Microservice Architecture Blueprint Strategy Episode 2

Kim Horn

Version 0.7 **1 August 2017**

The Business Value of Integration

- ✓ **APIs are a business product not a technology solution**
- ✓ **API Management is a Business Platform, not just a Technology Platform**
- ✓ **Cloud First is most effective if integration is location independent**

Best Practices

- Core services & their APIs should be managed as business products, from conception to retirement
- Integration needs to be simple and agile else the business will build up a “Shadow IT” structure
- Access control should be based on business requirements not technology constraints
- Governance must be lightweight, simple and effective
- Service standards should be designed for the future
- Keep competitive advantage functionality under internal control

API First Value

- APIs can extend the utility and reach of your business
- New products and services should offered as APIs
- Partner (B2B) integration is simpler with Web APIs
- B2B integration is no longer divergent from the B2C and B2E service model
- APIs provide the basis of modern identity services
- APIs can support significant internal productivity improvements

© 2015 Kim Horn

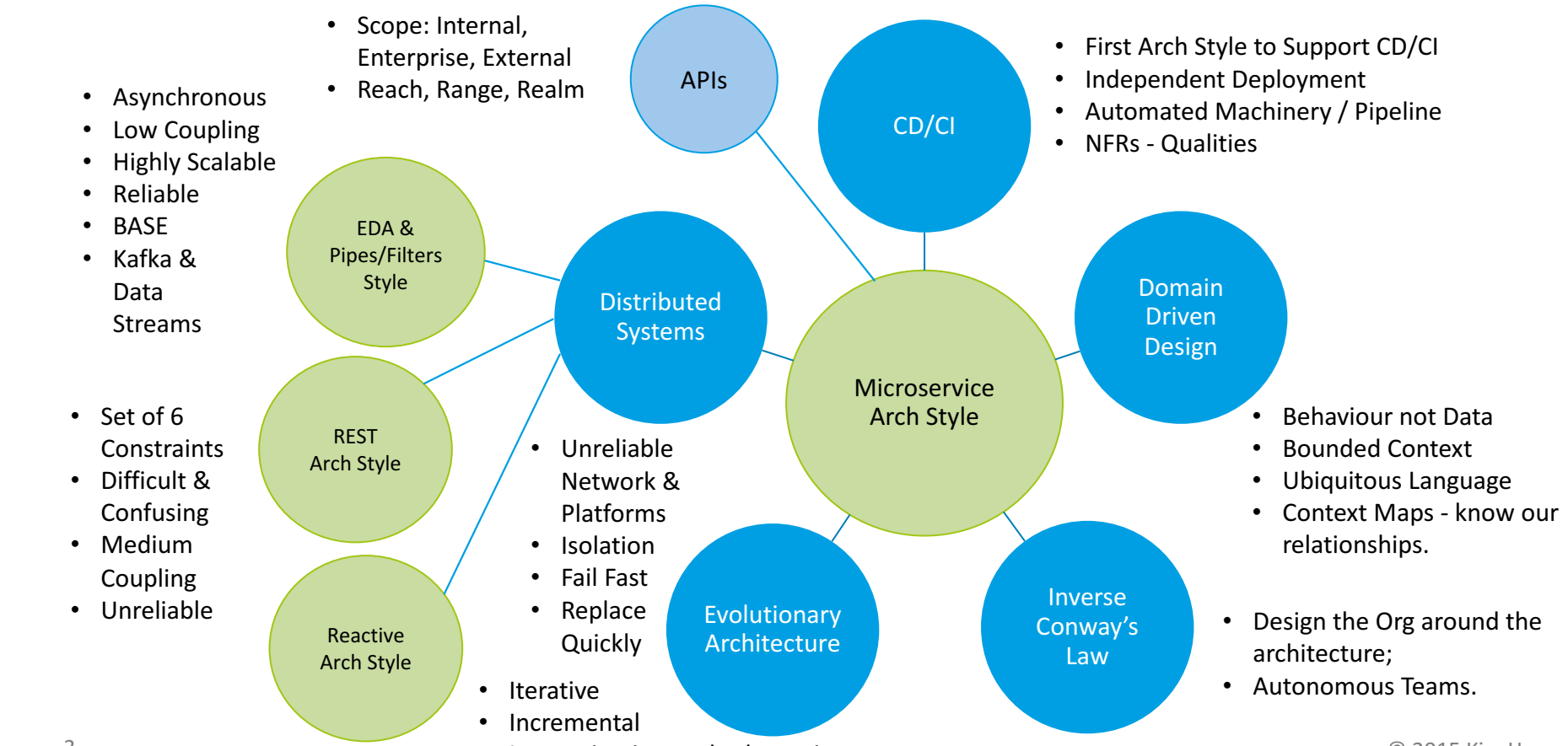
The Business Value of API Management

It is impossible to provide the platform for any digital strategy, and run an effective API program to benefit from the API economy, without full life cycle API management.

Gartner Magic Quadrant for Full Life Cycle API Management
Published: 27 October 2016 ID: G00277632

© 2015 Kim Horn

Dependencies – Practices and Styles



- Scope: Internal, Enterprise, External
- Reach, Range, Realm

- First Arch Style to Support CD/CI
- Independent Deployment
- Automated Machinery / Pipeline
- NFRs - Qualities

- Asynchronous
- Low Coupling
- Highly Scalable
- Reliable
- BASE
- Kafka & Data Streams

- Set of 6 Constraints
- Difficult & Confusing
- Medium Coupling
- Unreliable

- Unreliable Network & Platforms
- Isolation
- Fail Fast
- Replace Quickly

- Behaviour not Data
- Bounded Context
- Ubiquitous Language
- Context Maps - know our relationships.

- Iterative
- Incremental
- Innovation is standard practice

- Design the Org around the architecture;
- Autonomous Teams.

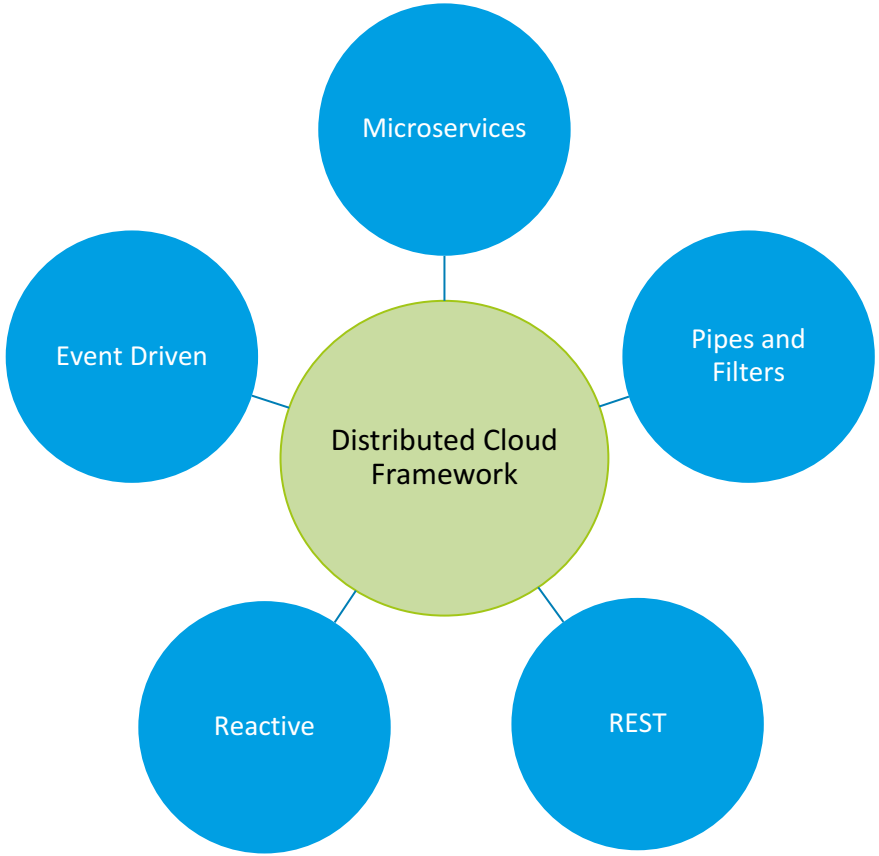
© 2015 Kim Horn

Implement by Inverting the Framework

A number of architectural styles are required to deliver quality distributed systems in house or in the cloud.

