an explicit dependency on pre-configured Cloud Foundry services. Configure Service Connections for Spring.
- Use spring cloud directly and programmatically to connect and configure the connection to Cloud Foundry services that are bound to the application. Datasource properties such as connection pooling can be passed in into the `cloud.getSingletonServiceConnector` method. Please see the code in the blog post that configures the `DataSource` with a connection pool of size 10.
- If Spring cloud is not explicitly used and if the application uses Spring, then the Java Buildpack causes an application to be automatically reconfigured to work with configured cloud services. It auto-configures the Spring application context for JDBC, Hibernate, JPA, MongoDB, RabbitMQ, and Redis beans that match bound services. This Spring auto-reconfiguration framework leverages Spring Cloud under the covers.
- Spring Data JPA or Hibernate can be used to encapsulate these cloud datasources from the business logic.
- For single data source spring boot application there is no need to declare any beans for a data source, entity manager factory or transaction manager. Spring boot will create those beans and wire them when the application configuration class is annotated with @EnableAutoConfiguration.
- When binding multiple databases of the same type from an application we can disable specific spring boot auto configuration features for DataSource, HibernateJpa and DataSourceTransactionManager and manually wire the data sources to the respective services.

10.7 API Gateway aka Software Reverse Proxy for Microservices

The most convenient way to route to the downstream microservice REST APIs is by employing a front end controller pattern with Spring MVC and other projects that build on top of it to provide REST capabilities like Spring Data REST, and Spring HATEOAS. Depending on how the user interface is structured you may have to implement a rest controller service for orchestrating and aggregating API calls to downstream services. The API gateway must be able to deal with service versioning and routing to different versions of the same service.

The UI for a microservice system can be composed on the server side by using a composite front end portal like pattern, integrating the services with a dumb UI or on the client side with the help of JavaScript libraries like AngularJS, Backbone.js, single page application's etc,. If the services in question are designed to be REST level 3 compliant with Spring HATEOAS, then the controller can be implemented to NOT have explicit hardcoded dependencies on URIs, rather just have a dependency on named resources and then can self discover the REST API using relational links.

As the number of microservices increases and the governance, monitoring, security, and availability concerns increase, Spring MVC based gateway will need to be replaced with a full featured API gateway software component that provides quota management, rate limiting, Restful API analytics, OAuth2 security, management API etc,.The two options that come to mind are TYK and Netflix Zuul. Zuul is a JVM based router and filter similar to httpd, nginx, or CF go router. Zuul supports fully programmable rules and filters on any JVM language. Zuul can be deployed as an application to Cloud Foundry. Tyk on the other hand is better suited as a service in Pivotal CF since it is a specialized reverse proxy based on nginx and redis. A third option is roll your own gateway with features like rate limiting and analytics using Rabbit for async messages and background workers to process the data.

The **recommended** way of designing and implementing a microservice API gateway is to configure a Spring Cloud Zuul Proxy by annotating a Spring Boot main class with `@EnableZuulProxy` and configure routing rules which use `Hystrix → Ribbon → Eureka` to forward local calls to `/proxy/*` to the appropriate service. When coupled with a configuration server routing rules are stored in the config server under `zuul.proxy.route.<APP_NAME>`. Spring cloud support for Netflix Zuul provides automatic registration of Zuul filters, and a simple convention over configuration approach to reverse proxy creation.

The entire netflix software stack is transitioning to reactive inter process communication stack. The entails use of RxNetty instead of Tomcat and leveraging the Reactive RxJava extensions in all services and service client libraries. The Netflix API makes the entire service layer asynchronous so all service methods return an Observable<T>. Making all return types Observable combined with a functional programming style frees up the service layer implementation to safely use concurrency. In such an architecture, Hystrix is primarily used for throttling and not for achieving asynchronicity. The Netflix client side stack bakes in RxJava for making multiple concurrent calls to services when writing service clients. If you prefer sticking to the core JDK, then the Java 8 CompletableFuture promise support and Jersey Managed async provides similar capabilities. Spring enables these programming styles with it support for Hystrix, Jersey and Java 8.

10.8 Service discovery

How do clients determine the IP and port for a service that exist on multiple hosts?

10.8.1 User Managed Services

RESTful HTTP Microservices should be wired to one another as CF user-managed services. All services including Cloud Foundry services can then be discovered using the VCAP_SERVICES environment variable. Cloud Foundry services are preconfigured with the microservices application and specified in the application manifest.yml. For high availability, each microservice application should have at least two instances running in production. All HTTP REST API calls between microservices will go through the frontend CF go router.

10.8.2 Spring Cloud - Netflix Eureka Integration

Discover services piggy backing on the the spring cloud support for Eureka. Eureka acts as the service registration server. Eureka can be configured to be highly available i.e. multi availability zone and region aware in AWS terms. The **recommended** approach for service discovery is to enable service clients to discover services registered with the Eureka Server using the @EnableEurekaClient annotation. Eureka instances can be registered and clients can discover the instances using Spring-managed beans. The Eureka server is a spring boot application.

10.8.3 External Data Stores

Another solution for service discovery is to use a service like ZooKeeper or etcd to register the URL for a service. In this model the services register themselves under a known path and a namespace in Zookeeper. Each service also maintains a list of dependencies in the environment. This technique entails the use of a smart service client that abstracts the use of Zookeeper to lookup the service endpoint. For an implementation of this pattern with Spring take a look at the 4finance service discovery module.

10.9 Rest API

10.9.1 Granularity

Services should not expose fine grained or CRUD based REST APIs. Tolerant reader, consumer driven contracts and a coarse grained API allow the service provider to independently evolve the REST API resulting in zero downtime and less chattier interaction between services. If clients have to call 10 or 20 microservices to render a response to render consider aggregating all these calls in a API service and have the client call the API service.

10.9.2 Linked Data

Use the Spring HATEOAS project to implement a rich level 3 hypermedia REST API for the service. Hypermedia representation enables service discovery and loose coupling by eliminating the need to pass hard coded REST URIs to other services. User agents and other clients can use tools like JSON browser plugins and Spring Rest Shell for interacting with HATEOAS compliant REST resources. Taking advantage of true hypermedia links allows programmatic API management and eliminates manual discovery which does not scale as the number of services in the system increases. Refactoring service APIs becomes easy since links can be changed to the new location or redirected. Providing these breadcrumbs through URLs builds enables maintainability by providing flexibility to change URLs.

10.9.3 Documentation

Document your Microservices Rest APIs with Swagger. Swagger™ is a specification and complete framework implementation for describing, producing, consuming, and visualizing RESTful web services. Microservices service templates should bake in swagger integration taking advantage of the swagger integration with Spring projects like Swagger-springmvc, swagger-spring-boot and swagger4spring-web.

10.9.4 Design

Design the API around the resources that are based on the business processes and domain events. For example, to update an existing bank customer's address, a POST request can be made to “ChangeOfAddress” resource. This “ChangeOfAddress” resource can capture the complete address change event data such as who changed it, what was changed, etc.,.

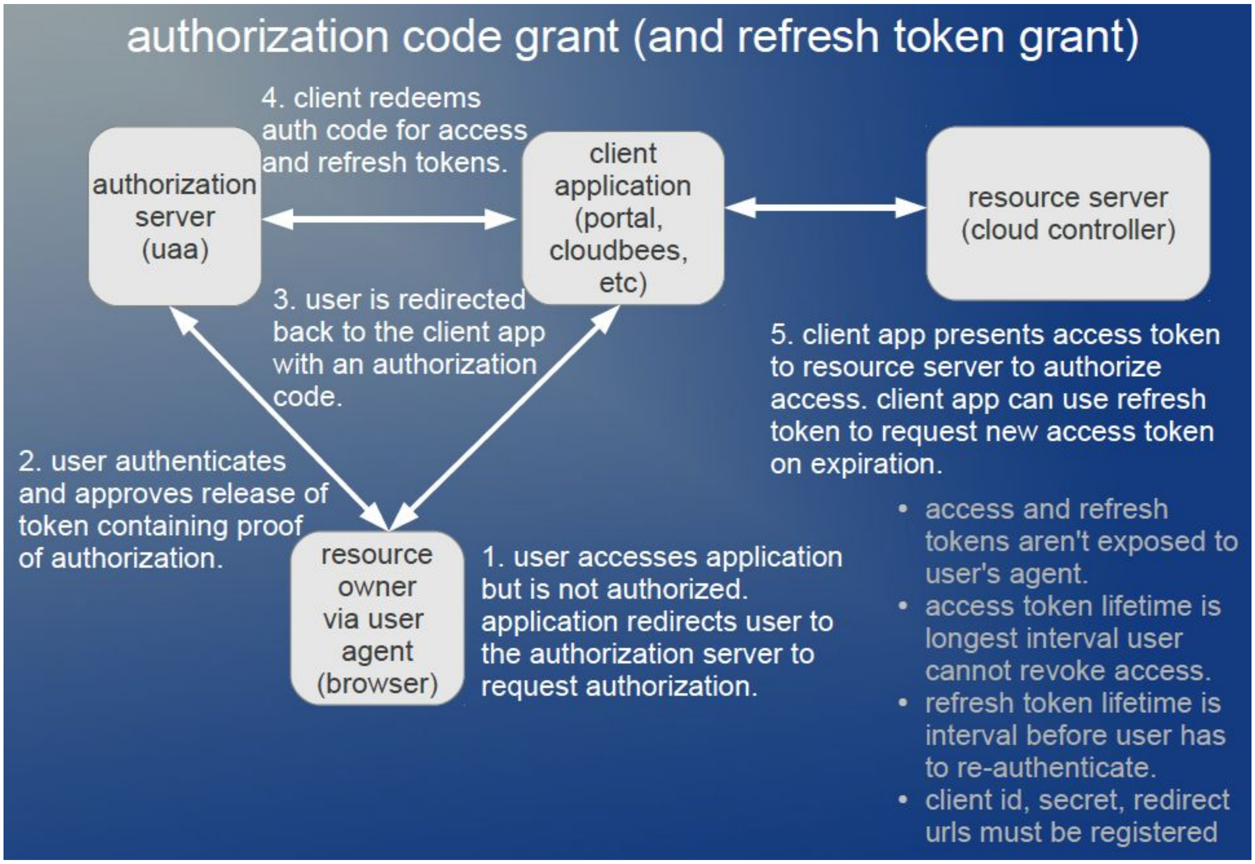
10.10 Security

Start with Spring Boot’s built-in faculties for quickly bootstrapping basic authentication for HTTP REST APIs and resources. If LDAP is needed for end user authentication then just plug spring security into the application and be done. When your authentication needs outgrow basic auth, microservice applications can leverage the full capabilities of Spring Security OAuth2 support for authentication and delegated authorization. Possible integration points include CF UAA LDAP integration or CF UAA for user authentication and authorized delegation for back-end services, and Single-Sign-On for your requirements. Reference.

For securing interactions between microservices and clients you can leverage the Cloud Foundry UAA component for Centralized Identity Management or as a central authentication service. There are a couple of OAuth2 grant types that are pertinent i.e. an Authorization Grant and a Client Credentials grant. Each microservice application will be a client application in OAuth2 terms and will need to be pre-registered with the UAA.

Although it is technically possible for customers to use the CF-deployed UAA for app authentication, it is **not supported or recommende**d. Downloading and setting up the UAAC ruby client for registering OAuth2 clients is a lot of trouble on windows. The incantations of uaac you have to deal with are not obvious either. Since the CF deployed UAA is a critical component of the PaaS, someone using uaac to set up app auth could potentially mess up platform auth, which would be a bad thing. There is little overlap between users of the platform i.e. developers pushing and managing applications and end-users of the apps running on the platform.

We **recommend** deploying the CF UAA component separately as an app/service on CF.UAA is a Spring-based Java web application that runs on Tomcat. It is fully configurable through environment variables and a YAML file. UAA’s primary role is that of an OAuth2 provider that can issue tokens to applications. It can also act as a login server and authenticate users and can manage user accounts and OAuth2 clients through a HTTP API. Samples use of UAA is demonstrated by these applications [uaa-samples, securing-soundbreak].



authorization code grant (and refresh token grant)

authorization server (uaa)

client application (portal, cloudbees, etc)

resource server (cloud controller)

resource owner via user agent (browser)

1. user accesses application but is not authorized. application redirects user to the authorization server to request authorization.

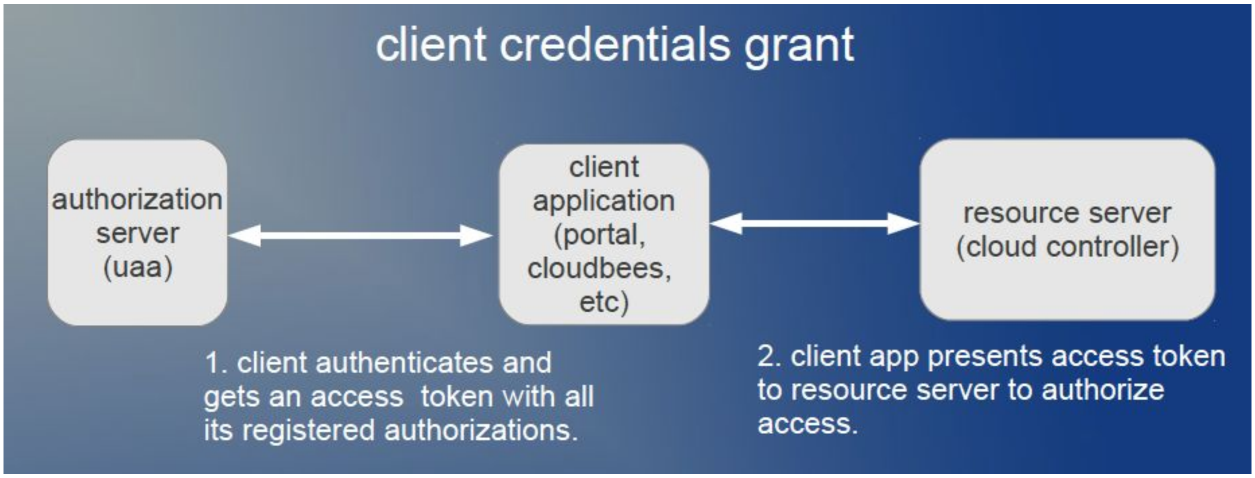
2. user authenticates and approves release of token containing proof of authorization.