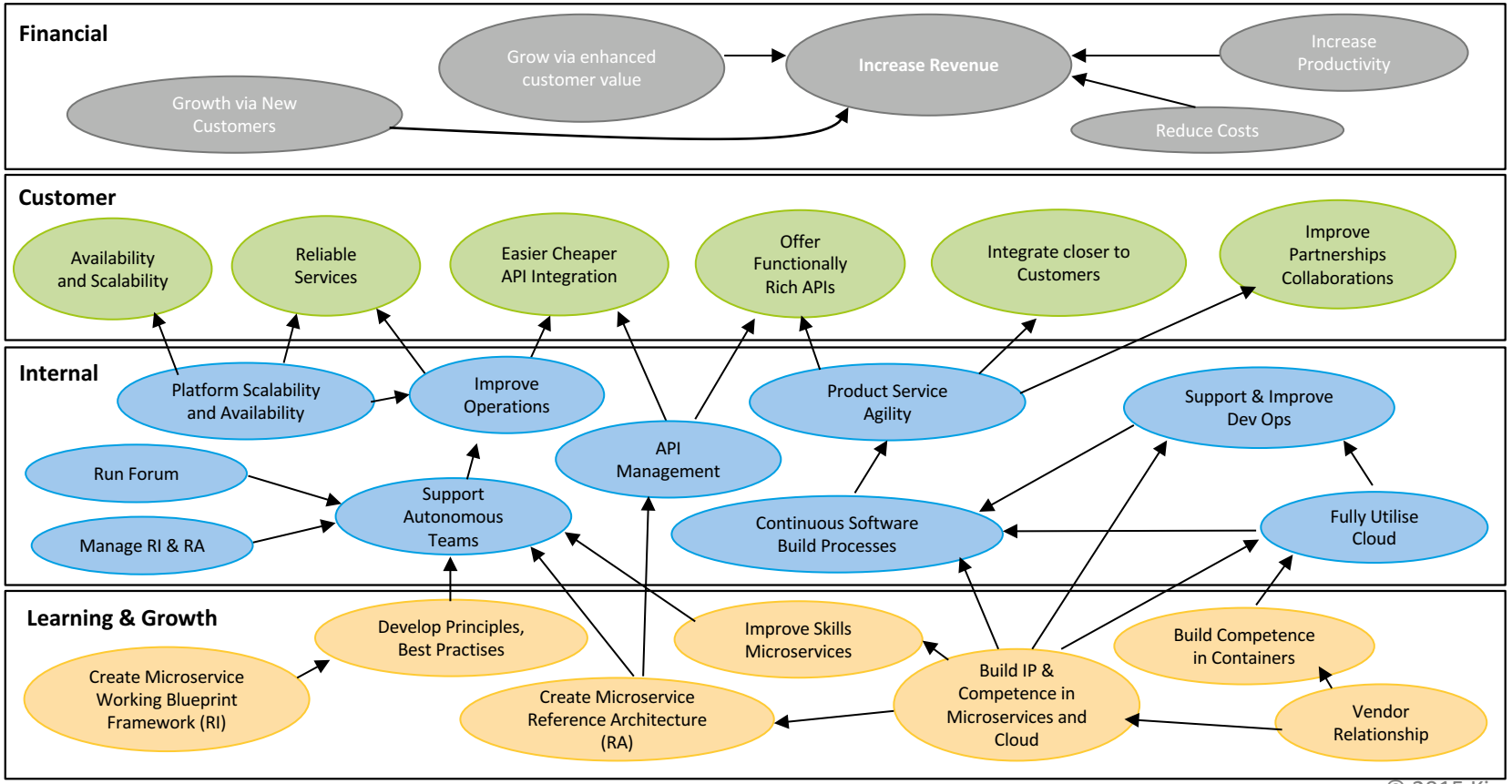
Microservices are one type of Distributed System.

Need not just Microservice Framework but a Cloud Framework



© 2015 Kim Horn

Microservice Strategy Map



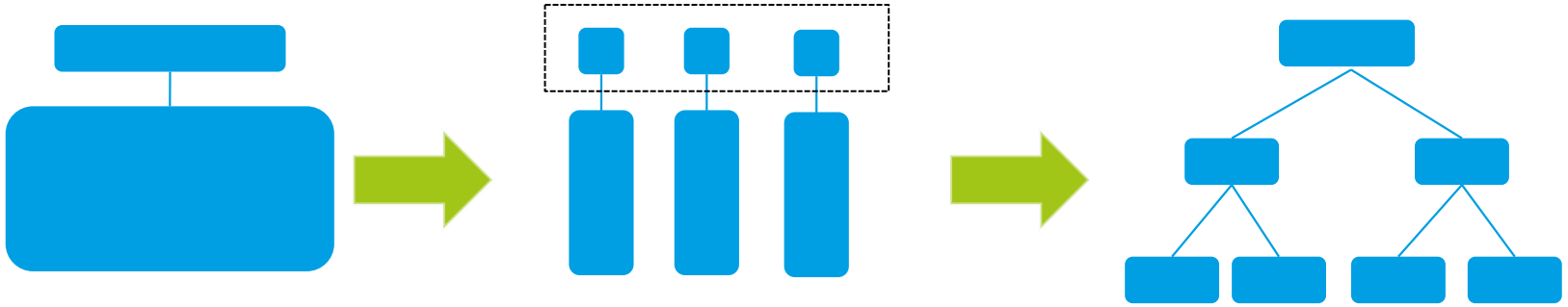
© 2015 Kim Horn

Reference Architecture

© 2015 Kim Horn

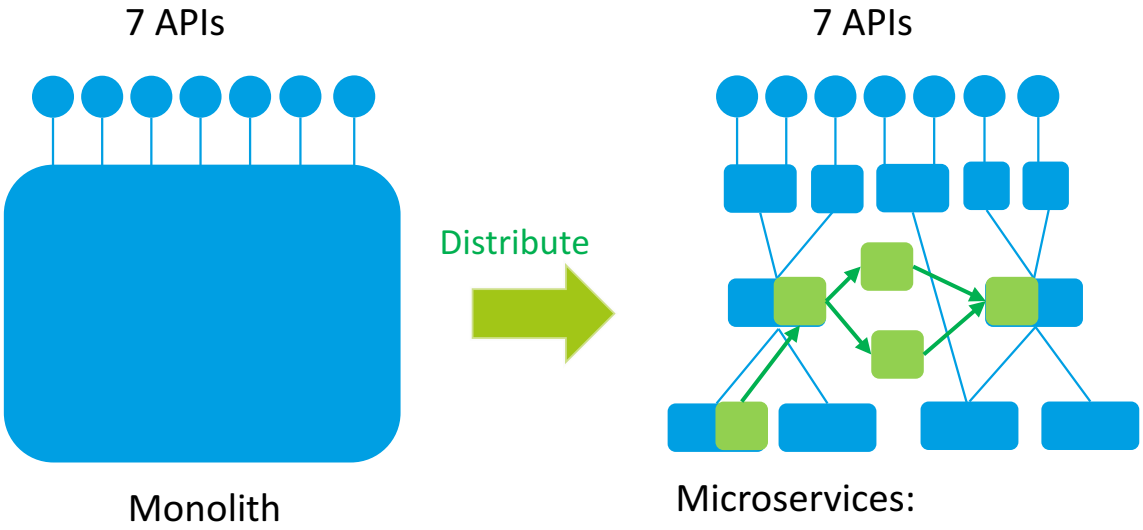
Monoliths to SCS to Microservices; a continuum

- A Self Contained System (SCS) is a vertical slice of a Monolith that is autonomous, it has its own UI, business logic and data storage. The components are integrated at the UI.
- Microservices employ distributed decomposition.
- SCS can be an intermediate state away from a Monolith to Microservices.



© 2015 Kim Horn

Microservices – are not about APIs or Integration ! !



Group	Heuristic
One	Bounded Context
Two	Layering/Tiering
	Distribution
	Exposure
	Functionality
	Generality
	Aspectual
Three	Coupling & Cohesion
	Volatility
Four	Configuration
Five	Planning & Tracking
	Work Assignment

Microservices:

- Component Decomposition (Build & Runtime)
- How to decouple workflow:
  - Choreography Vs Orchestration
  - Events - Loosely Coupled
  - No ESB - Magical Mystery Bus

© 2015 Kim Horn

Integration Architecture Subsets

	Batch	Events	Connected
Partner	B2B Batch Transfer	External Event Management	External APIs
Enterprise	Batch Hub/Spoke	Event Hub/Spoke	Internal Enterprise APIs
Local	Batch Transfer Direct	Event Point to Point	Realtime Direct Access

© 2015 Kim Horn

SOA and ESBs

SOA is an architectural style adopted by most enterprises in the last decade. Its main objective was to achieve a **higher level of reusability** and modularity by providing standard interfaces to enterprise applications as services and being able to build business processes from orchestrating these services using an orchestration and integration platform.

"We have seen so many botched implementations of service orientation - from the tendency to hide complexity away in ESBs, to failed multi-year initiatives that cost millions and deliver no value, to centralized governance models that actively inhibit change, that it is sometimes difficult to see past these problems." (Fowler, 2014).

Even though SOA concepts look like microservices from reusability and flexibility point of view, the huge investments by organizations didn't gain its expected results. *The main issue were:*

- The massive coupling that resulted between consumers and applications
- The teams running the ESB were not consumers and not applications teams, they were not invested or committed to either side of the ESB.
- Centralised governance that just adds cost and beurocracy, delays projects and kills agility.

© 2015 Kim Horn

The Re-use Myth - The focus for microservices is use, not re-use.

“there is no such thing as reusable code, only code that is reused.” Kevlin Henney

In Mythical Man Month (1974) Fred Brooks suggests that re-usable code costs three times as much to develop as single use code.

You have to use your “reusable” code three times before you break even. Writing a lot of reusable code is expensive; waste. While it may reduce the amount of code that is written it reduces it by artificially constructing supply.

Don’t plan for reuse but look for opportunities to reuse prior work. When you see the opportunity take previous work and enhance it just enough to reuse it, you only pay for what you actually need, when you need it.

Many things that are designed to be general purpose often end up satisfying no purpose. Software components should, first and foremost, be designed for use, and to fulfil that use well.

© 2015 Kim Horn

SOA and ESBs Vs Microservices

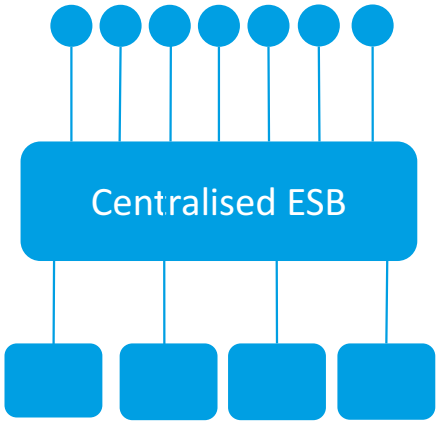
The main differences between SOA and microservices are:

- SOA services communicate over heavyweight smart-pipes (ESB);
  - microservices communicate over dump pipes with smart-end points (Fowler,2014);
- SOA aims to integrate enterprise large complex monolithic applications into enterprise-wide processes;
  - microservices aims to integrate small ‘services’ into a single project (Richardson, 2018).
- SOA is enterprise-wide based;
  - microservices are project (application) based (Wolff, 2016);
- In SOA, a business process is created as orchestration and managed by the platform;
  - A microservices architecture encapsulates business process as a higher layer microservice, at the application level, and can be replaced by another service if the process is changed (Wolff, 2016).

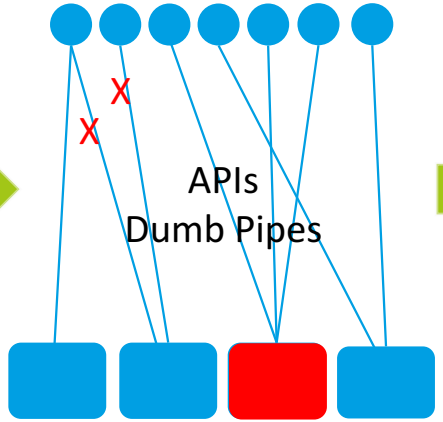
© 2015 Kim Horn

Evolve Architecture – Where's the 'Smarts' ?

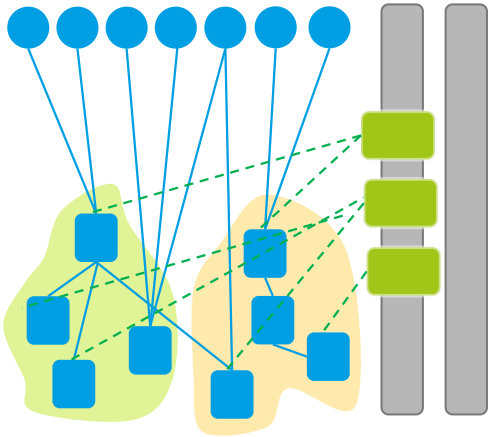
ESB holds 'Smarts'



Force 'Smarts' to End Points



'Smart' Frameworks



Monoliths via ESB

- ESB not equal SOA
- Low Scalability
- Low Availability
- Low Reliability/Resiliency
- Low Flexibililty
- Blame the ESB
- No one pays for ESB

Monoliths Direct APIs

- Low Scalability
- Low Availability
- Low Reliability/Resiliency
- Inappropriate Coupling
- Reduced Productivity
- Improved Traceability
- Producer Driven
- Reduced Autonomy

Microservices + Framework

- High Scalability
- High Availability
- High Reliability/Resiliency
- SOC / Decentralised
- Fast Flexible Build
- Complex Traceability
- Consumer Driven

© 2015 Kim Horn

Framework Foundation Capabilities

Support Arch Styles: REST, Reactive, EDA, Pipes Filters….

Scalability, Elasticity

Availability;

Workload distribution, shedding;

Service Discovery;

Reliability, Resiliency;

Async Notifications, Events;

Distributed Management and Operations;

Deployment orchestration;

Isolation, health, faults;

Monitoring;

Security;

Logging;

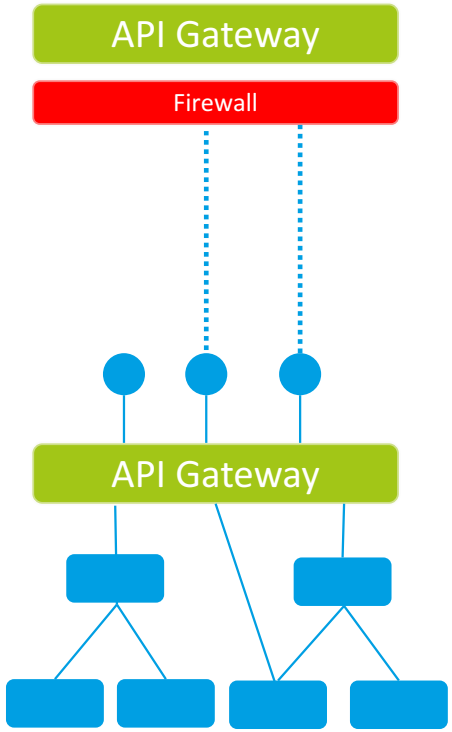
Tracing.

- Reliability requires a foundation
- To support autonomous teams capabilities need to be distributed, not centralised;
- Same building blocks used across architecture to provide a cohesive framework, from gateway to backend services; not a disparate mix of commercial products;
- Apply based on context, Quality Requirements, e.g. Internal Vs External Exposure – different needs.

© 2015 Kim Horn

Internal Vs External Exposure – Gateways Inside and Out

Expose APIs Externally



- Partner Mngt:
  - Admin
  - Portal
  - Billing
- Security – Strong
- Service Level Mngt
- Throttling
- Back to Front End
- Expose minimal mappings
- Basic Composition

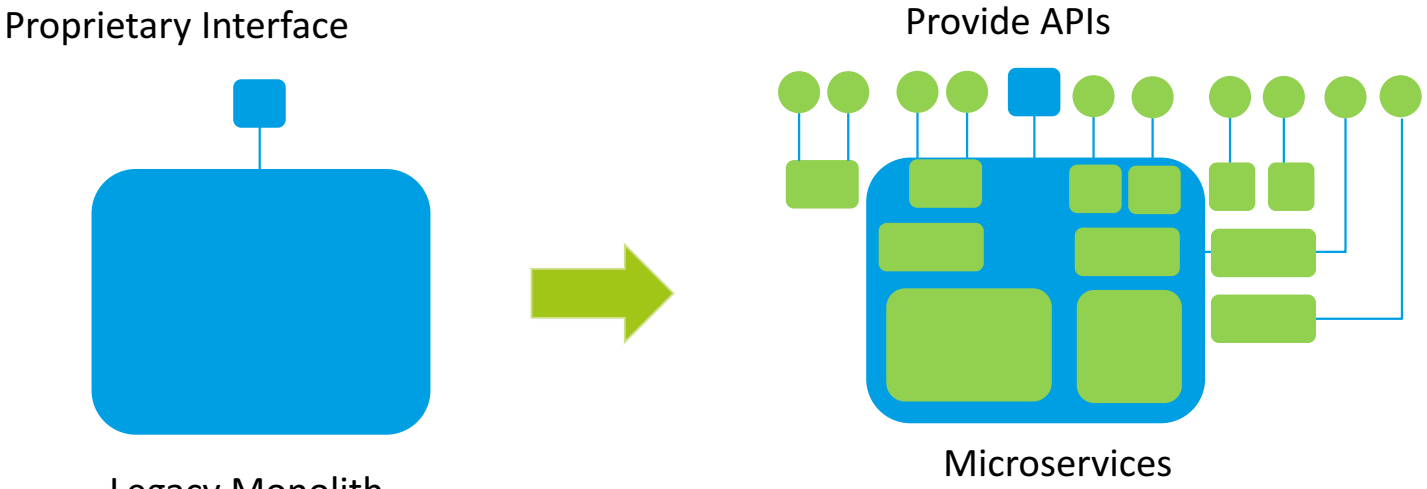
- Load Shedding
- Security Basic
- Real Time Service Discovery
- Expose all Mappings
- Orchestration and Composition

Internal App

Manage the Bounded Context

© 2015 Kim Horn

Microservices and Legacy Systems – Strangle Them



Slowly Strangle the System:

- Strangler Pattern
- Wrapper Pattern
- Use Gateway to route old and new

© 2015 Kim Horn

Practice 5 – Deliver Working Reference Implementation

"Deliver immediately deployable, production-ready platforms and APIs rather than document-based Reference Architectures (RAs) that are slow and difficult to apply……

In Agile, RAs consisting of reference models and written documents outlining design guidance and specifications tend to become shelfware."

Working Production-Ready Standards for:

- Platforms;
- Pre-built virtual machines or containers – use off the shelf;
- API libraries.

Use containers (or VMs) to lower cost of use.

© 2015 Kim Horn

Reference Implementations

“As we’ve done more Agile, we’ve realized that Architecture is something that we teach, not something that we own”.

- CEB

© 2015 Kim Horn

Reference Implementation Benefits

- Tangible, not abstract, product;
- Immediately usable by developers and architects;
- Netflix idea: “Follow the Paved Road”
- Default to a sound architecture;
- Baking in quality;
- Faster and easier to use;
- Objectively testable, not just theory;
- Accelerate speed to market;
- Minimise decision points;
- Drives innovation in other areas (bits not implemented);
- Provides a test reference for alternate implementations and product purchases;

We created a Blueprint Framework for the Reference Implementation

© 2015 Kim Horn

Netflix says:

“Our REST API, while very capable of handling the requests from our devices in a generic way, is optimized for none of them. This is the case because our REST API focuses on resources that are meant to be granular representations of the data, from the perspective of the data……

To solve this issue, it is becoming increasingly common for companies (including Netflix) to think about the interaction model in a different way……

For Netflix, **our goal is to take advantage of the differences of these devices rather than treating them generically**. As a result, we are handing over the creation of the rules to the developers who consume the API rather than forcing them to adhere to a generic set of rules applied by the API team. In other words, we have created a **platform for API development**.”