3. user is redirected back to the client app with an authorization code.

4. client redeems auth code for access and refresh tokens.

5. client app presents access token to resource server to authorize access. client app can use refresh token to request new access token on expiration.

- access and refresh tokens aren't exposed to user's agent.
- access token lifetime is longest interval user cannot revoke access.
- refresh token lifetime is interval before user has to re-authenticate.
- client id, secret, redirect urls must be registered



client credentials grant

authorization server (uaa)

client application (portal, cloudbees, etc)

resource server (cloud controller)

1. client authenticates and gets an access token with all its registered authorizations.

2. client app presents access token to resource server to authorize access.

In OAuth2 terms the UAA serves as both the resource and the authorization server. Within the UAA and it's UI front end the login-server, users can originate from multiple sources including LDAP or SAML. Basically you can use the Cloud Foundry UAA OAuth2 Provider as an an authentication and authorization protocol for lightweight services. In the simpler case where services never have to act on behalf of users then using OAuth2 is probably overkill and using Basic authentication over SSL based on shared secrets (username and password) and a shared user account database is good enough.

- http://tutorials.jenkov.com/oauth2/index.html

- http://blog.pivotal.io/cloud-foundry-pivotal/products/how-to-integrate-an-application-with-cloud-foundry-using-oauth2
- http://blog.pivotal.io/cloud-foundry-pivotal/products/securing-restful-web-services-with-oauth2
- http://blog.pivotal.io/cloud-foundry-pivotal/news-2/cloud-foundry-uaa-onwards-and-upwards-into-the-cloud

Here is an example of how a CF SSO service can be used to secure a web application / microservice. A Single Sign On service will be a policy-based authentication service that provides an easy to embed single sign-on capability for web applications. The service will supports an OAuth 2.0 authorization code flow with protected resources. Java code for authenticating with a CF SSO service:

- https://hub.jazz.net/project/bluemixsso/SingleSignOnSampleClient/overview
- https://github.com/spring-projects/spring-security-oauth/tree/master/samples
- https://github.com/cloudfoundry/uaa/tree/master/samples

In the above example, instead of a separate Oauth2 provider setup for the SSO service private PaaS developers and operators can leverage the Single Sign On capability provided by the UAA's built-in OAuth2 provider.

10.11 Persistence

As your system becomes more distributed prefer **B**asic **A**vailability **S**oft State **E**ventual Consistency BASE over ACID. Chose between Consistency and Availability since functional partitioning is a MUST for scalability. In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability [Base An ACID Alternative] . Once data is partitioned by functionality, the data constraints move from the database to the application. BASE provides a way of thinking about managing resources, decoupling operations and performing them in turn resulting in improved availability and scale at the cost of consistency.

Use a mixture of database technologies behind REST data access layers. Examine the results of the Jepsen torture tests for common distributed systems before choosing a backend. Generally implement one backend for a microservices or a group of microservices in a bounded context. There will be a lot of independent loosely coupled data stores. You may need to run a process like a Master Data Management that cleans up the data stores and makes them consistent. The separation of monolithic normalized persistence model to a denormalized model needs to be done incrementally. Start with breaking off materialized views and then proceed gradually to separate other parts of the persistence model. For best practices on database refactoring go through Refactoring Databases.

11 Governance & Social Engineering

Microservices have crossed the chasm from adoption by visionaries (early adopters) to pragmatists (early majority). Enterprise developers risk becoming the late majority or even worse laggards in the technology adoption lifecycle if executives and architects don't buy into microservices. Executives need to ask the question - Does Microservices solve a business problem and achieve outcomes. ? Microservices will not fix wrong requirements, weak processes, bad habits or automatically bring discipline and maturity to software development. This chapter brings some rigor around the processes and practices for gaming the social aspects of the organization to implement microservices based system on a CI/CD devops substrate.

Microservice Templates

Standardize in the gaps between microservices and be flexible about what is inside. Have a clear custodian model for services. Standardize around practices, tools, ownership and trust.
Standardize on integration and deployment platform. Standardize on logging, monitoring and metrics. Use service templates for standardization building upon spring starter projects for Spring Boot, Spring Cloud, Spring Data, Spring Data Rest, Spring Integration and Spring AMQP. Writing a new microservice should be as simple as cloning a git repository that bakes in these best practice patterns and implementation choices. These starter repositories will serve as the de facto reference architecture for your system. Please take care to get the microservice template repositories correct with the right technology choices.

For a sample microservices service templates see

1. Spring cloud samples customer-stores and processor apps from spring-cloud.
2. Groovy service templates from 4finance boot-microservice

Conway's Law

"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations" [conway's-law]. Communication structure of the organization has a direct bearing on the software structure it produces.

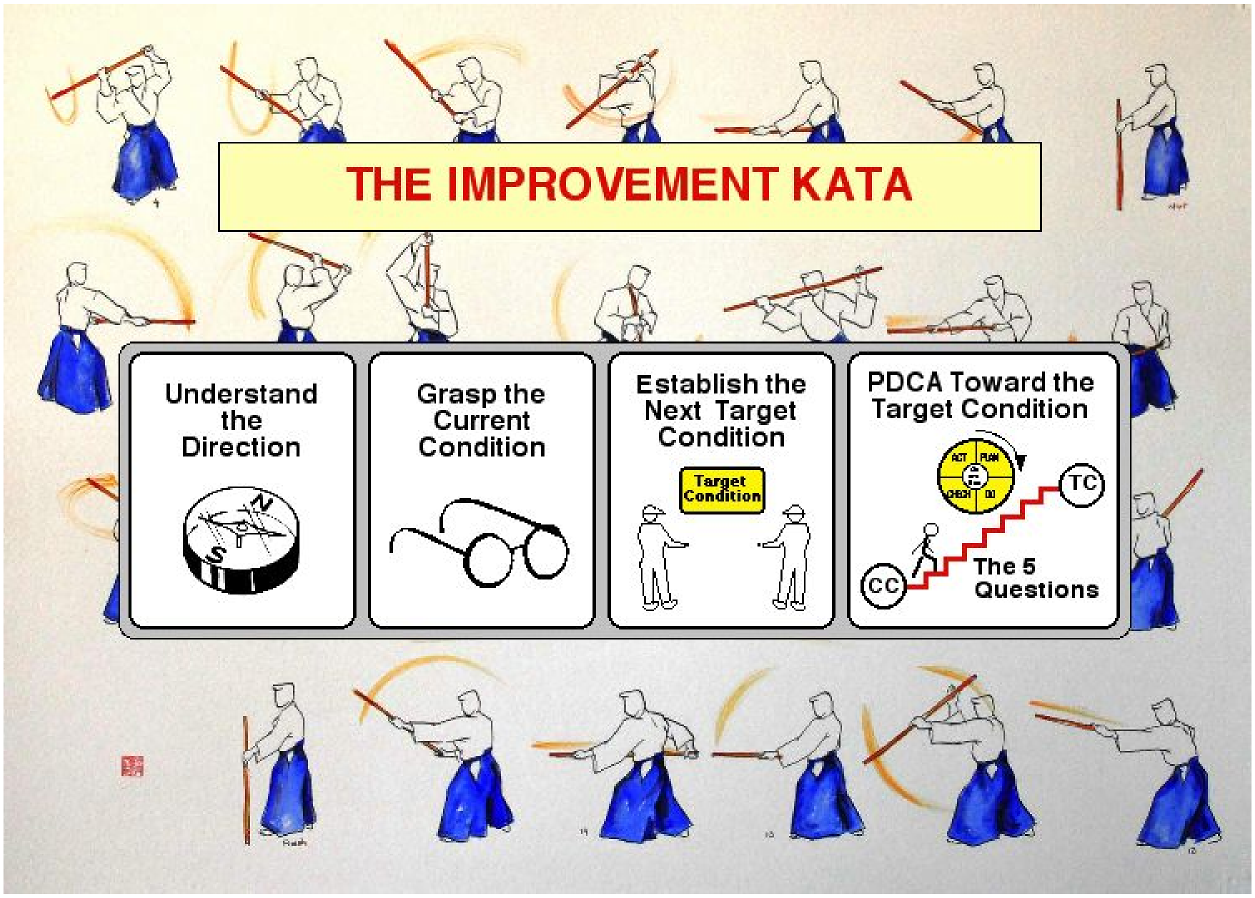
Inverse Conway Manoeuvre

Inverse Conway's law and build small teams that looks like the architecture you want. Strive for a low efferent coupling for your team. Teams with low outbound coupling deliver relatively independently into a common integration pipeline without fearing breaking each others builds. Process should support easy change. Reorganize teams around microservices or group of microservices that need to be built.

Improvement Kata

The Improvement Kata provides a way to align teams and, more generally, organizations by taking goals and breaking them down into small, incremental outcomes (target conditions) that get us closer to our goal. The Improvement Kata is also an effective way to develop the capabilities of people throughout the enterprise so they can self-organize in response to changing conditions. The paradox is that when managers focus on productivity, long-term improvements are rarely made. On the other hand, when managers focus on quality, productivity improves continuously. Good habits, especially when enforced from the top down, can actually help open up new and improved procedures and processes all the way down the line. [power-of-habit]. A kata is a pattern you practice to learn a skill and mindset. Through practice the pattern of a kata becomes second nature - done with little conscious attention - and readily available. Follow the adaptive, iterative approach of the improvement Kata for continous process evolution across all areas of business including software development, organization alignment, financial budgeting etc., Keep in mind that the target conditions are all specific, measurable, achievable, relevant, and time bound. It is up to the people doing the work to run a series of experiments using the Deming [Plan, Do, Check, Act] cycle. The improvement kata is a meta-methodology that teaches teams how to evolve their existing playbook. It is not a playbook; rather, as with the Kanban Method, it teaches teams how to evolve their existing playbook. In the Improvement Kata process and practices you use are expected to evolve over time. This is the essence of agile: teams do not become agile by adopting a methodology. Rather, true agility means that teams are constantly working to evolve their processes to deal with the particular obstacles they are facing at any given time. [lean-enterprise]

[lean-workshop]



It is important to not to get too hung up on a particular methodology such as Kanban, Scrum or XP. No matter which tool or process your team is using, at the end of the day, it has to be about getting the software out the door given the people you have and the resources at your disposal. No methodology is perfect and a good team makes adjustments to suit the reality of their project. Remember Agility is a culture not a process. [agile-manifesto]

Team Structure

We want to move from project based teams to product based teams. The ideal long term model for organizational and technical agility to achieve speed with minimal risk is prescribed below -

To this end establish a Centralized (CORE) team that owns

- Operational Excellence Leadership
  - Tools, Pipeline, Services, Infrastructure
  - Testing Tools
  - Expertise (Best Practices, Questions to ASK)
  - Keeping a topic in focus - availability
  - Monitoring and Alerting services - Dispatch to Dev Team - Dev makes Fallback work
  - Metrics and Trending and Telemetry

Establish individual Business Product teams that own

- Functionality (Obvious)
  - Builds & Deployments
  - Performance (Non-Obvious)
  - Scale
  - Availability (Redundancy, Dependencies, Configurations)
  - Security
  - Privacy
  - Retiring Technical Debt

In Google, teams working on a new product must pass a “production readiness review” before they can send any services live. The product team is then responsible for its service when it initially goes live. After a few months, when the service has stabilized, the product team can ask operations—called Google’s Site Reliability Engineers, or SREs—to take over the day-to-day running of the service, but not before it passes a “handover readiness review” to ensure the system is ready for handover.

Ownership

Establish long lived fluid teams with clear owners for each microservice. For shared cross functional microservices establish an open source git pull request workflow with a custodian akin to the benevolent dictator model. Establish a team culture that helps the the team to do the right thing out of the box. Governance is best done by making it easy to do the right thing. Trust but verify.

Sharing

Avoid duplication of data, sharing data sources, use of the same database by multiple microservices. Be cognizant of sharing common code like technical infrastructure libraries, between microservices as this ends up in “build the world” CI storms if one of the base libraries is changed. For sustained innovation and ability to change *anything* as desired, you must work to reduce the sharing between areas of the system to allow them to move independently of each other. There is a middle ground between isolation and reuse. Be aware of the the choices you make as leaning towards one is leaning away from the other. [sharing-code]

Collaboration

By learning from the distributed, asynchronous collaboration found particularly in open-source communities and largely virtual companies (e.g. GitHub), teams can benefit greatly in terms of the quality of people they can hire as well as their ability to collaborate more effectively.

Balancing Risk With Speed of Delivery

Retire risk by and transition to a microservices based architecture by using Pathfinder projects. This project's primary purpose is to solve a particular pressing problem using the desired technology, iterate on the implementation and best practices and then roll it out gradually to the rest of the enterprise architecture. Microservices scales the development process by incorporating an independent micro-culture in each bounded context.