© 2015 Kim Horn

Blueprint Principles

- Based on and demonstrates the use of Documented Cloud Patterns;
- Provides Implemented Deployed Framework based on working code;
- Based on Trusted, well documented Open Source re-usable components that implement the Cloud patterns;
- Deployable anywhere: data center, AWS, Azure, Google, My Laptop;
- Employs best practices proved by best of breed organisations;
- Integrated and Cohesive Framework of components, that can be used individually or as a set.
- Guidelines to help apply the right patterns and components to meet NFRs.

© 2015 Kim Horn

Goals of the Blueprint

- Provide a proven and **robust open source alternate** to expensive and limiting commercial products;
- Provide an E2E **integrated and cohesive stack of plug and play components** combined into a lightweight framework. There are many commercial products in this space but they do not integrate or work seamlessly together.
- **Stop guessing capacity needs**, decisions to fix before will be wrong, either resources sitting idle or limited, causing failure. Provide elasticity; use as much or as little capacity as you need, and scale up and down automatically.
- Provide mature components that exhibit **proven scalability, availability, security, reusabilty, conceptual integrity, manageability and buildability.**
- Provide **distributed communications synchronously and asynchronously**, where applicable, providing streams for business events and data.
- Provide a **Cloud enabled** set of components that run anywhere.
- Choose components with **quality documentation** and readily available examples; to nurture our distributed systems capabilities.
- Capitalise on the vast experience of **best of breed organisations**;

© 2015 Kim Horn

Further Goals

- Accept that both **networks and platforms are unreliable** and provide graceful mechanisms to prevent faults from cascading into complete system failure.
- Accept that teams will adopt what they like to deliver (tools, process, tech), only API is exposed;
- Distributed processes will come and go, change their locations, e.g. with faults.
  - Provide a way to **discover the locations of the healthy services**;
  - Provide mechanism to **detect failure** and allow components to fail fast;
- Provide **decomposition heuristics** that help determine the right services with right level of granularity (coarse to fine);
- **Automate Change:** Supports best and creative practices for build with fully automated deploy;
- Provide mechanisms to integrate non-java, and non-framework java components;
- Provide services to **consistently and reliably distribute configuration, live, in real time,** to many services running in many environments;;
- Provide approaches to maintain **visibility into the composite behavior** of a system that emerges from the evolving topology of maybe 1000s of autonomous services, so that the system can be monitored, managed and operated;

© 2015 Kim Horn

Drivers and Strategy Map

© 2015 Kim Horn

Main Drivers

- Support Autonomous teams;
  - they have the freedom to get things done, any way they want;
- Support Smaller Teams;
- Teams aligned to support as a business area, not a technology;
- Continuous Integration / Continuous Delivery;
  - Eliminate big complex releases, e.g. 4-5 months across the board.
  - Always integrate and build,; always have a working release;
- Improve the Agility of Product Innovation;
- Improve reliability and scalability;
- Reduce the need for complex capacity planning.
- Make infrastructure faster and easier to procure, e.g. 3 month wait for a VM.

© 2015 Kim Horn

Top 6 Microservice Quality Drivers

Qualiity	Comment	Related Quality
Scalability	Provide applications that can scale up to meet new demands or down to reduce costs, when required. Eliminate capacity planning.	Performance, Throughput, Elasticity.
Availability	Provide applications that are operating when and where needed, and that are in a known state of health.	Reliability, Testability, Maintainability, Serviceability
Reliability	Provide distributed systems that are in a known state, robust to failure, with components that fail fast and replaced fast.	Reproducability, Immutability
Integratability	Provide distributed systems that can easily exchange data and cooperate both internally and with customers.	Manageability
Buildability	Support small autonomous and distributed agile teams, aligned to a business capability not a technology. Support devops approaches, continually deploying working systems.	Modularity, Development Distributability, Variability
Flexibility	Provide systems that allow business changes rapidly, in an agile manner. Systems that can be deployed anywhere, internally or in the Cloud.	Extensibility, Portability Interoperability, Reusability

© 2015 Kim Horn

Standard Qualities (NFRs) + Support for CD/CI

Manifest	Performance
	Availability
	Usability
	Reliability
	Accessibility
	Mobility
Developmental	Buildability
	Testability
	Understandability
	Code Quality Measurability
	Conceptual Integrity
	Budgetability
	Planability
	Traceability
	Development Distributability
	Modularity / Packagability

Operational	Throughput
	Security
	Manageability
	Maintainability
	Serviceability
	Deployability
	Reproducibility
Evolutionary	Scalability / Elasticity
	Variability
	Flexibility
	Extensibility
	Reusability
	Portability
	Substitutability
	Integrateability

Red = CD/CDP Driver

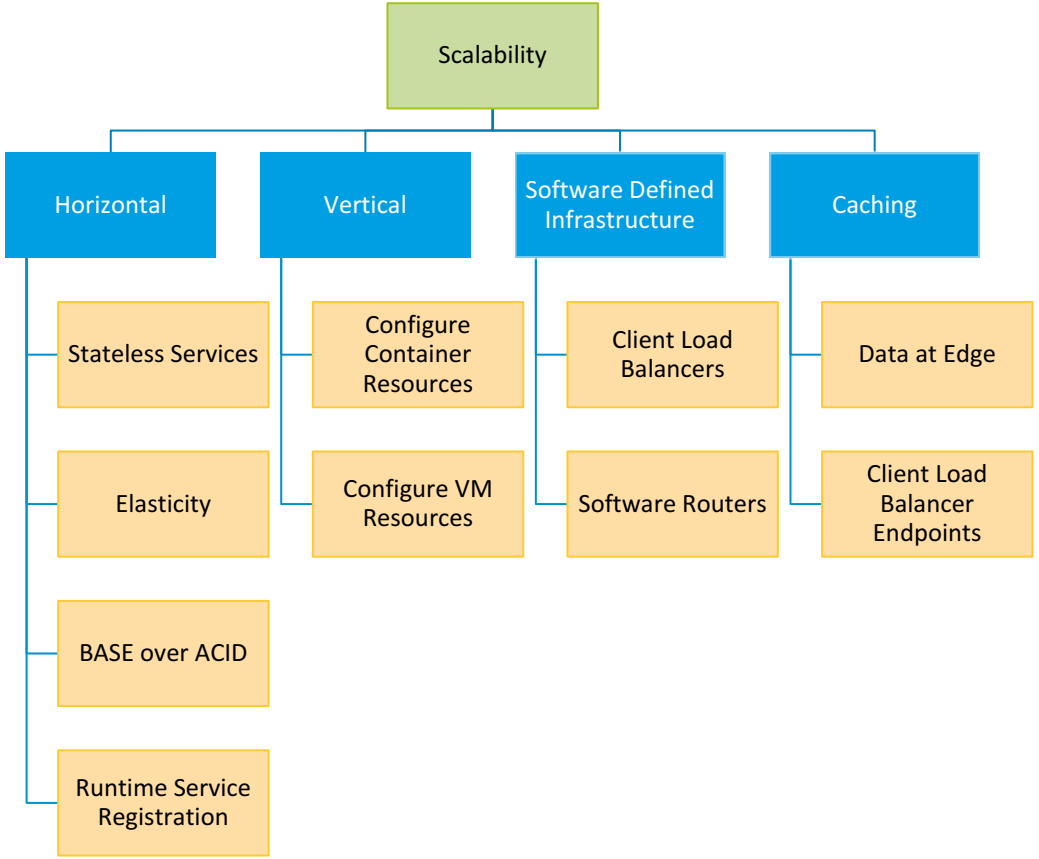
Purple = Immediately Impacted by CD Architecture.

© 2015 Kim Horn

Solve Quality with Tactics

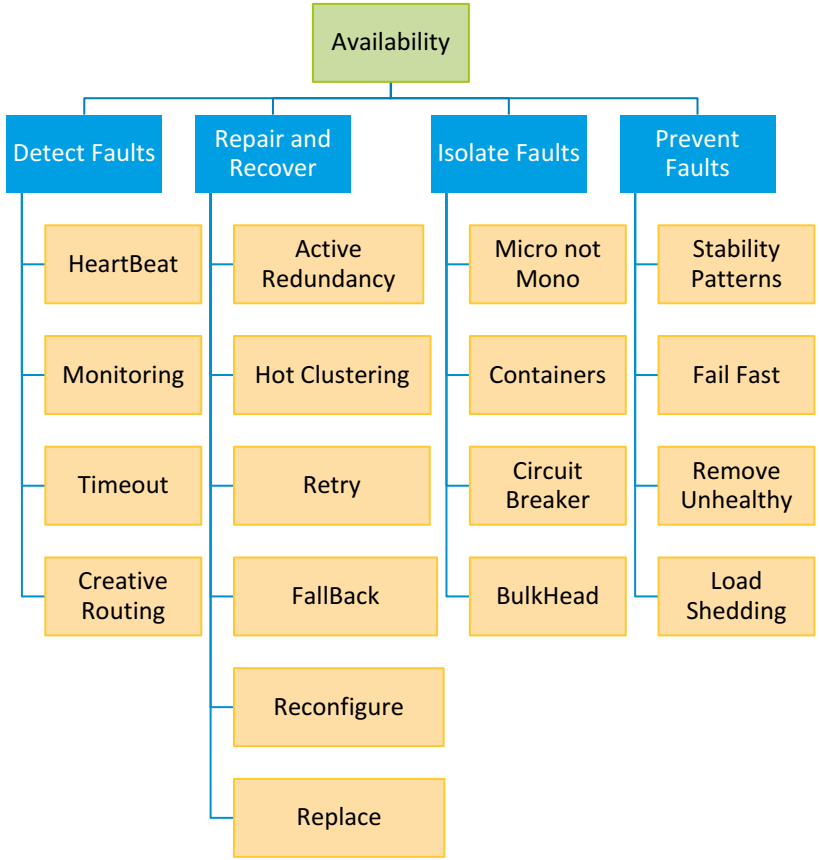
© 2015 Kim Horn

Scalability Tactics



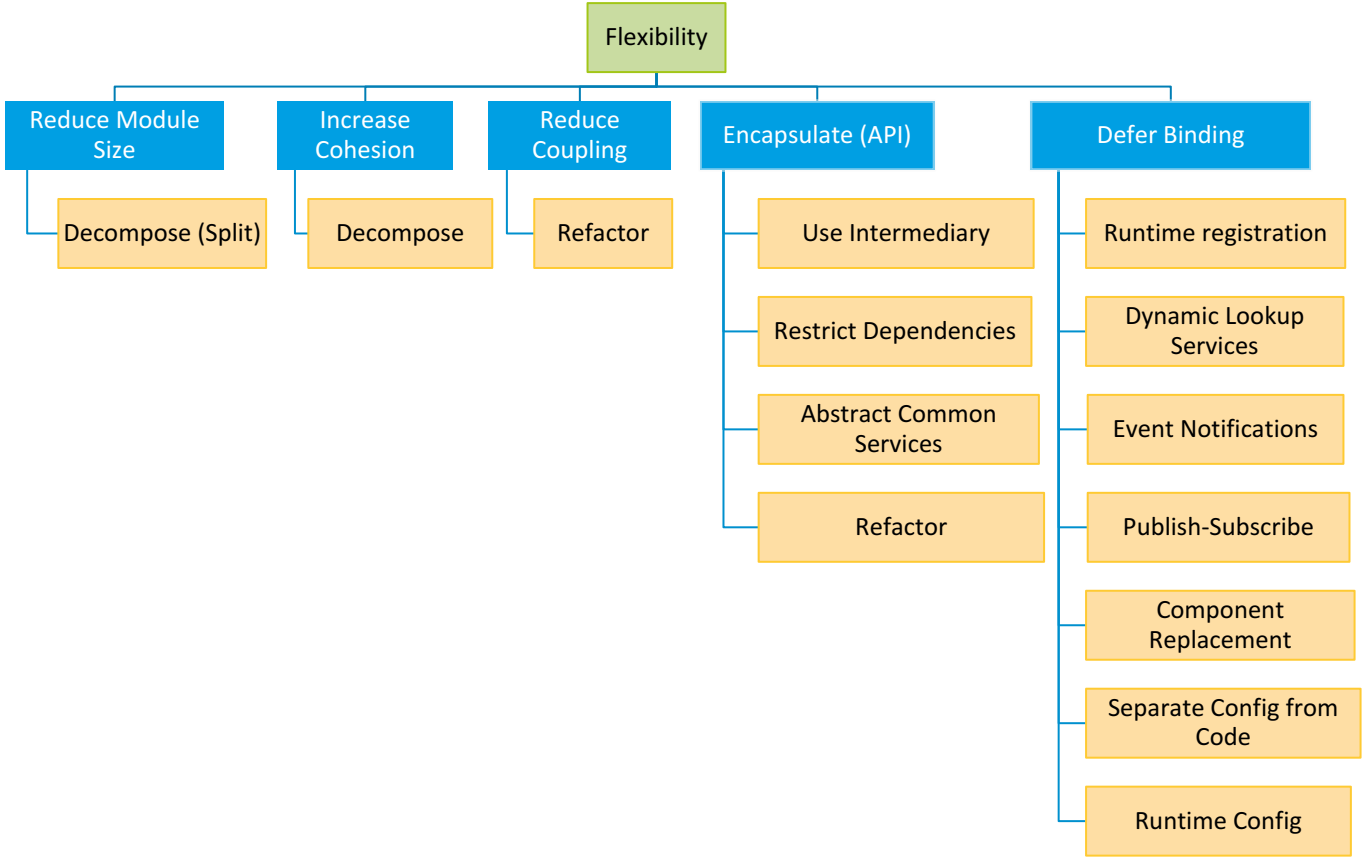
© 2015 Kim Horn

Availability Tactics



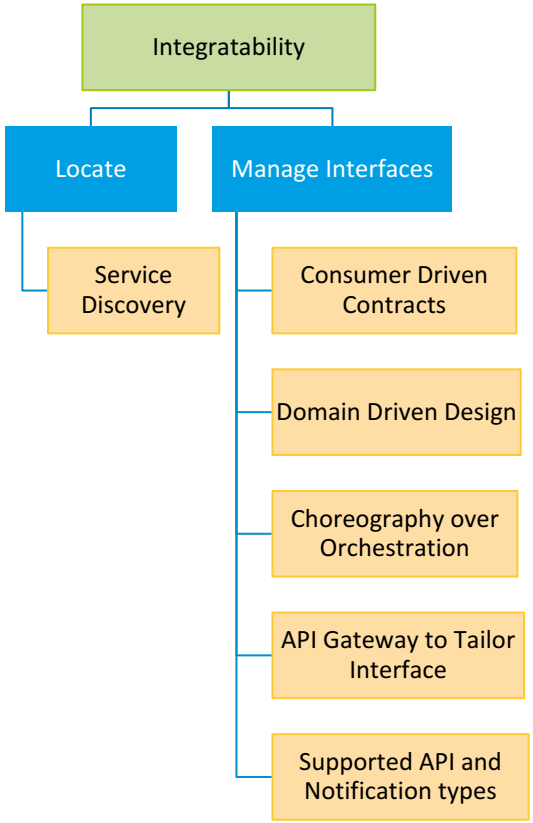
© 2015 Kim Horn

Flexibility Tactics



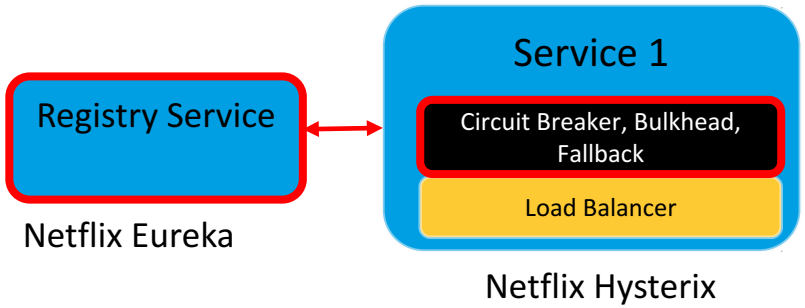
© 2015 Kim Horn

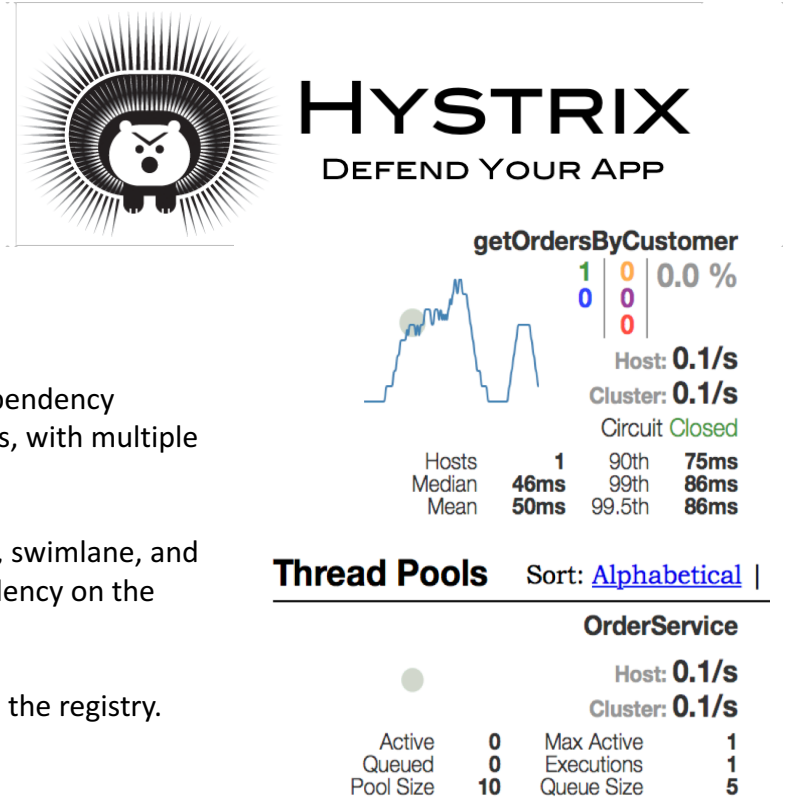
Integratability Tactics



© 2015 Kim Horn

Health: Reliability, Scalability, Availability

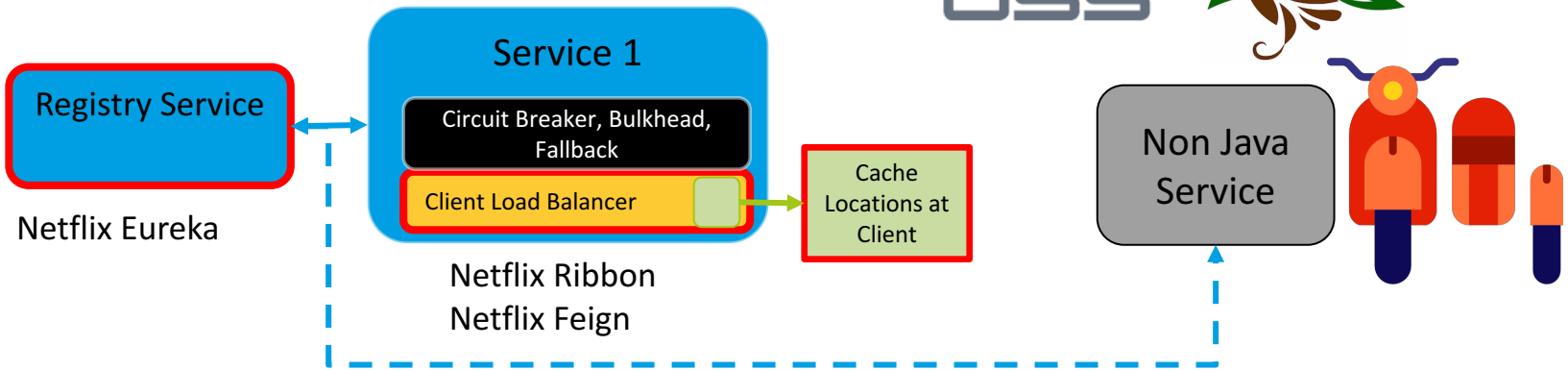




- Hystrix is essential, it protects applications from cascading dependency failures, an issue common to complex distributed architectures, with multiple dependency chains.
- Multiple isolation techniques are employed, such as bulkhead, swimlane, and circuit breaker patterns, to limit the impact of any one dependency on the entire system.
- Provides health monitoring across a cluster of services back to the registry.

© 2015 Kim Horn

Service Discovery is Critical



Finding services dynamically is critical for reliability, availability and scalability:

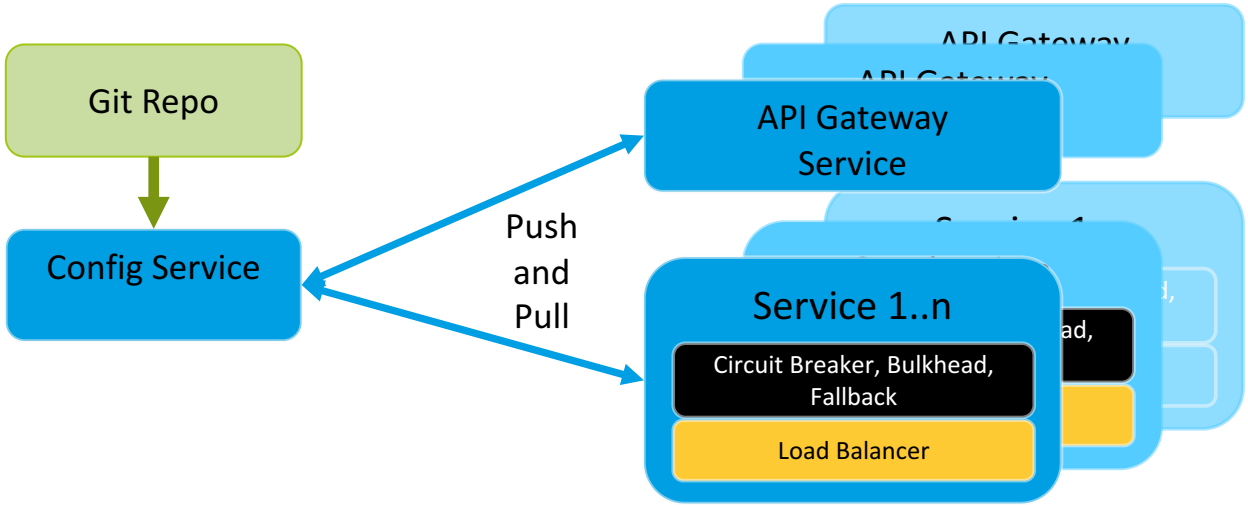
- The Eureka Registry stores service meta data to provide dynamic discovery and availability of healthy services to clients, Non Java supported by Prana SideCars;
- Feign and Ribbon allow clients to find services, and communicate with them reliably, via REST and provide full HATTEOS;
- Caching service locations and load balancing within the client allows direct communications, unimpeded by intermediating infrastructure such as load balancers;
- Re-usable Netflix OSS (Open Source Software) components, provides conceptual integrity to the whole framework, from the Gateway to the backend.

© 2015 Kim Horn

Configuration– Know What's Deployed

Provide Configuration Best Practice:

- Strict separation of Configuration from Code in all environments.
- One Immutable Service code image (containerised) for all environments;
- Store configuration in environment variables;
- Services pull config from central server, and new config can be pushed;