Infrastructure and platforms can be agile. A waterfall ‘build it and they will come’ approach is often used to build IaaS deployments; try an agile approach for PaaS to ‘try, fail/succeed, iterate, succeed’. Gradually refine your adoption of platform infrastructure features with real information based on experience rather than theory. Taking apps to production can expose unintended side-effects. These are important to know now rather than later.

Taking One App to Production is Valuable then expand horizontally exposes. Exposes you you to process challenges and allows you to gain acceptance for process modification without impacting ongoing development. Allows you to exercise your continuous delivery process and gain experience operationalizing Pivotal CF in a production environment – key insight that can be leveraged in a feedback loop to continuously improve. Iterate the development of processes and procedures guided by the lessons learned from the deployment of the apps in production.

Which Apps To Pick First

Not All apps are great candidates for migration to a PaaS e.g. commercial off the shelf apps, mainframe or large immovable monoliths that depend on scaled-up or stateful scaled out infrastructure. Draw a line through your application portfolio and migrate apps above line; Adjust the line over time. Incrementally modernize, transitioning apps from a diversity of heavy runtime containers to a few lightweight containers adopting basic 12 factor concepts. Abandon container dependent functionality and container-specific tooling.

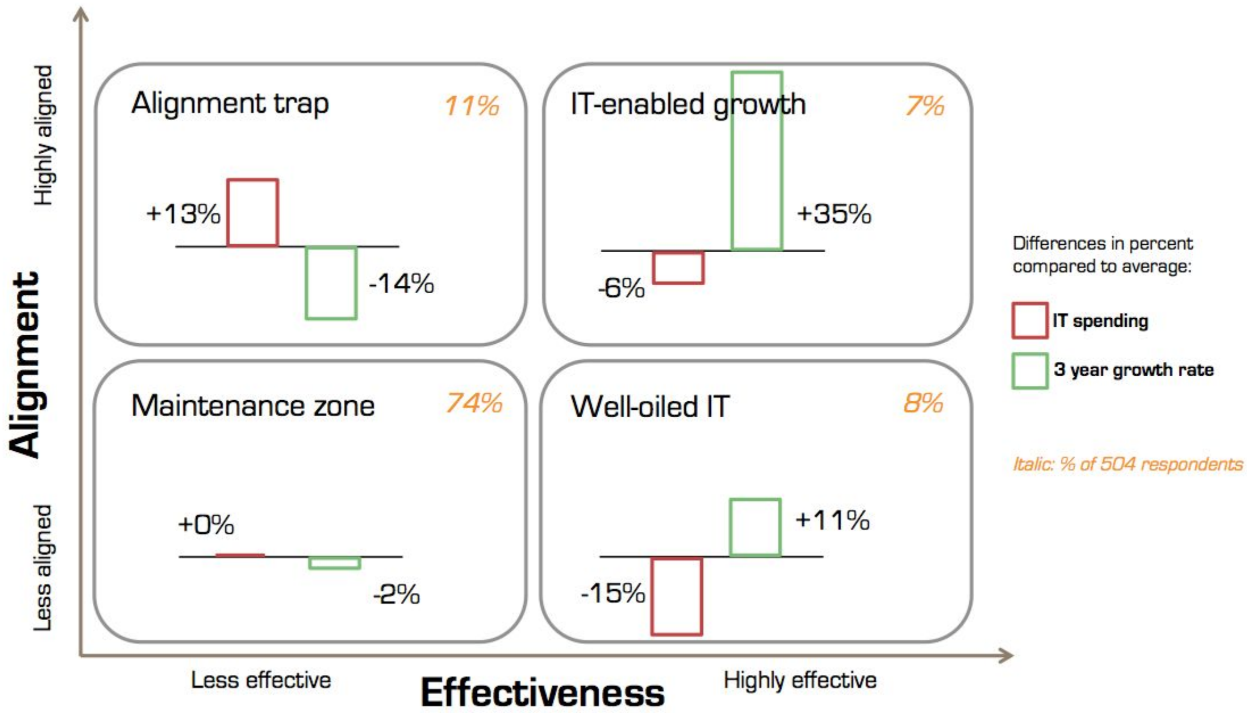
It is possible to make no changes to applications. But it is advisable to make some common changes like - environment specific configuration, injection of external service dependencies instead of up-front configuration, log to stdout/stderr instead of files and converging the app to a smaller number of configurations for app server. language and framework.

Transitioning to a PaaS

Hopefully you are not stuck in a command-and-control type organization. If you do find yourself saddled in an organization that is not receptive to change follow some of the strategies below to get the elephant to dance.

Avoiding the Alignment Trap

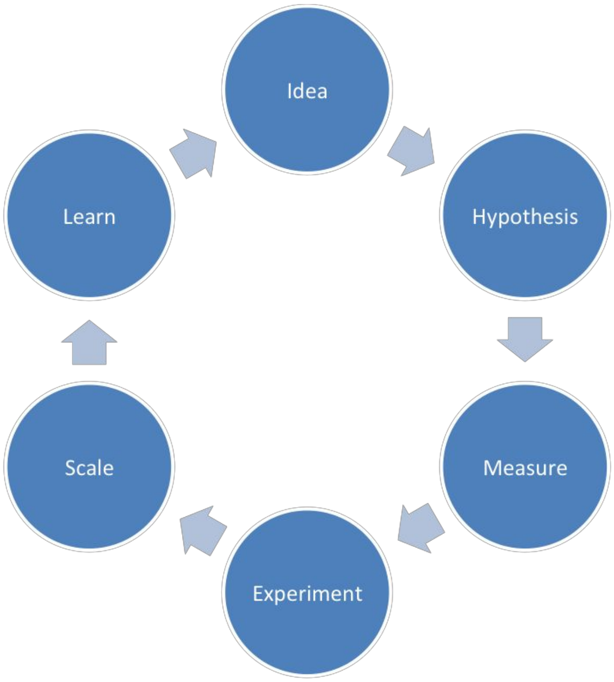
As a techie the Alignment trap data really resonated with me and will appeal to IT exec's who need to be convinced of the need to change. [alignment-trap]. The data reveals that companies whose IT capabilities were poor achieve worse results when they pursue alignment with business priorities before execution, even when they put significant additional investment into aligned work. In contrast, companies whose engineering teams do a good job of delivering their work on schedule and simplifying their systems achieve better business results with much lower cost bases, even if their IT investments aren't aligned with business priorities. In order to affect IT enabled growth and reduce spending it is critical for IT align with business priorities. This is the same thinking behind the Lean movement. The aim of lean thinking is to create a lean enterprise, one that sustains growth by aligning customer satisfaction with employee satisfaction, and offers innovative products or services profitably whilst minimizing unnecessary over-costs to customers, suppliers and the environment. [wikipedia-lean-thinking]



High Functioning Teams

Empower everyone in the organization to innovate and explore rapidly changing landscape. The higher order needs on Maslow's hierarchy will only be met when individuals are given freedom, responsibility and autonomy to control their own destiny. Provide insight and understanding to enable sound decisions . Follow Lean product development methods to execute the innovation loop (see below). Leverage PaaS as a platform for innovation. Innovate and lean process applies to all aspects of the enterprise including software, systems, products, user experience, logistics and business models.

Create highly aligned loosely coupled structured innovation teams to execute ideas and create value. Minimal cross functbetween teams except for alignment. Ground breaking ideas don't look so in the first incarnation. Use learning via analytics, A/B Testing and market data to test assumptions and hypothesis and pick winning ideas.



Affect Organizational Change

John Shook, Toyota City’s first US employee, reflected on how cultural change was achieved at the NUMMI plant where extraordinary levels of quality, productivity and reduced costs were achieved despite rehiring employees that were considered the worst workforce in the automobile industry in the United States - [mit-sloan-review]

What my NUMMI experience taught me that was so powerful was that the way to change culture is not to first change how people think, but instead to start by changing how people behave—what they do. Those of us trying to change our organizations’ culture need to define the things we want to do, the ways we want to behave and want each other to behave, to provide training and then to do what is necessary to reinforce those behaviors. The culture will change as a result…What changed the culture at NUMMI wasn’t an abstract notion of “employee involvement” or “a learning organization” or even “culture” at all. What changed the culture was giving employees the means by which they could successfully do their jobs. It was communicating clearly to employees what their jobs were and providing the training and tools to enable them to perform those jobs successfully.

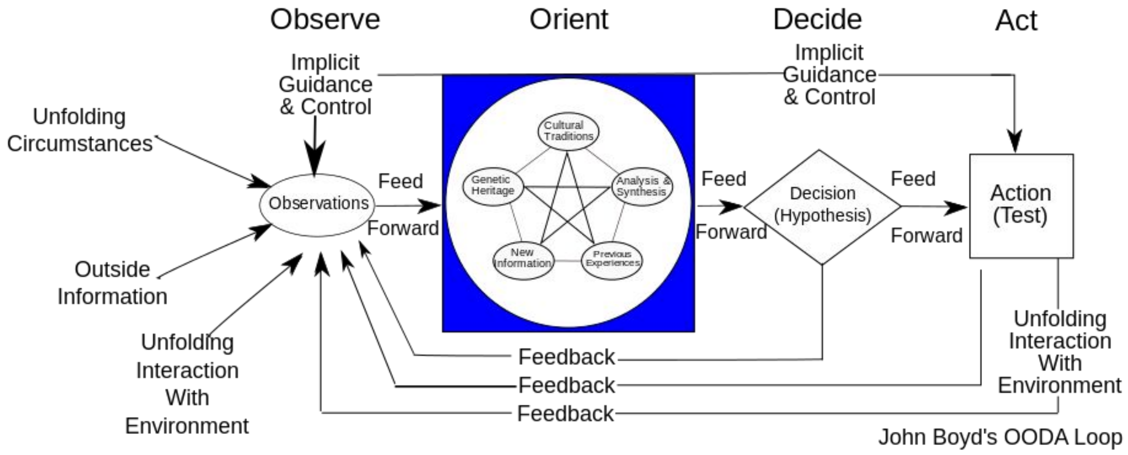
Another way to achieve sustained systemic change is to wait for a crisis. You never let a serious crisis go to waste. And what I mean by that it's an opportunity to do things you think you could not do before. [Rahm-Emanuel]. Crisis is a necessary condition of successful transformations; Because humans avoid unpredictability and uncertainty. For change to succeed, survival anxiety must be greater than learning anxiety, and to achieve this, “learning anxiety must be reduced rather than increasing survival anxiety. [Schein-Corporate-Culture Survival Guide]

All of the top executives, need to believe that considerable change is absolutely essential. The essence of a lean mindset is understanding that this should be the case not just in a crisis but all the time. Change, improvement, and development are habitual in a truly lean organization. Changing culture is achieved by the deliberate, repeated, mindful practice of everyone in the organization. Leaders and managers must facilitate this by investing in employees' development and creating conditions to support people working together to continuously improve processes, knowledge, and the value delivered to customers. Finally, it's essential that leaders model the behaviors they expect the rest of the organization to adopt. [lean-enterprise]

Treat a microservices based product on the Cloud Foundry PaaS as an opportunity to modernize and change 'the way things are done'. ' A New Way' of getting apps to production. The corollary is to not re-implement old processes with new tools. You will fail to realize the benefits of the new way if you do.

Begin your journey with the following principles - 1. Define and limit your initial scope 2. Pursue a high-performance culture of continuous improvement 3. Start with the right people 4. Find a way to deliver valuable, measurable results from early on. Finally as you experiment and learn, share what works and what doesn't. Hold retrospectives to reflect on what you have achieved and use them to update and refine your vision. Always, keep moving forward. Fear, uncertainty, and discomfort are your compasses toward growth. Establish a center of excellence whose job is to pioneer and implement forward looking lean techniques for CI/CD and full life-cycle devops. Implement process improvement through experimentation and incubation. Follow the OODA loop and keep changes small. Idea -> hypothesis -> experiment -> measure -> scale -> learn.

"OODA.Boyd" by Patrick Edwin Moran - Own work. Licensed under CC BY 3.0 via Wikimedia Commons



Our end goal is to create a foster a culture of creativity and self-discipline. One where engineers have freedom and take responsibility for their actions.

Testing

Creation of clear, well defined service boundaries is key to successfully testing a distributed system modeled as cooperating microservices. Clear boundaries makes it easier to see the components of the system and treat them in isolation. Test code in both pre and post production environments. Rely on a combination of test strategies to get high coverage.

Organization

There should not be a separate QA team responsible for testing. At Google, Test exists within a Focus Area called Engineering Productivity. Eng Prod is made of:

1. A **product team** that produces internal and open source productivity tools that are consumed by all walks of engineers across the company.
2. A **services team** that provides expertise to product teams on a wide array of topics including tools, documentation, testing, release management, training and so forth.
3. **Embedded Test engineers** that are effectively loaned out to product teams on an as-needed basis.testers report to Eng Prod managers but identify themselves with a product team.

Every developer is expected to do their own testing. The job of the test embedded engineer is to make sure they have the automation infrastructure and enabling processes that support this self reliance. Testers enable developers to test. This strateguy puts developers and testers on equal footing with i.e. true partners in quality. [how-google-tests-software]

Test Strategies for Microservice Architecture

A substantial portion of the section below has been taken and inspired from Martin Fowler and Toby Clemson's Infodeck on Testing Stratgies in a Microservice Architecture see http://martinfowler.com/articles/microservice-testing/

Unit Testing

Can be categorized as sociable or solitary. Social testing tests behavior of modules by observing changes in state. Solitary testing looks at interactions and collaborations between an object and its dependencies. Unit testing should not constrain the implementation of module under question. The danger in microservices based testing is an over reliance on mock based approaches.

Component Testing

In process component tests allow components allow comprehensive testing while minimizing moving parts. The component under test has to start under a `test` profile. Spring's support for profiles helps an application start with a different configuration based on the environment. Tests leverage embedded in-memory implementations of distributed datastores and databases like H2. Tests expose internal resources for state manipulation and failure injection.

Exposing application internal resources is useful in a number of use cases besides testing such monitoring, maintenance and debugging. These resources expose features such as logs, feature-falges, DB commands, system metrics, health check, timings of key transactions and details of config parameters, ping end points for LB. Spring Boot provides production features listed above with the actuator module. It is critical to lock down internal non-public resource endpoints with techniques like authenticating and hiding by namespacing using different URL conventions or exposing on a different network port.

Out of process component tests exercise the fully deployed microservice thereby pushing complexity into the test harness. External stubs can be leveraged as placeholders for external services. External stubs come in three flavors 1. Dynamically programmed via API, 2.Hand crafted fixture data & 3. Record play mechanism of capturing request/response behavior to the real external service.

Integration Testing

In microservice architectures integration testing is typically used to verify interactions between layers of integration code and external components to which they are integrating such as other microservices, data-stores, caches. Plan and build your integration test framework early. Integration Put in place integration testing for all service API interactions to enforce and codify implicit contracts between services. Integration test automation is a MUST.

Integration testing is hard. Integration tests do NOT scale in a system with hundreds of microservices. Use Consumer Driven Contract testing as part of the build to identify breaking changes in a service. CDC tests run together with the provider tests can be an effective replacement for integration tests and lead to decentralized deployment pipeline where the integration test sink for all services is replaced with a consumer test in each service build and deploy pipeline. These are tests contributed to and created with the consumer of the service. Any downstream dependencies or services should be stubbed out. A couple of JVM based test frameworks that provide explicit support for testing CDC's are pact-jvm and groovy based 4finance-stub-runner. In addition to testing these frameworks also provide the ability mock the services under test.

Contract Testing

Test at the boundary of an external service verifying it meets the contract of the consuming service. Maintainers of each consuming service write an independent test suite that verify 1. Input and ouput of the service calls contain required attributes 2. Response latency and throughput are under acceptable limits. Contract test suites are then packaged and run as part of the CI pipeline of the producing service ensuring that the maintainers of the service know the impact of their changes on consumers. Its prudent to remember Postel's law when designing service API contracts and implmentations - "Be conservative in what you send, be liberal in what you accept". [robustness-principle].

Evolve rest APIs following the principle of "Do no harm first" i.e. First expand the API maintaining multiple versions of the API allowing plenty of time for consumers to get familiar with the newer version and migrate. Thereafter after all consumers have migrated over to the new API, contract the API and remove the older version. Remember to follow semver guidelines when versioning the API.

End-2-End Testing

The purpose of end-2-end testing to confirm that the system as a whole meets business goals. They provide confidence that business will remain intact during large scale refactoring. If possible run test cases in language of domain in this phase using business readable DSLs. Apply a time budget when executing end-2-end tests. Some of the lessons learnt during end to end testing are - Focus in customer and user personas. Embrace the IaaS to provision an environment from scratch for every system test. Make tests independent of data. Do not rely on pre populated data. Support construction of entities via public/internal APIs. The tests should define their own world before execution. Some teams forego end-2-end testing in favor of production monitoring and testing against production.

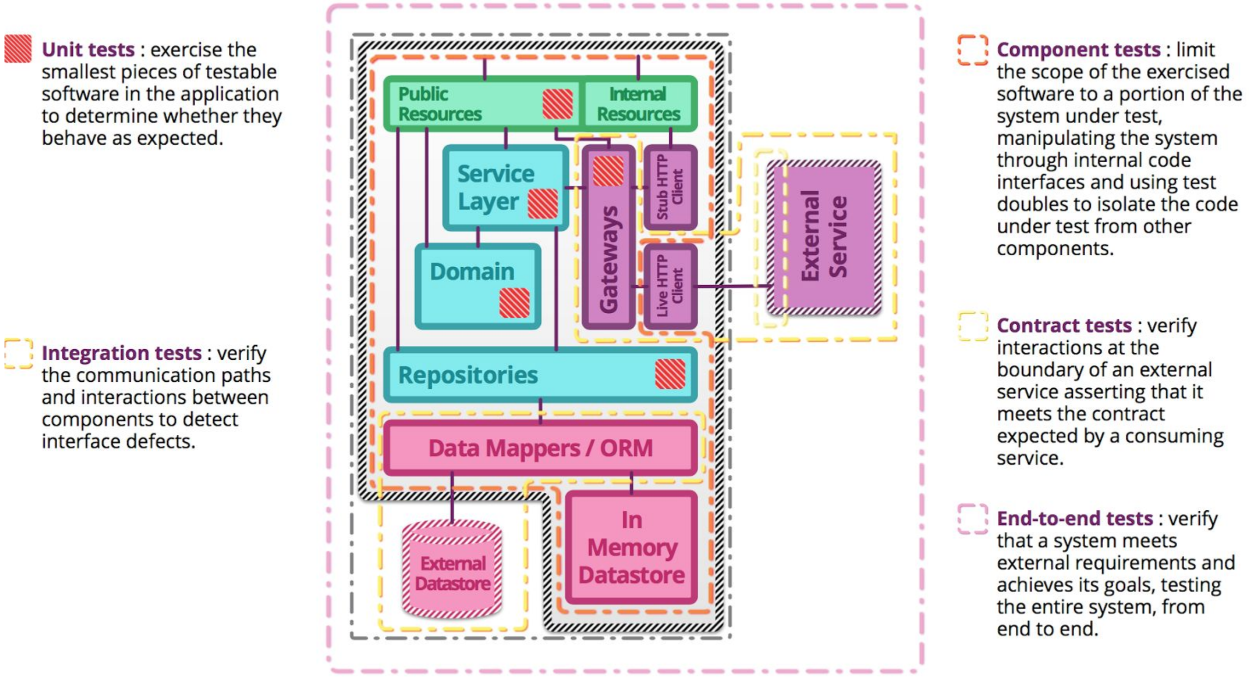
Synthetic Monitoring

Post-production testing entails creating synthetic transactions and fake requests to exercise staging and production systems. Application code needs to have the ability to differentiate a synthetic conversation/request/transaction from a real one to avoid unintended consequences. Use end-to-end journey tests for synthetic and semantic monitoring of the production system. In the same vein, employ the Netflix simian army to stress your production system for high availability, audit, security and disaster recovery scenarios. Focus on a small number of core journeys to test for the whole system instead of adding a new end-to-end test for every function. Optimize for fast feedback and separate tests accordingly. Testing on real users at scale is possible, and provides the most precise and immediate feedback.

Production Testing

Expose, test and validate certain percentage (2%) of production users to the new system while routing the majority to the older system. Alerting is put in place when key business metrics fall outside of acceptable boundaries. Once results have been validated dial up the traffic on the new system.

Testing Strategies for Microservice Architectures
http://martinfowler.com/articles/microservice-testing/#conclusion-summary



▨ **Unit tests** : exercise the smallest pieces of testable software in the application to determine whether they behave as expected.

Integration tests : verify the communication paths and interactions between components to detect interface defects.

Component tests : limit the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces and using test doubles to isolate the code under test from other components.

Contract tests : verify interactions at the boundary of an external service asserting that it meets the contract expected by a consuming service.

End-to-end tests : verify that a system meets external requirements and achieves its goals, testing the entire system, from end to end.

Testing Frameworks

We **recommend** using Spring Boot's @IntegrationTest and Spring's `TestRestTemplate` to create functional tests. TestRestTemplate conveniently subclasses RestTemplate in way that is suitable for integration tests. RestTemplate is a powerful means of functionally interacting with your RESTful services, regardless of your testing framework. @IntegrationTest is used to instruct Spring Boot that the embedded web server should be started which reduces test turnaround time. The very same container is used for developing, testing and production. [Tutorial-RestTemplate]

Dropwizard and Ratpack also have excellent support for testing via the dropwizard-testing module and the ratpack-test libraries respectively. These extensible micro-containers have been built with testing as a first class concern for REST resources. This is reflected in the quality of helper classes and ease and speed with which tests can be written not only for REST APIs but deeper aspects of the container integration. For those coming to Java from the Ruby behavior driven testing world, Cucumber-jvm provides a pure Java implementation of Cucumber and the best tradeoffs from a bevy of BDD Java frameworks. Front end testing is typically accomplished with libraries like Selenium, Watir or Geb.

Mocking

For every dependency create a test double. Never mock internals. Mock externals instead.
Use wiremock for stubbing Services. It has direct support for JUnit using @Rule feature. Unit tests can be written to stub out different http requests and verify those requests and responses easily. Betamax inspired by the VCR record-replay style stubbing library for Ruby is another tool for mocking HTTP REST API calls. Betamax integrated with interaction test framework Spock provides a supercharged way to do HTTP based microservice testing. A honorable mention in

this category goes to the Mountebank. open source tool that provides dynamic fixture based stubs. Moco and Stubby4J are also worthy of consideration.

Use libraries like REST-assured for testing and validating REST APIs. REST Assured is a Java Groovy DSL for simplifying testing of REST based services built on top of HTTP Builder. It supports POST, GET, PUT, DELETE, OPTIONS, PATCH and HEAD requests and can be used to validate and verify the response of these requests. Unlike VCR, it does not stub out the underlying HTTP client library, but rather is an actual service that is configurable to respond in arbitrary ways. As an immediate result it is language agnostic, supports spying, XPath-Validation, Specification Reuse, easy file uploads, cookie handling and many other goodies. [testing-restful-webservices]

When a microservice has a non-trivial number of consumers, it pays to build and distributed a custom test stub specific to the service so that consumers don't end up writing 10 stubs for the same service.

Testing on Cloud Foundry

Junit or HTTPUnit tests targeting Cloud Foundry should use the cf-java-client library to communicate with the CF deployment. When writing & validating test results use the performant loggregator based streaming APIs like `CloudFoundryClient.getLogs` & `CloudFoundryClient.getRecentLogs`. Please note during testing, it is extremely important to do error checking when application staging fails in CF to avoid false negatives. Tests need to be failed immediately with a proper output of response and error codes returned by CF through the client. If the test fails, sometimes a retry of the same test works, therefore implement a retry mechanism with a random jitter for failing tests.

13 Continuous Integration and Continuous Delivery

It is fitting to begin this chapter with some definitions coined by Jez Humble & other stalwarts -

Continuous integration is the practice of working in small batches and using automated tests to detect and reject changes that introduce a regression. It is the most important technical practice in the agile canon, and forms the foundation of continuous delivery, for which we require in addition that each change keeps the code on trunk releasable. [lean-enterprise].

Continuous delivery is the ability to get changes experiments, features, configuration changes, bug fixes—into production or into the hands of users safely and quickly in a sustainable way. [lean-enterprise]

DevOps practices are the ideal intersection of people, processes, and tools across the application lifecycle, facilitating the seamless delivery of software to the business. [ema]

CI and CD cannot be realized without the tooling for build, test and deploy of code from source control. Tens and hundreds of microservices created by diverse product teams cannot march forward without automation paving the road ahead. On a large, distributed team Continuous Integration is the only process that is known to scale effectively without the painful and unpredictable integration, stabilization, or "hardening" phases associated with other approaches, such as release trains or feature branches. Continuous delivery is designed to eliminate these activities. [lean-enterprise]

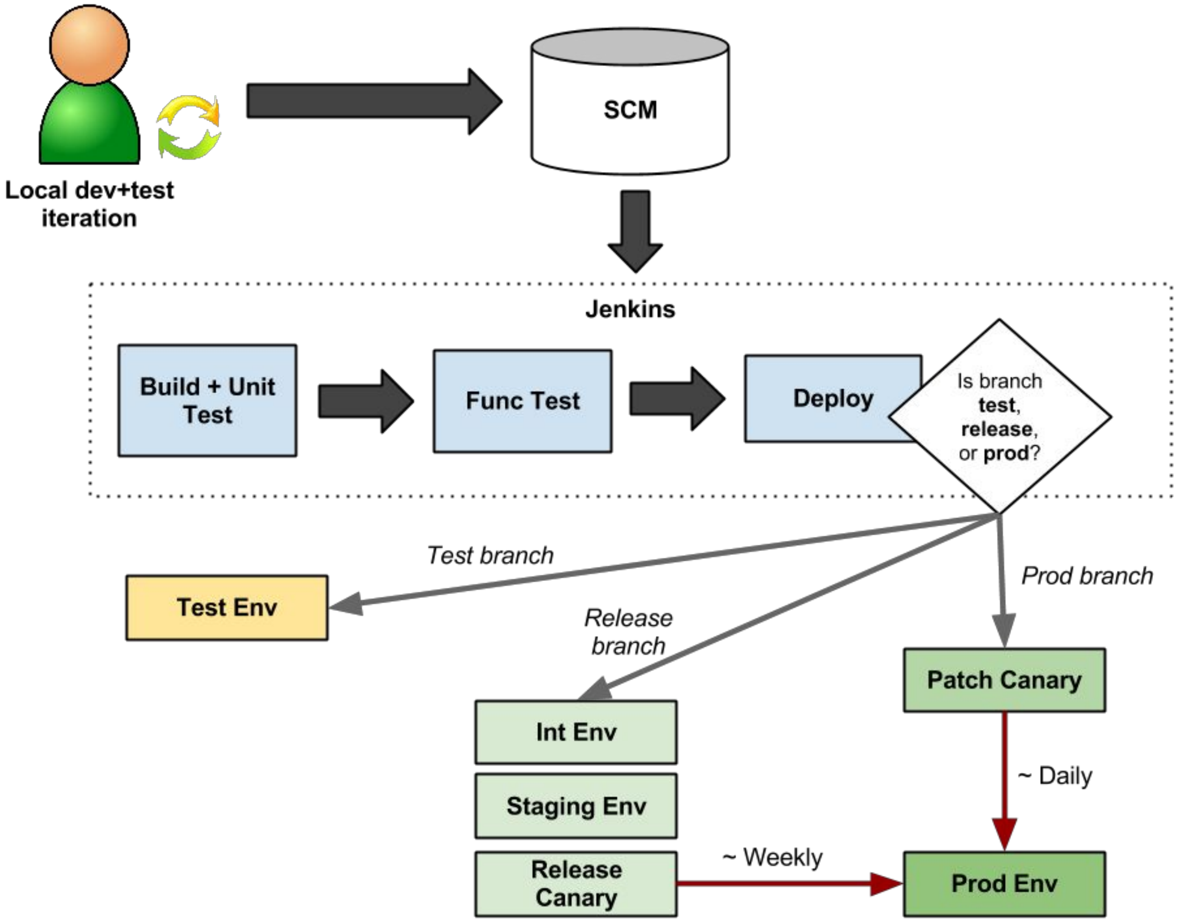
Each phase in the CI flow provides feedback about the "goodness" of the code, informs successive step providing more insight into and confidence about feature correctness and system stability. The pipeline is automated, and tooling gives insight into code as it moves from one state to another.