Across all environments:

- Dev
- Test
- SIT
- Pre-prod
- Production

Containers, e.g. Docker, are further used to support dev ops and provide isolation.

© 2015 Kim Horn

Service Discovery in the Cloud.

- **Highly Available**. Support "hot" clustering, where service lookups can be shared across multiple nodes in a service discovery cluster. If a node becomes unavailable, other nodes in the cluster should be able to take over.
- **Peer-to-Peer**. Each node in the service discovery cluster shares the state of a service instance.
- **Load Balanced**. Needs to be able to dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it.
  - Service discovery replaces the more static, manually managed load balancers
- **Resilient**. The service discovery's client should be able to "cache" service information locally. Allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the local cache.
- **Fault Tolerant**. Needs to detect when a service instance is not healthy and remove the instance from the list of available services automatically.

© 2015 Kim Horn


NETFLIX ZUUL
API Gateway
github.com/Netflix

ZUUL - API Gateway Goals

Quickly and nimbly apply functionality to edge service, including:

Authentication and Security - identifying authentication requirements for each resource and rejecting requests that do not satisfy them.

Insights and Monitoring - tracking meaningful data and statistics at the edge in order to give us an accurate view of production.

Dynamic Routing - dynamically routing requests to different backend clusters as needed.

Stress Testing - gradually increasing the traffic to a cluster in order to gauge performance.

Load Shedding - allocating capacity for each type of request and dropping requests that go over the limit.

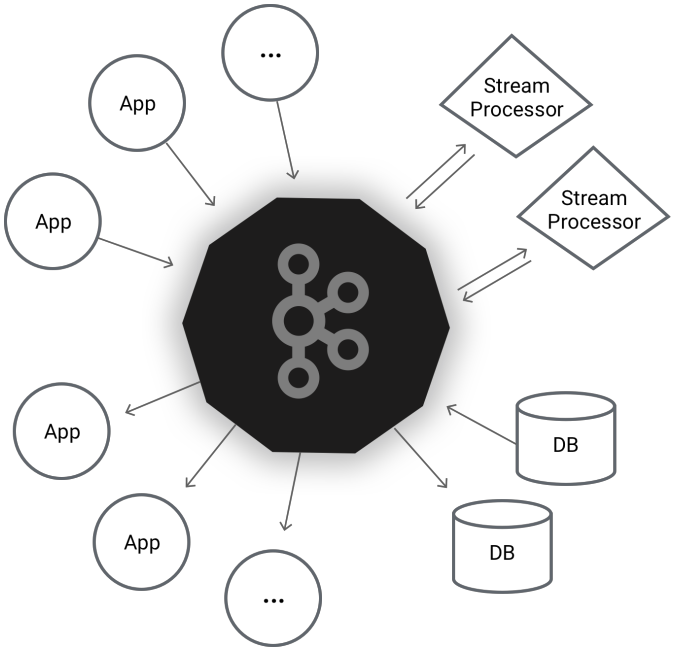
Static Response handling - building some responses directly at the edge instead of forwarding them to an internal cluster

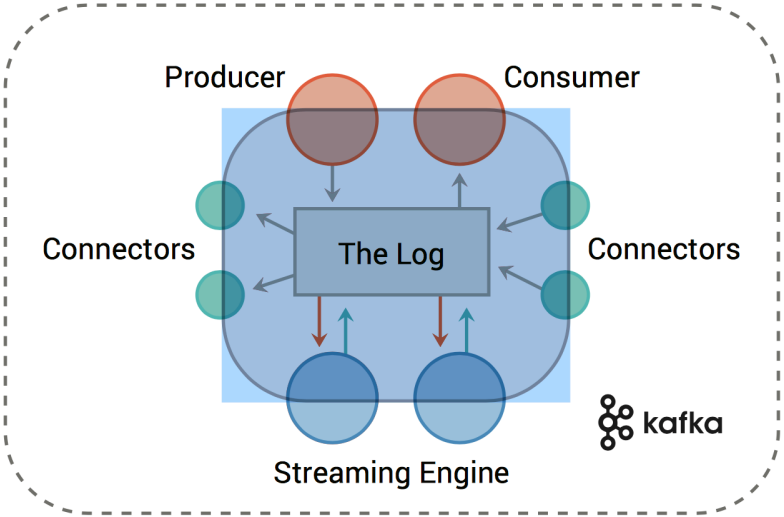
Multiregion Resiliency - routing requests across AWS regions in order to diversify our ELB usage and move our edge closer to our members