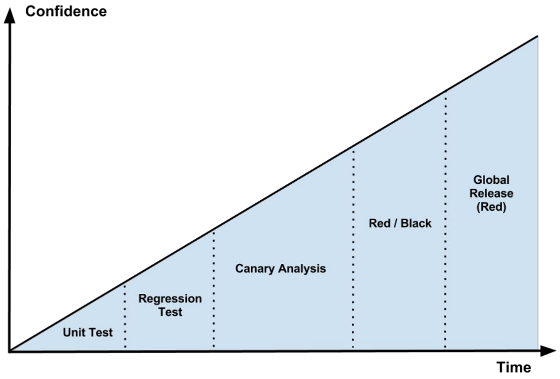
Lessons Learnt From Netflix

Aim to automate the delivery pipeline and provide insight into both the tools used to develop and deploy the service as well as the monitoring systems used to track the health of running applications. The ability to trace code as it flows from SCM systems through various environments and quality gates on its way to production, helps distribute deployment and ops responsibilities across the team in a scalable way. Tools that surface feedback about the state of the CI pipeline and running apps give confidence to move fast and help us quickly identify and fix issues when things break. Establish automated canary analysis and generate a source diff report, cross-linked with Jira/GH, of code changes in the application binary, and a report showing library and config changes between the baseline and canary. These artifacts increase visibility into what's changing between deployments. Create automation for deployment tooling with provision for auto rollback. [netflix-api-deploy]

CI Pipeline and Deploy Flow

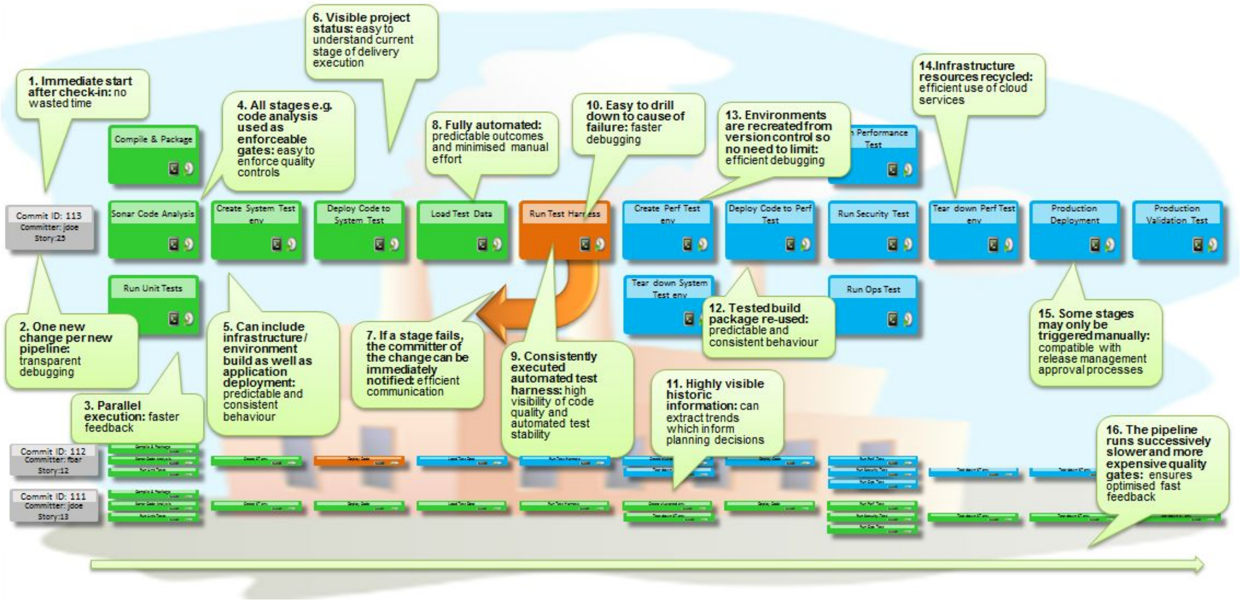
[deploying-netflix-api]





Build for continuous delivery from day 0. Your first task in a product should be to push a HelloWorld application to production, run contract tests and then write services. Every new hire should have the ability to push a delta change to production on their first day. Don't let changes build up - release as soon as you can and preferably one at a time. Each Microservice should have its own code repository and all the services are integrated via continuous integration.

Practical Benefits to Continous Delivery [markosrendell]


6. Visible project status: easy to understand current stage of delivery execution
1. Immediate start after check-in: no wasted time
4. All stages e.g. code analysis used as enforceable gates: easy to enforce quality controls
8. Fully automated: predictable outcomes and minimised manual effort
10. Easy to drill down to cause of failure: faster debugging
13. Environments are recreated from version control so no need to limit: efficient debugging
14.Infrastructure resources recycled: efficient use of cloud services
Compile & Package
Performance Test
Commit ID: 113 Committer: jdoe Story:25
Sonar Code Analysis
Create System Test env
Deploy Code to System Test
Load Test Data
Run Test Harness
Create Perf Test env
Deploy Code to Perf Test
Run Security Test
Tear down Perf Test env
Production Deployment
Production Validation Test
Run Unit Tests
Tear down System Test env
Run Ops Test
2. One new change per new pipeline: transparent debugging
3. Parallel execution: faster feedback
5. Can include infrastructure / environment build as well as application deployment: predictable and consistent behaviour
7. If a stage fails, the committer of the change can be immediately notified: efficient communication
9. Consistently executed automated test harness: high visibility of code quality and automated test stability
12. Tested build package re-used: predictable and consistent behaviour
11. Highly visible historic information: can extract trends which inform planning decisions
15. Some stages may only be triggered manually: compatible with release management approval processes
16. The pipeline runs successively slower and more expensive quality gates: ensures optimised fast feedback
Commit ID: 112 Committer: fbar Story:12
Commit ID: 111 Committer: jdoe Story:13

Infrastructure As Code

Just like application source infrastructure is also code and should be maintained as such in its own repository. Infrastructure repository should be linked with the application repository for every production release of the microservice system. Excellent configuration management underpins continuous delivery. All provisioning of the underlying IaaS (containers or VMs) that host the service should be automated. All changes to any system or the environments it runs in should be made through version control and then promoted via the deployment pipeline. This includes not just source and test code but also database schemas, documentation, deployment and provisioning scripts, as well as changes to server, networking, and infrastructure configurations.

Service Versioning

Each service (typically REST over JSON/XML) should have its own lifecycle and has publicly versioned contract. Deliver small changes to a service very frequently. Smaller changesets are easier to reason about. Services should be semantically versioned based on the semver specification. The key tenets of semver are repeated below - Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Each service should be independently deployable as an unit otherwise you have created a distributed monolith NOT a microservice architecture. Service function needs to be validated at the end of the CI pipeline on a canary server that is healthy. The basic idea of a canary is that you run new code on a small subset of your production infrastructure, for example, 1% of prod traffic, and you see how the new code (the canary) compares to the old code (the baseline).

Dark Launching Features

Reduce risk of deployment in production by dark launching features i.e deploy with the feature turned off or employing feature flags. Components are deployed and features are released. Use feature flags to turn on/off features. Applications consist of routing. Deploy the component, turn feature on, start routing and check to see if service is working as expected. If the service falls on its face, route around the service. Use monitoring to do integration and disintegration. Practise the 1. Blue/Green 2. Deploy 3. Release 4. Route 5. ReRoute cycle upon failure in production.

Enterprises that do multiple deploys to production every day have a bulletproof failsafe in terms of process. To achieve this level of productivity the following pieces have to be in place

- The ability to dynamically configure the fraction of the total traffic received by the application. Dial it up or down depending on the feature correctness, capacity and performance.
- Selective control over the number of users internal or external that can see the feature/service.
- Start incrementally, verify, tweak, course correct and eventually transition all traffic to the new feature/service.

Developers protect new features with “feature flags” so that administrators can dynamically grant access to particular sets of users on a per-feature basis. In this way, features can be made available first to internal users and, then to a small set of users as part of an A/B test. Validated features can then be slowly ramped up to 100% of the user base—and switched off under high load or if a defect is found. Feature toggles can also be used to make different feature sets available to different groups of users from a single platform.

Spring Framework support for profiles are the natural way to implement feature flags in a spring application. There is no direct support for canary deployments in Cloud Foundry. Canaries can be implemented via a dynamic routing filter in the service API gateway tier or the software edge JVM router like Netflix Zuul.

Blue/Green or Red/Black Deployment

Deploy newer versions of the app without any downtime utilizing techniques like Blue/Green or Red/Black deployment. Blue-green deployment is a release technique that reduces downtime and risk by running two identical production environments called Blue and Green. At any time, only one of the environments is live, with the live environment serving all production traffic. For this example, let’s say Blue is currently live and Green is idle.

As you prepare a new release of your software, deployment and the final stage of testing takes place in the environment that is not live: in this example, Green. Once you have deployed and fully tested the software in Green, you switch the router so all incoming requests now go to Green instead of Blue. Green is now live, and Blue is idle. This technique can eliminate downtime due to application deployment. In addition, blue-green deployment reduces risk: if something unexpected happens with your new release on Green, you can immediately roll back to the last version by switching back to Blue.

Use Scripts [pivotal-zero-downtime or bluemix-zero-downtime] for Blue Green deployment of a service in production. Another option is to use the Zero-downtime deployment task provided by the CF gradle plugin. Running two concurrent versions of the same service long term is not recommended due to operational, maintenance and consistency issues.

Continuous Integration with Jenkins

Setup a Jenkins server for CI either on 1. CF or 2. on a VM/bare metal or 3. a docker instance. Jenkins can be pushed as an application to Cloud Foundry to avail of the HA capabilities. If you chose 1 or 2 please see instructions here on how to install and configure jenkins on tomcat. The **recommended** best practice is to leverage the the Jenkins Enterprise by CloudBees solution is also available as an add-on service aka a tile in Pivotal CF.

Another popular CI system is Travis. If using travis, you have the ability to deploy directly to CloudFoundry after a successful build on Travis CI. Caching functionality in Travis is in beta and only available to private repositories or by request. By default Travis will download all the project's dependencies from scratch for each build.

Continuous Integration with Wercker

We also like Wercker as a continuous delivery service. Wercker is like Travis with community features or github of delivery tooling. When deploying with wercker use the the following wercker step to push to cloud foundry. Use the step-wercker-flowy-deploy step to do automatic deploys inspired by gitflow. Please note that Wercker is currently in Beta with free registration.

Cloud Foundry Application Lifecycle Management

CF provides APIs to set environment-specific configuration for an app or groups of apps in a space. Your CI process should be aware of the env specific configuration and be able to effect change in it when required.

For basics on orgs, spaces and roles see. For detailed instructions on how to create a build job in Jenkins, create a deployment job for the development environment, and promotion of an app across spaces dev → staging and staging → production please use the Cloud Foundry maven and gradle plugins. If you do use one of these plugins, pay special attention to not re-run the build to generate a new artifact. We want to build one single deployable artifact that can be promoted through all environments and ultimately to production. If we were to re-build the artifact prior to deploying it to each environment, we are essentially deploying an un-tested artifact each time, which is not desirable.

Rollback

There needs to be a feedback loop put in place between monitoring and rollback. To put this into practice several things have to come together. 1. Always remember the last good configuration on a central configuration server. Rollback of configuration variables matters as much as code. 2. Code needs to have the ability to dynamically read and affect the changed configuration 3. Monitoring metrics for latency and system health need to be less than the human attention span. Unforeseen effects can then be caught and remediated before anyone notices. Use the hystrix turbine dashboard and/or other commercial tools that provide sub-second latency monitoring like vividcortex and boundary to determine if a service needs to be rolled back.

Who Stole My Process ? ITIL/Change Management Anyone ? We need to put some reins on this Horse

Fear and uncertainty are natural reactions when you try something new. The lack of detailed change processes and other traditional gates to production code deployment automatically raise questions around what sort of control is being imposed on production deploys. The keys to the golden kingdom will not be relinquished readily. The only way to fight this FUD is through data. The 2014 State of Devops Report survey compared external change approval processes with peer-review mechanisms such as pair programming or the use of pull requests. Statistical analysis revealed that engineering teams that held themselves accountable for the quality of their code through peer review, lead times and release frequency improved considerably with negligible impact on system stability. The data suggests that it is time to reconsider the value provided by heavyweight change control processes. Peer review of code changes combined with a deployment pipeline provide a powerful, safe, auditable, and high-performance replacement for external approval of changes. [lean-enterprise]

Remove Impedance to Continuous Delivery

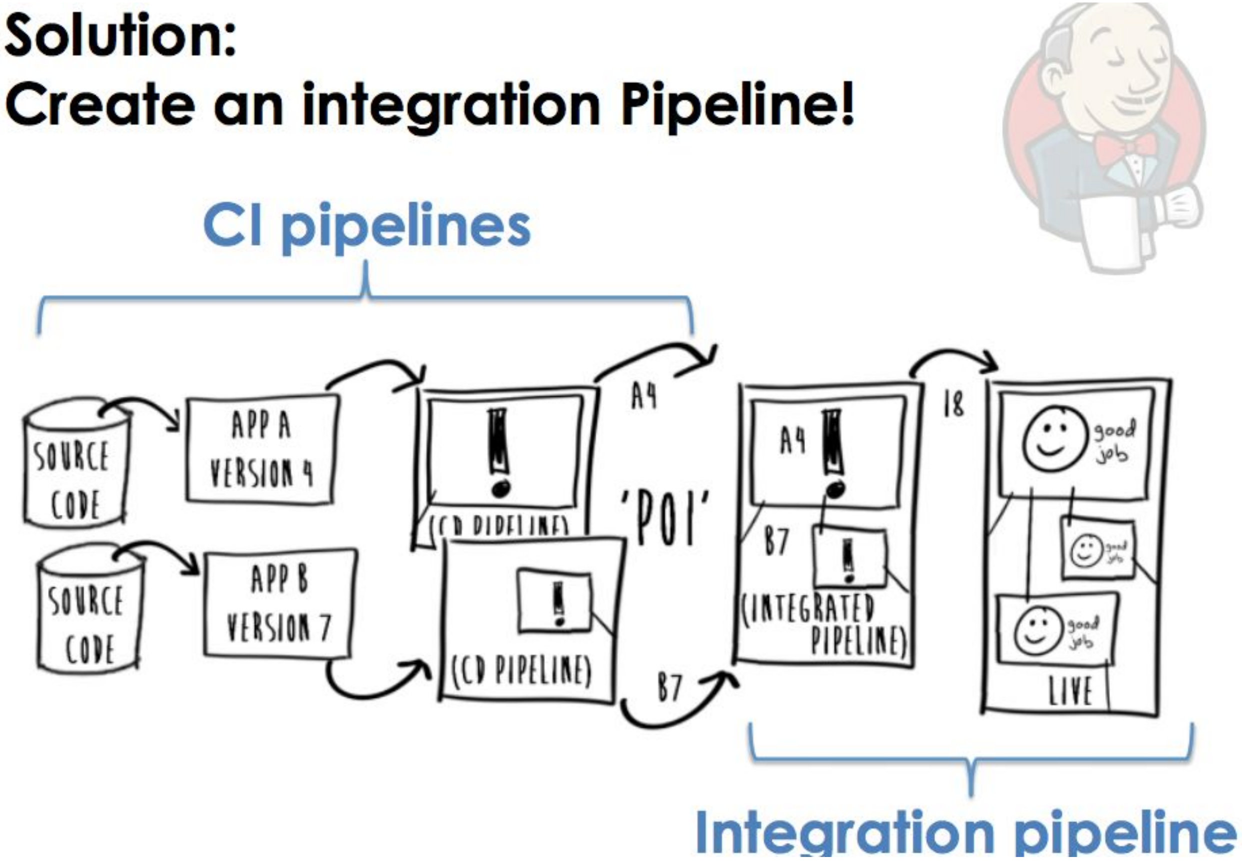
Continuous delivery can be exponentially harder to do, the less your infrastructure solution resembles public cloud. To reduce the impedance from infrastructure to CD, starting small by enabling shadow IT, allowing development and test to use public cloud and understand first-hand how powerful it can be. Demonstrate the benefits achieved there as early as possible and start the cultural acceptance of cloud. Tackle solution complexity by advocating for a PaaS and instituting integrated pipelines for deploying multiple applications under current development that need to work and tested in an integrated manner before release to production. Commercial-Off-The-Shelf Software solutions can be slow or difficult to build and not allow good SCM practices like checking in all the configurable items into flat files. Deployments can also be slow, complex and unpredictable. COTs can be tough to setup and install. Quarantine the COTS package and replace with suitable open source alternatives at the earliest. The last and toughest problem is people objecting to CD either due to ignorance, ambivalence, obedience or disobedience. Some of these can be addressed by simply creating awareness and organizing demos of any aspects of CD already adopted and already adding value. Management has to publicly endorse Continuous Delivery and cite it as the core methodology for ongoing delivery. Adoption can be increased by providing training, tutorials and latitude to explore the various open source tools. Assigning clear ownership and governance over pipeline orchestration, configuration management, automated deployments etc is a very important step as well. Finally a culture shift has to occur wherein management moves away from a project based teams to cross-functional product teams. Engineers traditionally responsible for deployment and maintenance of the service are now embedded with software developers to break down traditional functional silos.

CI Integrated Pipelines for Microservices

When someone checks in code we kick off an orchestration of different process that test the quality of code and perhaps even deploy it. This CI pipeline works well for a single component triggered from one source code repository. This gets complicated quickly when components in multiple software repositories with different quality gates want to be tested together. We dont want to integrate components straightway because components may become shy and not checkin code into trunk/master. One solution to this is using the fat CI-CD pipeline. Every software repository triggers one common fat pipeline for changes to any software component. This is blocking and slow. The CD pipeline needs to provide a path to production that involves carefully integrating lots of components along the way.

Markos Rendell has put together the Integrated Pipelines [pattern] for integrating multiple pipelines to generate a stable tested composite. The solution defines two types of pipeline in Jenkins. One CI pipeline for each microservice/component and an integrated pipeline. There can be multiple of each type. For deploying the right version of the application, several approaches could be used: Maven, Nexus or even a simple plain text file. Whilst components can be dealt with independently, they continue through individual pipelines and at the point they are ready for integration groups of components are deployed to a common environment and once tests pass in that environment then they are put into the integrated pipeline and we move down through the process into successively fatter pipelines.

You might use similar jobs configurations along your different pipelines. The CloudBees templates plugin templatizes your different jobs, allowing you to save time and making the process more reliable. It also allows you to do a one time modification in the template which will automatically be pushed to all the jobs without going individually from one job to another.

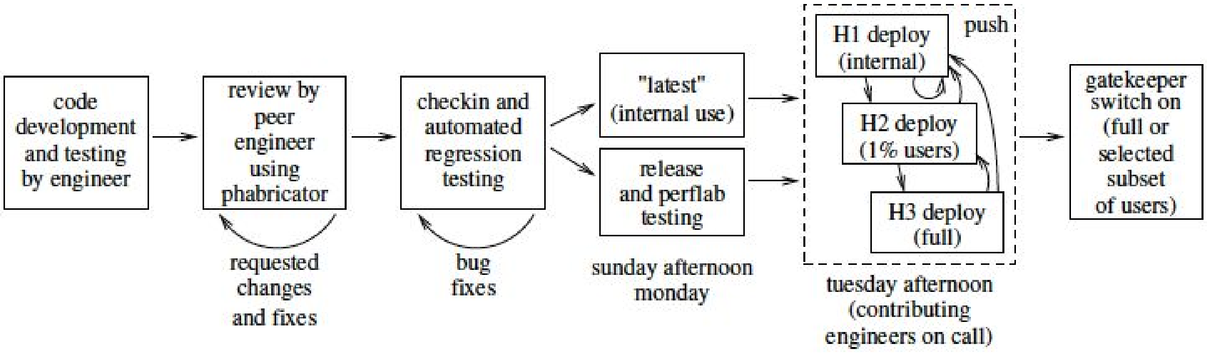


Solution:
Create an integration Pipeline!

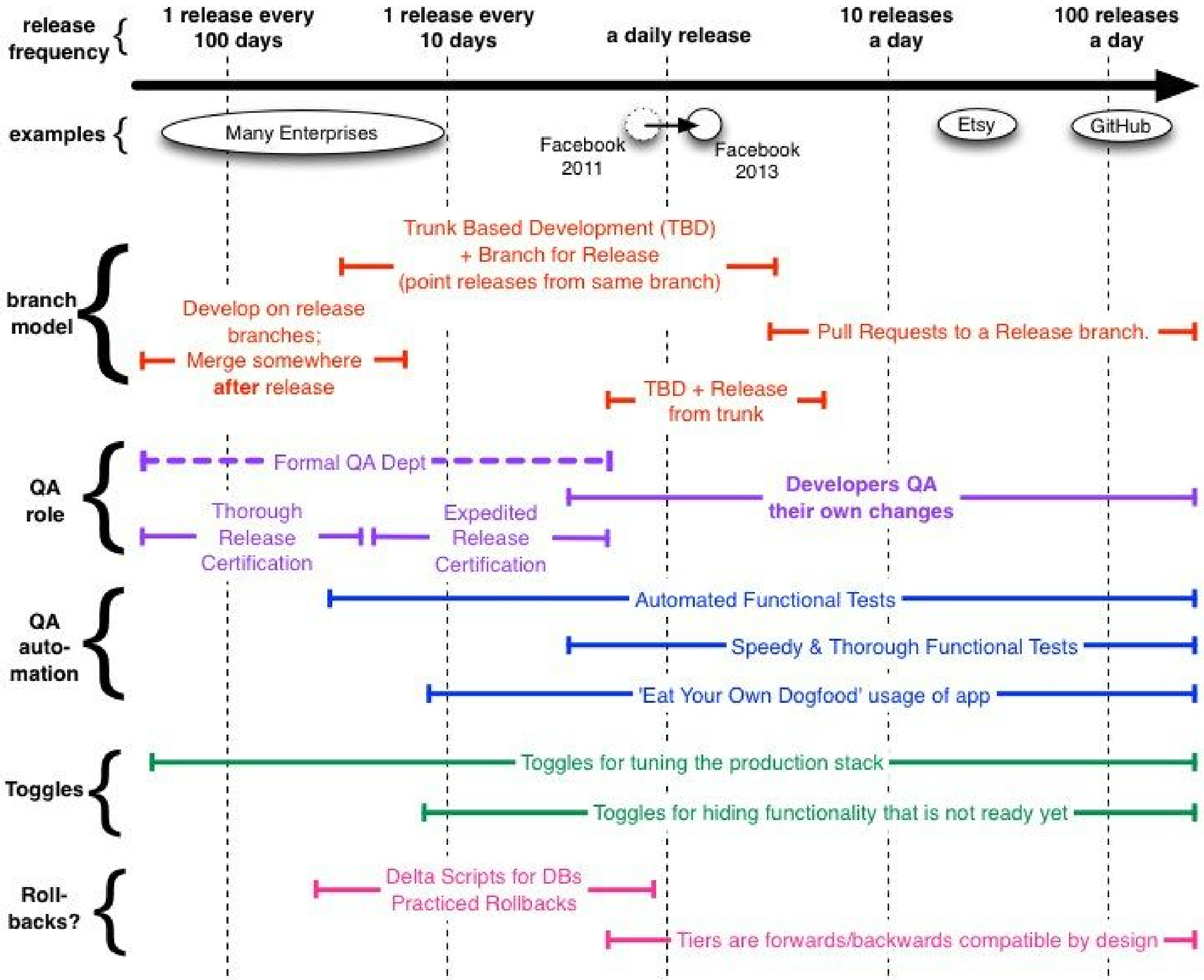
Trunk Based Development

The practice of releasing code from a single stable branch of the code instead of long lived release branches is called Trunk Based Development (TBD). TBD promotes rapid development, since no effort is spent on merging long-lived branches into the trunk. To achieve the agility and frequency of production pushes like a Facebook, two things have to fall in place i.e. Tools and Culture. Facebook releases from trunk every week on Tuesday.The push process balances the rate of innovation with risk control. IRC and IRC Bots is the preferred communication mechanism of developers interfacing with the release engineering team. So the longest time unrelease code stays in trunk is a week. Developers can also push daily. Developers must be available when their changes go live so they can support them. Release engineering cherry picks roughly 100 commits every day to push to production. A huge suite of tests both functional and performance run on every cherry picked commit. Tools like gatekeeper enable feature flags in production, allowing features to be turned on selectively for a internal employees. Tools like Phabricator and Claspin provide deep visibility into every phase of the CI pipeline starting from what is going into the build to runtime post-mortem error log and trend analysis after the changes are live. All new Facebook employees undergo a six-week bootcamp in which they're encouraged to commit new code as soon as possible, partly to overcome the fear of releasing new code. The ability to deploy code quickly in small increments and without fear enables rapid innovation. Developers take personal responsibility for code quality and face the consequence of their actions via oncall duties. Each developer is assigned karma points based on their history and success rate with pushes. Lower the karma higher the bar for your commits to be cherry picked for a daily push. Most changes are backward compatible with the extensive use of gatekeepers to gradually move API versions or schemas.The key KPIs for success of trunk based development are 1. Unique developers per week 2. Commits per month 3. Codebase size & 4. # of Cherry Picks in Production every day. [Facebook-Release-Process]