© 2015 Kim Horn

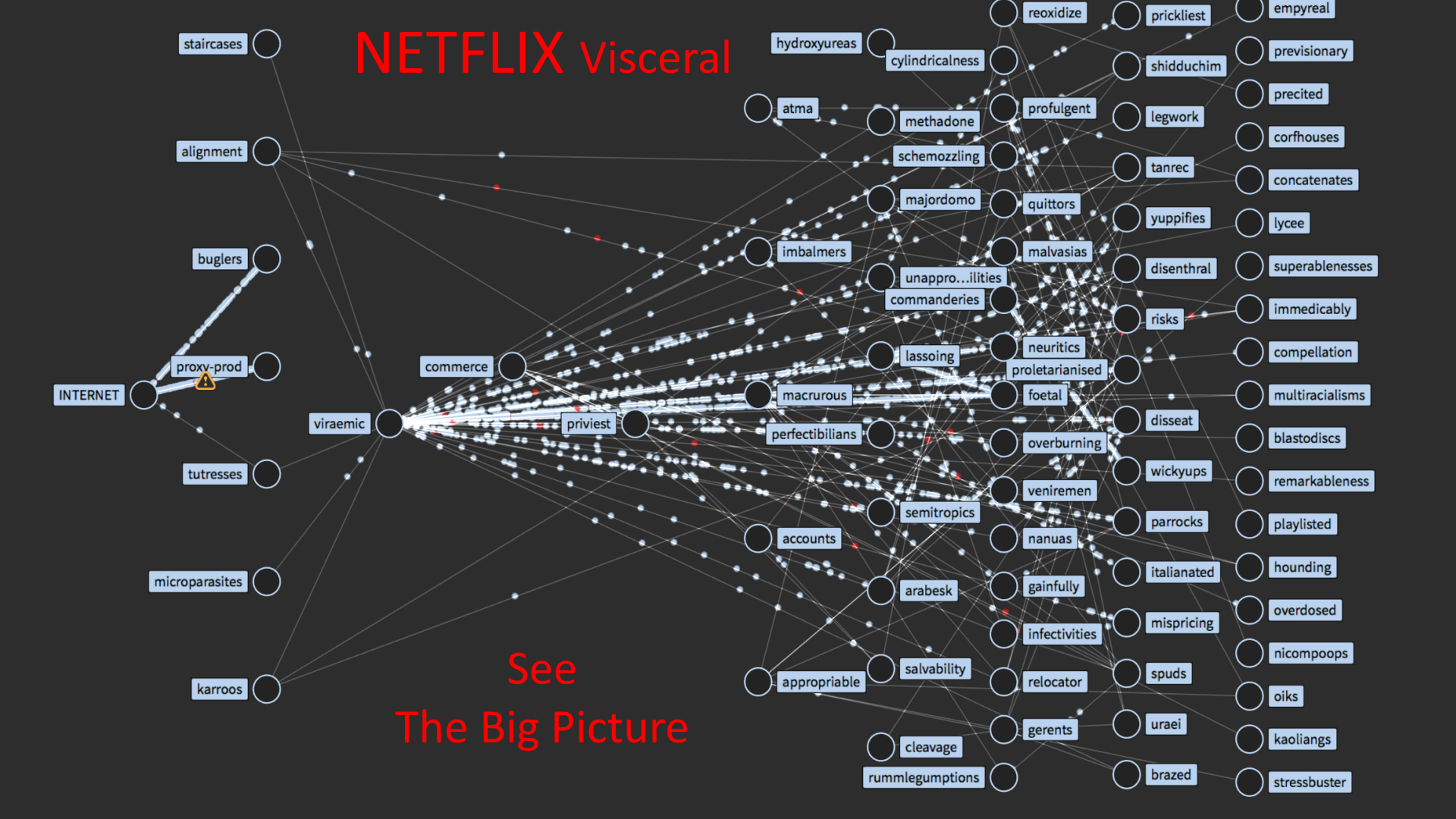
Kafka Platform – Async Events are Critical

Apache Kafka™ is *a distributed streaming platform*. Provides decoupling, reliability and scalability.





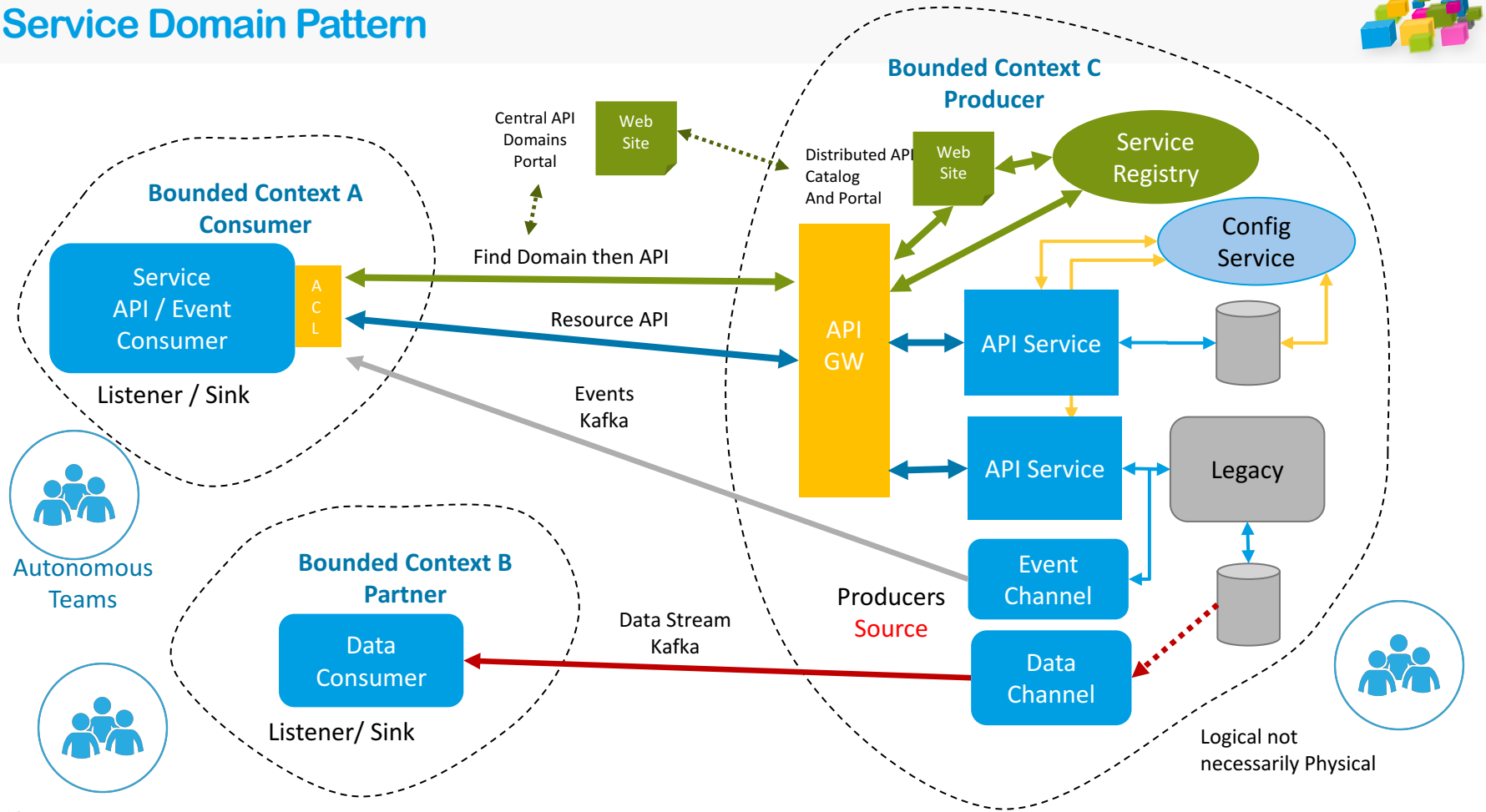
© 2015 Kim Horn


NETFLIX Visceral
staircases
alignment
buglers
proxy-prod
INTERNET
tutresses
microparasites
karroos
viraemic
commerce
priviest
hydroxyureas
cylindricalness
atma
methadone
schemozzling
majordomo
imbalmers
unappro...ilities
commanderies
lassoing
macrurous
perfectibilians
semitropics
accounts
arabesk
salvability
appropriable
cleavage
rummlegumptions
reoxidize
profulgent
quittors
malvasias
neuritics
proletarianised
foetal
overburning
veniremen
nanuas
gainfully
infectivities
relocator
gerents
prickliest
shidduchim
legwork
tanrec
yuppifies
disenthral
risks
disseat
wickyups
parrocks
italianated
mispricing
spuds
uraei
brazed
empyreal
previsionary
precited
corfhouses
concatenates
lycee
superablenesses
immedicably
compellation
multiracialisms
blastodiscs
remarkableness
playlisted
hounding
overdosed
nicompoops
oiks
kaoliangs
stressbuster
See
The Big Picture