Facebook's version of the deployment pipeline, showing the multiple controls over new code. [facebook-development]


code development and testing by engineer
review by peer engineer using phabricator
requested changes and fixes
checkin and automated regression testing
bug fixes
"latest" (internal use)
release and perflab testing
sunday afternoon monday
push
H1 deploy (internal)
H2 deploy (1% users)
H3 deploy (full)
tuesday afternoon (contributing engineers on call)
gatekeeper switch on (full or selected subset of users)

Mapping Branching models to Release Frequency - [Paul Hammant]


release frequency
1 release every 100 days
1 release every 10 days
a daily release
10 releases a day
100 releases a day
examples
Many Enterprises
Facebook 2011
Facebook 2013
Etsy
GitHub
branch model
Trunk Based Development (TBD) + Branch for Release (point releases from same branch)
Develop on release branches; Merge somewhere **after** release
Pull Requests to a Release branch.
TBD + Release from trunk
QA role
Formal QA Dept
Developers QA their own changes
Thorough Release Certification
Expedited Release Certification
QA auto-mation
Automated Functional Tests
Speedy & Thorough Functional Tests
'Eat Your Own Dogfood' usage of app
Toggles
Toggles for tuning the production stack
Toggles for hiding functionality that is not ready yet
Roll-backs?
Delta Scripts for DBs Practiced Rollbacks
Tiers are forwards/backwards compatible by design

14 Monitoring - Is my System Healthy ?

When it comes to monitoring a general philosophy to follow is “if it moves graph it”. “If it doesn’t move graph it anyway it maybe taking a break”. Cultivate a mechanical sympathy by understanding numbers every programmer should know. [interactive-latency]. Without good tooling and monitoring in an application that is composed of hundreds of microservices there is no way of finding out who is who, who is talking to what and who depends on what. New ways of aggregation of monitoring data are required since categorization by IP address is no longer practical. The act of putting something in the cloud should result in instantaneous routing, monitoring and graphing. Any automated threshold analysis has to account for this high rate of change since everything looks unusual and there is no notion of normal in these transitional systems. An ecosystem of open source tools enables collection, logging, filtering and graphing of logs and metric data. The key questions that need to be answered are - Is my system healthy ? What capacity do I need for my apps ? Is my system running at optimal efficiency ? Have I made the right tradeoffs for performance and high availability ?

Centralized Logging

When it comes to logging, we really like the logback implementation. Spring Boot uses the commons-logging facade over any pluggable log implementation including logback. To understand this quagmire of java logging implementations read [gordian-knot]. Establish a standardized logback log pattern across all microservices. For instance

```
%d{yyyy-MM-dd HH:mm:ss.SSSZ} | %-5level | %X{correlationId} | %thread |
%logger{1} | %m%n
```

In the past application and system logs were primarily post mortem artifacts whereas today logs are another source of real time monitoring. Persist log information by draining logs to a third-party log management service. See [CFandLogstash ,3rdPartyLogMgmt] for documentation on capturing logs from CF system and application components and draining them to an external log management and analysis system like Splunk or logstash. Pivotal CF has native integration with ELK/logstash via a Pivotal CF Elastic Runtime Tile. This MUST be leveraged to aggregate log data. Once the logs show up in kibana you can do advanced querying on unique IDs and setup custom dashboards. The ELK stack combines Elasticsearch, Logstash and Kibana to deliver actionable insights. ElasticSearch and Kibana can be run on open source CF with some effort in the following ways - 1. [running-elk-for-a-dollar] or 2. [logsearch-boshrelease] [multi-tenant-logsearch-boshrelease]

X-Ray Microservice Call Flow

It is key to put in place a system of trace collection and monitoring that allows a sampling of a representative subset of requests to your application, and traces all RPCs made by your application when handling those requests. The traces will find performance bottlenecks in your application. These can include:

- Unnecessary service calls that are not needed to process the request
- Repeated service calls to get the same data
- Serial service calls that can be batched or executed in parallel

Monitoring dashboard should allow you to view the latency distribution for a set of requests and extract trace samples for different latency percentiles by analyzing the traces.Compare performance of two sets of requests before and after a release by comparing the traces for requests received. [google-cloud-trace]

Distributed Tracing

In a complex, large-scale distributed system composed of hundreds of microservices developed by different teams in different languages spanning multiple data centers, tools that aid in understanding system behavior and reasoning about performance issues are invaluable. [dapper-distributed-tracing]. Inspired from Google's Dapper paper, two such large-Scale Distributed Systems Tracing Infrastructures have emerged - 1.Twitter's Zipkin and 2. Netflix's Servo. A distributed tracing infrastructure not only helps with firefighting and inferring service dependencies but also helps with finding untapped performance optimizations, such as unnecessary serial requests along the critical path, end-user resource consumption patterns, needless queries, slow database SELECTs, incorrect service timeouts, expensive query patterns, query cost, long tail request latency. Finding and correcting these types of performance bottlenecks helped both Google and Twitter. They key to enabling distributed tracing is the ability to follow distributed control paths with near-zero intervention from application developers by relying almost entirely on instrumentation of a few common libraries

Synthetic Monitoring

Monitor production systems with a combination of active monitoring—also known as synthetic-user monitoring (SUM) or, more commonly, synthetic transaction monitoring (STM) and passive monitoring i.e. real user monitoring (RUM) to monitor production systems real time and to effect remediation when new features are to be rolled back based on user feedback or errors. Semantic monitoring uses tests to continuously evaluate your application, combining test-execution and real time monitoring.

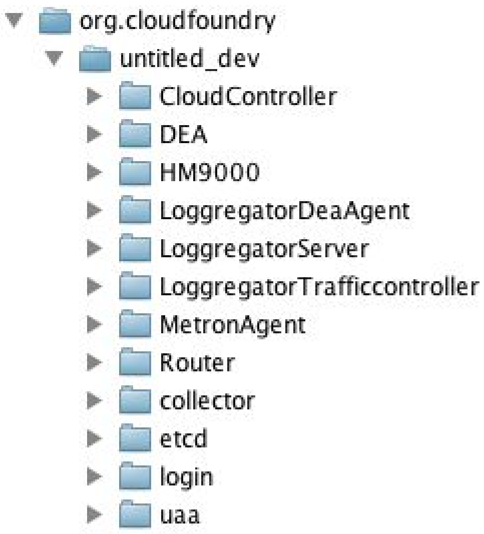
Cloud Foundry

The Operations Manager is a web application that you use to deploy and manage a Pivotal CF. Pivotal Ops Metrics is a JMX tool for the Elastic Runtime deployed using the operations manager to help monitor your installation and assist in troubleshooting, Pivotal Ops Metrics collects and exposes system data from Cloud Foundry components via a JMX endpoint. By having Ops Metrics available, we now have a JMX interface for host-level statistics on every VM running in our Pivotal CF environment. These statistics include CPU, disk, load average and a "is healthy" check. The components of cloud foundry i.e. Cloud Controller, DEA, Health Manager, Loggregator, Router and the UAA post in excess of 150 JMX metrics. These metrics need to be aggregated and analyzed continuously for Healthy System monitoring/alerting, Capacity planning (i.e., need more DEA's) and diagnosing issues (streaming, tailing the logs).

Below are some metrics to monitor on Ops Metrics, divided by component, as shown on the JMX tree. The list of important metrics can be found online [using-pivotal-metrics].

```
java -jar cmdline-jmxclient-0.10.3.jar <username>:<password> <IP>:<port>  > all-mbeans.txt
```

- username:password are from the "JMX Provider Credentials" provided during the Ops Manager tile installation. Retrieved from the "Credentials" tab on the Ops Metrics tile.
- The Ops Metrics VM's IP address available from the "Status" tab on the Ops Metrics tile.
- '44444', is the default port that JMX is configure to listen on in Ops Metrics



```
▼ org.cloudfoundry
  ▼ untitled_dev
    ▶ CloudController
    ▶ DEA
    ▶ HM9000
    ▶ LoggregatorDeaAgent
    ▶ LoggregatorServer
    ▶ LoggregatorTrafficcontroller
    ▶ MetronAgent
    ▶ Router
    ▶ collector
    ▶ etcd
    ▶ login
    ▶ uaa
```

Some of the important CF system component KPIs to monitor are listed below:

CloudController

```
cc.http_status.1XX
cc.http_status.2XX
cc.http_status.3XX
cc.http_status.4XX
cc.http_status.5XX
cc.requests.outstanding
log_count[level=error]
log_count[level=fatal]
log_count[level=warn]
```

DEA

```
can_stage[stack=lucid64]
reservable_stagers[stack=lucid64]
available_memory_ratio[stack=lucid64]
log_count[level=error,stack=lucid64]
log_count[level=fatal,stack=lucid64]
log_count[level=warn,stack=lucid64]
```

HealthManager

```
HM9000.HM9000.NumberOfAppsWithAllInstancesReporting
HM9000.HM9000.NumberOfAppsWithMissingInstances
HM9000.HM9000.NumberOfCrashedIndices
HM9000.HM9000.NumberOfCrashedInstances
HM9000.HM9000.NumberOfDesiredApps
HM9000.HM9000.NumberOfDesiredAppsPendingStaging
HM9000.HM9000.NumberOfDesiredInstances
HM9000.HM9000.NumberOfMissingIndices
HM9000.HM9000.NumberOfRunningInstances
HM9000.HM9000.NumberOfUndesiredRunningApps
```

Loggregator

```
LoggregatorServer.httpServer.numberOfMessagesDroppedInParseEnvelopes
LoggregatorServer.httpServer.numberOfMessagesUnmarshalErrorsInParseEnvelopes
LoggregatorServer.httpServer.numberOfMessagesUnmarshalledInParseEnvelopes
```

Router

```
router.rejected_requests
router.latency.1m[component-={NAME},index={#}]
router.responses[component-={NAME},index={#},status=xxx]
router.responses[component-={NAME},index={#},status=2xx]
router.responses[component-={NAME},index={#},status=3xx]
router.responses[component-={NAME},index={#},status=4xx]
router.responses[component-={NAME},index={#},status=5xx]
```

Monitoring DEAs

DEAs provide the core runtime engine of CF. It is critical to monitor the performance of the DEA VMs in CF. DEA VMs provide the warden containers used to stage and run droplets. Set up a report for each individual DEAs `available_memory_ratio` to identify hot spots and then an overall average of both. The implementation will need to be customized for the APM tool. These are the formulas that will be most useful for operations staff and some good info. for app teams.

Assuming 5 DEA's and each having 16GB ram per VM. Each DEA VM should run with `~30%` headroom. If we lose one DEA, all the app load can be transferred onto the remaining VMs without degrading the applications while bosh rebuilds the fallen machine. It is for the same reason that each app in production should run at least two instances for maintaining uptime through upgrades and high availability.

Monitor/Alert the following per DEA.

- `mem_free_bytes < 3221225472(3 GB). This accounts for the OS RAM usage as well, which is why the number is smaller.`
- `avilable_memeory_ratio < 0.3`
- `avilable_disk_ratio < 0.3`

Create an alert that monitors overall capacity of the system. If any of these drop below 30% deploy a new DEA

- `(sum( available_memeory_ratio) / num_dea) < 0.3`
- `(sum( disk_memeory_ratio) / num_dea) < 0.3`

Implicit in this design is that no single application instances consumes more than 30% of the ram available on any one DEA. Double check with your app teams that it is indeed the case.

Monitoring orgs and spaces

The platform does not emit metrics about memory usage for orgs and space. The Cloud Controller API does offer endpoints which provide this information. http://apidocs.cloudfoundry.org
See this quick and dirty python script that gets the job done and can be used as a starting point.

Cloud Foundry Metrics Firehose

The "Firehose" feature allows operators to receive a stream of all logs and metrics flowing through the CF system. This currently includes application logs, application response time metrics, and CF component metrics such as cpu and memory utilization, and many more types to come in the future. The firehose is accessed by opening a websocket connection to one of the loggregator traffic controllers. A new client library called noaa has been released, which provides some examples of how to view and make use of the data. The firehose metrics catalog can be found on [github].

Metric Event Streams

The Spring Boot Actuator module provides a number of endpoints to help monitor and manage your application when it's pushed to production. Metrics for all HTTP requests are automatically recorded. In addition to HTTP request/response metrics, system metrics (`processors`, `uptime`, `instance.uptime`, `gc.xxx.count`, `systemload.average`) and DataSource metrics (`datasource.xxx.max`, `datasource.xxx.active`) are also exposed. These metrics should be dumped into a big data lake based on hadoop for raw collection and analyzed in real time with Spring XD leveraging anomaly detection [breakout-detection] and other [practical-data-science] techniques.

The endpoints can be accessed via HTTP endpoints or JMX. Auditing, health and metrics gathering can be automatically applied to your application. Spring boot out of the box comes with a metrics endpoint with gauge and counter support. Consider adding your own public metrics based on the advice below and persisting the metrics to a back end repository like Redis Cloud. Another appealing option is to leverage the Codahale MetricsRegistry support or Dropwizard metrics and publishing the metrics to an aggregating server like ganglia or graphite. Graphite metrics can be viewed using the Grafana dashboard. For a detailed list of tools that work with graphite see [graphite-tools]. It is critical to setup just the right amount of filtering in Grafana and Metrics metrics dashboards so as not to drown in the data. For a field practitioners guide to Graphite monitoring please see [practical-graphite]. For a Cloud Foundry Deployment, leverage Ops Manager to get an aggregate view of system, application and service metrics.

Another option when it comes to application metrics is to use StatsD, a front-end proxy for the Graphite/Carbon metrics server. StatsD has a number of client libraries and numerous server implementations [Etsy-StatsD].

The following actuator endpoints are available for spring-boot: [learning-spring-boot]

Actuator Endpoint	Description
`/autoconfig`	This reports what Spring Boot did and didn't autoconfigure and why
`/beans`	This reports all the app and spring internal beans app configured in the application context
`/configprops`	This exposes all configuration properties
`/dump`	This creates a thread dump report in JSON
`/env`	This reports on the current system environment

`/health`	This is a simple endpoint to check life of the app
`/info`	This serves up custom content from the app
`/metrics`	**This shows counters and gauges on web usage**
`/mappings`	This gives us details about all Spring MVC routes
`/trace`	This shows details about past requests

Customization of Spring Boot Actuator Endpoints

Spring Boot apps will typically be registered as services in one or more PaaS environments. Service discovery of these microservices entails checking the health of the application via a pre-registered end-point. This is where customizing the output of the /health end-point specific to the application function is critical for the sustained availability of the service. Health information is collected from all HealthIndicator beans defined in your ApplicationContext. Spring Boot includes a number of auto-configured HealthIndicators and you can also write your own. [customizing-endpoints] [custom-health-indicators]. These management endpoints should be protected with spring-security with a different set of security credentials and roles than normal users of the applicaton.

JMX Monitoring

If you prefer JMX for monitoring then use the Jolokia for JMX over REST support provided by Spring Boot. MBeans can be accessed with with the `/jolokia` endpoint. For instance if we push an application to the URL fizzbuzz22.cfapps.io then the HeapMemoryUsage mbean value can be retrieved using the a GET request like this http://<ROUTE>/jolokia/read/java.lang:type=Memory/HeapMemoryUsage or the more general form `http://<CF_APP_URL>/jolokia/read/<JMX_OBJECT_NAME>.`

Jolokia has a very expressive and function rich protocol to retrieve mbeans via synchronous, asynchronous fetch in batch, with a callback or scheduled mode.. There are also a number of clients Javascript, Java and Perl to talk to the Jolokia mbean server over HTTP. Application instance JMX metrics can be integrated into the Pivotal CF Operations Manager or the JMX mbean statistics aggregator of your choice. In order to enable JMX mbeans you will need to add the following dependencies to your application classpath via maven or gradle -

```
org.jolokia:jolokia-core
org.springframework.boot:spring-boot-starter-actuator
```

Most containers and PaaS restrict incoming traffic to only HTTP or HTTPS. This necessitates the streaming of JMX metrics over HTTP. Jolokia’s HTTP/JSON bridge for remote JMX access enables consumption of JMX metrics over HTTP. Jolokia transparently bridges the JMX world, talking to clients on the frontend via a REST-like protocol and to a JMX MBeanServer on the backend.

End-To-End Request Flow Correlation

End-to-End request flow measurements are very important. Use correlation IDs across request calls to track microservice call flow to nail down nasty bugs and track the originating event. It is critical to understand the latency between the edge service to the API Gateway and from the API gateway fanout to all the downstream services. This is achieved by inserting correlation GUIDs and transaction IDs in HTTP headers, correlation identifiers in messages and and tracking them across requests. Spring Integration provides the CorrelationStrategy abstraction to correlate with an existing correlation key (HTTP request header). The intent is to create a mapped diagnostic context for an end to end synchronous and asynchronous request-response flow [mdc-logback].

- Correlation IDs can be implemented as servlet filters. A better way to inject this cross cutting concern across microservices code is by defining an aspect. Building on the servlet filter approach, such a capability is provided by the com.ofg:micro-infra-spring-base module. The Correlation ID support works across Spring Reactor threads, Servlet 3.0, async servlets, Spring MVC Controllers, Spring RestTemplate and Apache Camel.
- The Zipkin project provides an instrumentation for google's Dapper-style distributed tracing in services. Zipkin provides detailed structured logging of individual requests, full view of a request's path through SOA and customization through your own annotations. ZipkinTracer support is baked into the Finagle library. Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers at Twitter.It should be reasonably straightforward to add Zipkin tracing support to Spring boot and other protocols and libraries.
- Cloud Foundry services like AppDynamics and NewRelic also provide this capability by automatically instrumenting the java code via javaagent's to insert these unique IDs and then visualize the request flow using a dashboard.

To give you a concrete use case. The Cloud Foundry PaaS sets the following environment headers on every application

```
VCAP_APPLICATION={"instance_id":"451f045fd16427bb99c895a2649b7b2a",
"instance_index":0,"host":"0.0.0.0","port":61857,"started_at":"2013-08-12
00:05:29 +0000","started_at_timestamp":1376265929,"start":"2013-08-12 00:05:29
+0000","state_timestamp":1376265929,"limits":{"mem":512,"disk":1024,"fds":16384}
,"application_version":"c1063c1c-40b9-434e-a797-db240b587d32","application_name"
:"styx-james","application_uris":["styx-james.a1-app.cf-app.com"],"version":"c10
63c1c-40b9-434e-a797-db240b587d32","name":"styx-james","uris":["styx-james.a1-ap
p.cf-app.com"],"users":null}
```

The Cloud Foundry PaaS sets the following Request Headers on every request

- `X-CF-InstanceID`
- `X-CF-ApplicationID`
- `X-Vcap-Request-Id`

Therefore the `VCAP_APPLICATION` environment variable **`(application_name, application_version) + Request headers`** have everything you need to create an application request correlation ID that enables request correlation across JVM process boundaries. You correlate app name via `X-CF-ApplicationID + VCAP_APPLICATION: application_version`. In a microservices architecture where one request could potentially traverse multiple JVMs this is very useful. The application name stays the same across pushes of an app, or follow some convention like appname_v123. So the logging system could use a rule like startswith or split the name by “_”. This enables long term trend analysis of the logs across multiple app pushes.

Log Format For Apps On CF

```
[Timestamp] [[CF Application]/[CF Application Instance]] [CF Source] [CF Space:CF Application Name] [EUID] [[Correlation ID]] [Log Level] [[Thread Pool]-[Thread Name]] [Package].[Class] - [Message]
```

Column	**Description**
Timestamp	time the message was generated
Application	The application instance name according to CF
Application Instance	The application instance number according to CF. If there are 20 instances of the application running on CF, then this indicates which number the message came from.
Source	The CF source of the message. Ex. OUT is from system out.
Cloud Foundry Space	The cloud foundry space that the application is running in. We intend to have a space for dev, test, stage, prod.
Cloud Foundry Application Name	The cloud foundry application name that was assigned when the application was pushed to CF.
UID of user	The user who's request generated the log message. It is code be a hash that represents the user
Correlation ID	A unique ID (GUID) that is generated for a user’s transaction. It will be passed through for all of the actions taken on behalf of the user. It can be used to see how the system responded to a users request.
Log Level	The log level that the application assigned. Ex. DEBUG, INFO, WARN, ERROR, etc.
Thread Pool - Thread Name	The Java thread pool name and thread name
Package	The Java package.
Class	The Java class name.
Message	The log message from the application.

Sample Log Message:

```
2015-01-15T12:47:37.73-0500 [App/1]OUT [sandbox:logproducer] aa12345
[25892e17-80f6-415f-9c65-7395632f0223] DEBUG [pool-1-thread-29]
com.kelapure.rohit.cloud.service.LogProducer - 383928 logging logging
logging...She sells sea shells on the sea shore
```

Performance Testing

Performance Testing via "Microservice Tee" Pattern - “The idea is to deploy a new version of a technology and make sure initially all requests, while serviced by the older version, get sent in duplicate to the new technology.” This real world load test uses Zuul to copy all requests to a parallel deployment that is running active profiling. This approach can also be leveraged for functional, chaos and scale testing without fear of affecting production.

Service Monitoring

Track inbound response time, response time of downstream calls, response error rates and application-level metrics. Standardise on how and where metrics are collected. Log into a standard location, in a standard format. Monitor the underlying operating system to track down rogue processes and do capacity planning. [building-microservices]. Monitor routing of services to garbage collect services that no traffic is being routing to. Do not confuse latency with throughput and optimize for something that won’t occur in production.

System Monitoring

Aggregate and correlate VM and container metrics like CPU, network and I/O together with application-level metrics. Ensure your metric storage tool allows for aggregation at a system or service level, and drill down to an individual machine. Ensure your metric storage tool allows you to maintain data long enough to understand trends in your system. Have a single, queryable tool for aggregating and storing logs. Understand what requires a call to action, and structure alerting and dashboards accordingly. Investigate the possibility of unifying how you aggregate all of your various metrics. Software load balancers like API service gateways or edge Service routers need to be constantly probing endpoints for availability and health. [building-microservices]

Automatic Baselines & Percentiles Based Alerting

Regardless of which APM solution you use for long term monitoring of system and application metrics it is critical the monitoring framework does automatic baselining and allows for percentile based alerting, reporting and dashboarding. Averages are ineffective because they are too simplistic and one-dimensional. Percentiles are an easy way of understanding the real performance characteristics of your application. They also provide a basis for automatic baselining, behavioural learning and optimizing your application with a proper focus. [averages-suck].

Next Generation Monitoring

Most alerting is based on naive thresholds. For continuous delivery at scale, organizations need to put in place 1. alerting based on percentiles and standard deviations, 2. auto scaling based on algorithms like Time series and FFT, and 3. anomaly detection based on advanced analytics like Breakout-Detection, Holt-Winters, Prelert & SumoLogic. Log analysis has to move forward from basic pattern-matching and grouping of repetitive entries to pinpointing problems before thresholds are reached. Most ops monitoring suffers from tremendous data overload. Gleaning insight from the metrics requires specialized frameworks like Kale. Thresholds are good at helping you react proactively to known issues, by triggering alerts before things go out of hand. Search allows you to investigate unknown issues, but only after they occur. Anomaly detection is the only way to react to unknown issues proactively. Donnie Berkholz is right - "Monitoring needs to go back to data science school for its Ph.D. to cope with this new generation of IT". [sysadmins-learn-datascience]

15 Conclusion

This paper covered the current set of best practices in the design and implementation of microservices based cloud aware applications. We arrived at the correct software architecture for microservices and expounded on the design, implementation, test, monitoring and deployment of cloud native microservices using Spring on Cloud Foundry. This paper provides a reference architecture for timelessness of your enterprise software leveraging the patterns, methods and tools of model and domain driven design, implementing domain driven design and hexagonal architecture.

Microservices are quickly emerging as an industry buzzword for implementing all new software systems in this Web-Scale IT world. We went under the hood and stripped microservices to their very core giving an in depth and detailed understanding of why, when and how to use them. This book focuses on the Cloud Foundry PaaS and microservices architectures that are enabled by it. Building microservices is easy with the Spring framework. Springt takes care of the boilerplate in distributed system. The benefits of microservices apply to any modern PaaS that provides a uniform way of deploying and managing software. We coherently integrated disparate threads of software design currently in play for developing services such as actor models, event based architecture, layered, hexagonal and onion architecture.

We expect software developers to come away with a swiss army knife for working with microservices and a deep understanding of various paradigms for modeling, designing and implementing the problem space of their domain on Cloud Foundry.

16 Videos & Articles

1. Java Applications in the Cloud - Slides from CF Summit 2014
2. Ben Hale - Java Buildpack Year in Review (Cloud Foundry Summit 2014)
3. Matt Stine - Cloud Foundry and Microservices: A Mutualistic Symbiotic Relationship
4. Exploring Micro-frameworks: Spring Boot
5. Pattern: Microservices Architecture
6. Microservices reading list
7. Netflix Global Edge Architecture
8. The Netflix Techblog
9. Bootiful Applications with Spring Boot
10. DevOps Guru Adrian Cockroft on DevOps + MS.
11. http://github.com/spring-cloud-sample
12. http://blog.spring.io
13. http://presos.dsyer.com/decks/cloud-boot-netflix.html
14. Domain Driven Design - Eric Evans
15. SpringOne 2gx 2014 sessions on Microservices
16. http://www.tigerteam.dk/2014/soa-hierarchy-or-organic-growth/
17. http://2014.javazone.no/presentation.html?id=cce167fb
18. http://blog.zilverline.com/2012/07/04/simple-event-sourcing-introduction-part-1/
19. http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh
20. 4Finance microservice pilot
21. http://www.infoq.com/articles/implmenting-domain-driven-design-vaughn-vernon
22. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/maximizing-cloud-advantages-through-cloud-aware-applications-paper.pdf

17 Credits

Without the support and love of my immediate family (Aaie, Papa, Rushil and Renu), writing this book would not have been possible.

This architecture has been built on the shoulders of giants like Martin Fowler, Sam Newman, Chris Richardson, Adrian Cockroft, Neal Ford, Matt Stine, Pivotal Teams, Thoughtworks and many more.

My teammates in Pivotal Services and Pivotal Field Engineers were instrumental in motivating me to start and finish this body of work.