Teams and Relationships

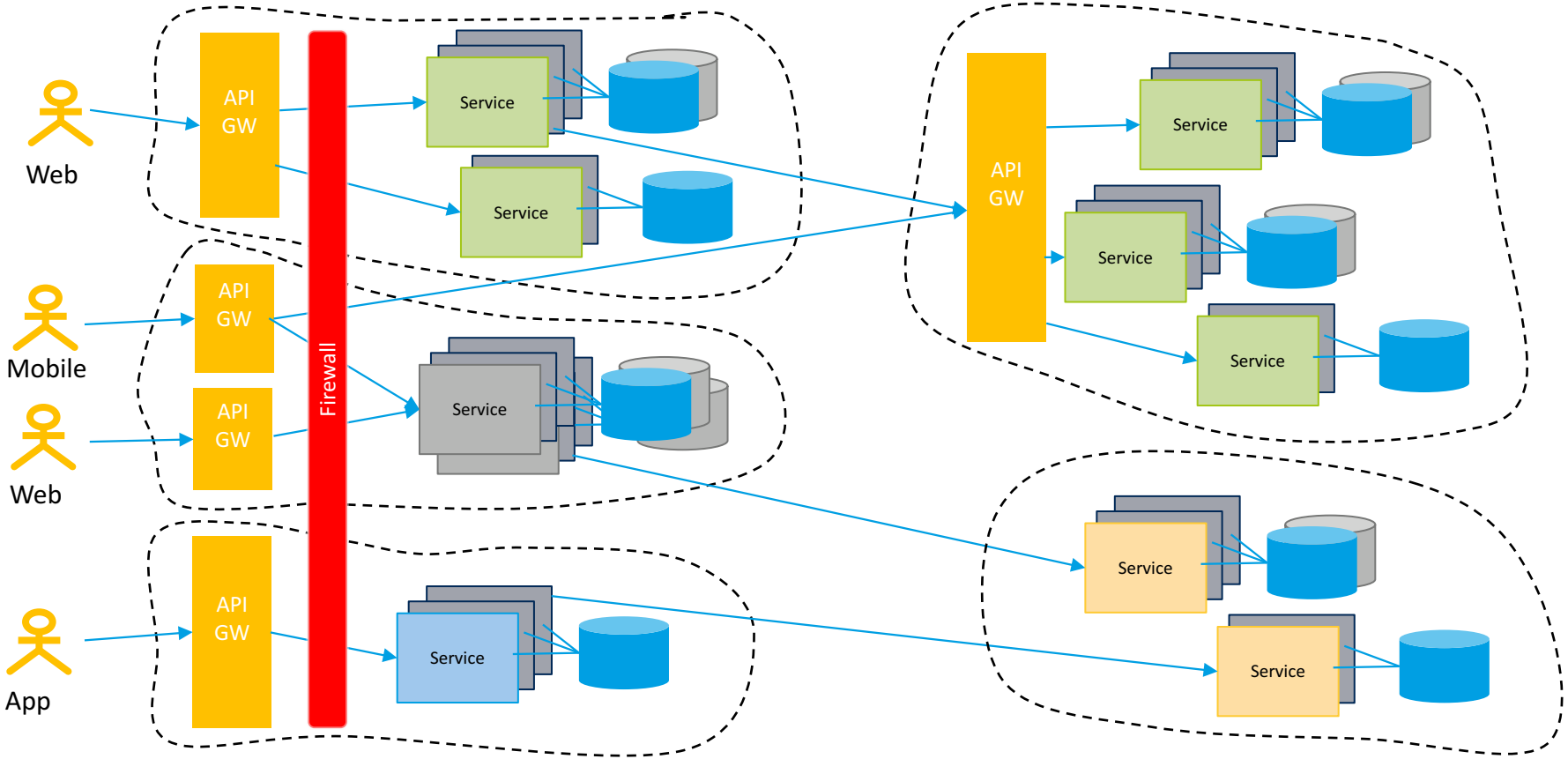
© 2015 Kim Horn

Service Domain Pattern



© 2015 Kim Horn

Enterprise Integration Options



The same components are used across the architecture

© 2015 Kim Horn

Distribute Operations - Autonomy

A single team should not become a bottleneck nor need to have expertise on every client application to create optimized endpoints.

- Each client team should now manage the deployment lifecycle of their own web service endpoints.
- Operational tools for monitoring, debugging, testing, canarying and rolling out code must be exposed to a distributed set of teams so teams can operate independently.
- They manages their contacts with their consumers.
- They manage their API catalog, the APIs exposed, the events.
- They operates and manages their systems.
- They manages their data and the data streamed externally.
- They provide a web site and portal for their customers and internal developers providing API catalog and developer guides and documentation.

© 2015 Kim Horn

Knowing Relationships is Crucial

Integration Reduces Autonomy. Microservices can involve a direct relationship between 2 parties, that can create high coupling (dependency) and reduce autonomy. Degree of autonomy is completely dependent on the type of relationship.

The relationship types indicate who has power, who controls autonomy. Domain driven design uses a tool called a 'Context Map' to understand the relationship types and dependencies between Service Teams.

For example:

- A 'Partnership' relationship indicates the 2 parties have agreed to become coupled; they agree to forgo autonomy.
- A 'Conformist' relationship indicates that one party cannot afford to be autonomous although it wants to be be; this dependency is not desired.
- An ACL (anti corruption layer) promotes autonomy fully by protecting against coupling.

Relationship Types:

- Partnership
- Customer/Supplier
- Conformist
- ACL
- Shared Kernel
- Open Host Service
- Partners
- Published Language
- Big Ball of Mud
- Separate Ways

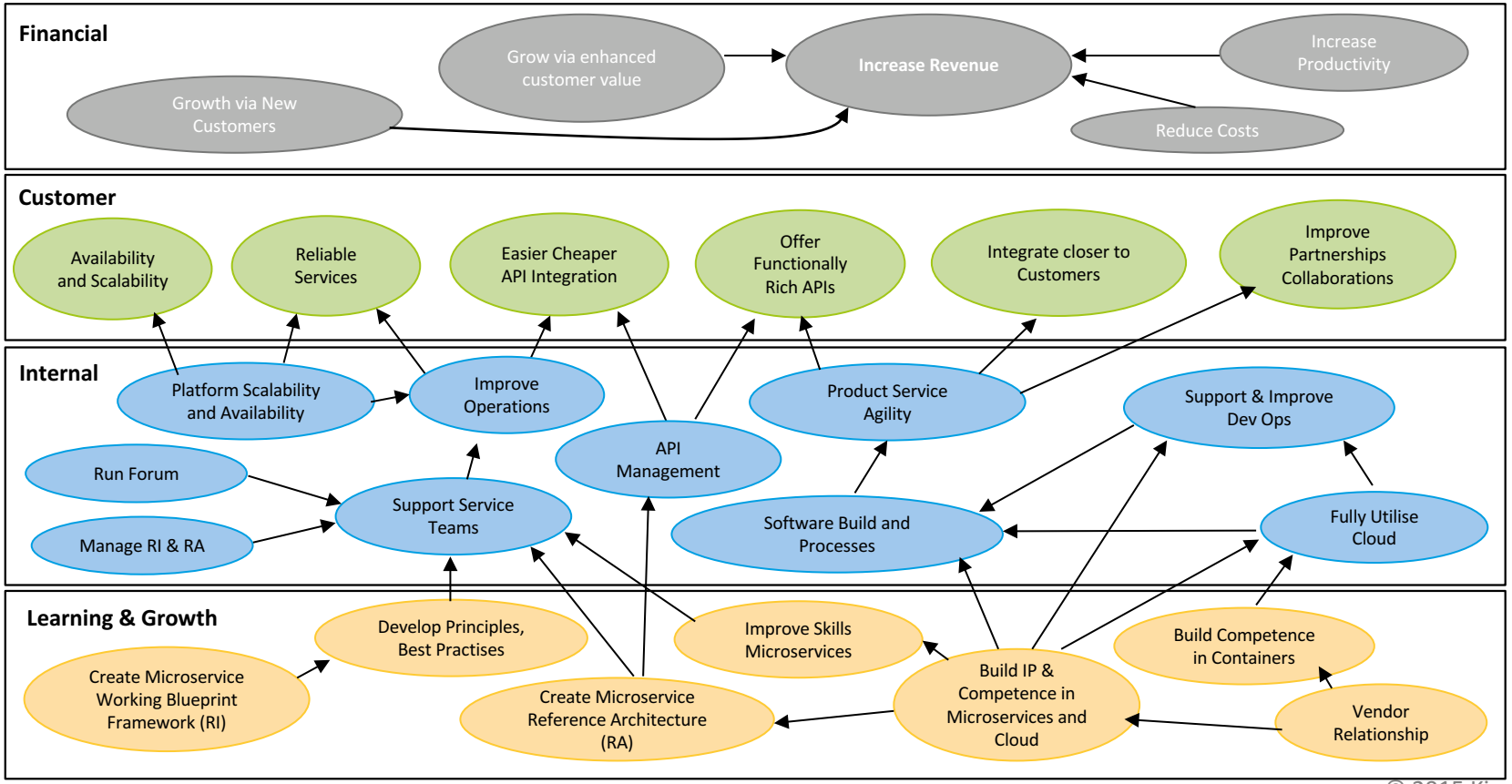
© 2015 Kim Horn

In the Future – Target State

- All applications within an enterprise are neatly bounded (by context) with standardized interfaces;
- Most importantly those interfaces are exposed and administered by the teams writing the application, not by a completely separate integration team with their own tools, which is very common today.
- These teams deal with external and internal exposure;
  - **This means federating out, distributing and decentralising the whole integration problem over to the teams.**
  - **Carefully understanding relationships to determine the impact on autonomy.**
- It will be a long haul, and evolutionary:
  - Service teams re-structured based on bounded context;
  - New applications built as microservices;
  - Some backend legacy systems modernized, possibly strangled and broken up into microservices;
  - Will be applications not changed at all, lacking any reason for it.

© 2015 Kim Horn

Microservice Example Strategy Map



© 2015 Kim Horn

The End

© 2015 Kim Horn