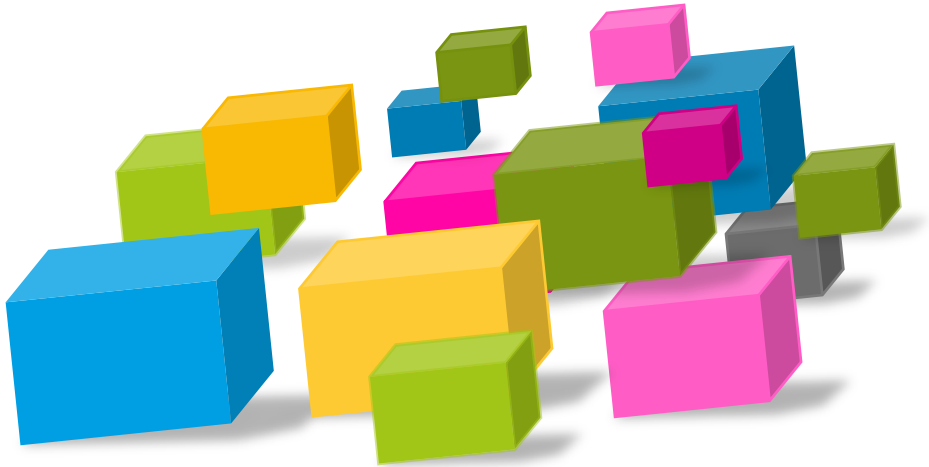
Principles and Choices Microservices

Kim Horn

Version 0.91 **10n June 2019**



Agenda

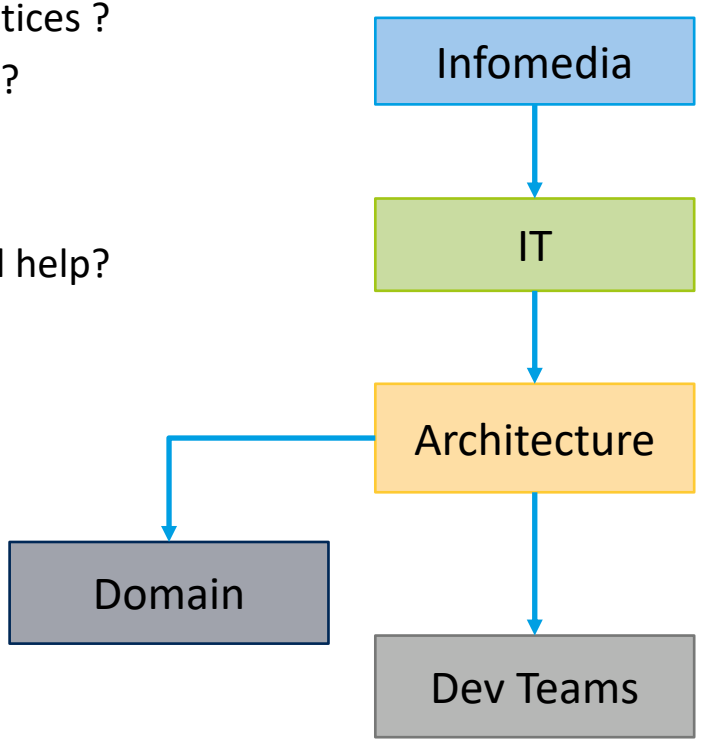
- Microservice Principles
- IT Principle Choices
- Microservice Choices
- Examples
- Appendix:
  - Tactics
  - Blueprints
  - Principles

Introduction

- How do we Govern Microservice Technology and practices ?
- How do we make choices and support other's choices ?
- How do we make teams more agile and life easier ?
- How do we make new decisions quickly ?
- Humans are really bad at making decisions so we need help?

Three Main Tools:

- **Principles (What we believe in)**
- **Practices (Way of Working)**
- **Blueprints (Reference Implementation)**



Why ?

Guide behaviour by removing obstacles, to doing the right thing, and making it easy.

Verses

- **Threats** – if you don't do it this way we can't allow project to proceed;
- **Argument** – you are not doing it the right way, this is the right way.
- **Promises** (Incentives) – reduce the number of bugs, meet the milestone, and you will get a plaque and shopping voucher;

From 2019 Talk by Danial Kahneman

Principles

"As to methods there may be a million and then some, but principles are few. The man who grasps principles can successfully select his own methods. The man who tries methods, ignoring principles, is sure to have trouble."

— Harrington Emerson

Principles Outlive Tactics (Farnam Street Principle)

In American football, thousands of plays have been invented over the years, some with more success than others. Good coaches will be able to create plays based on their teams' strengths and their opponents' weaknesses. Their plays are based on principles that they fully understand. Play-stealers, on the other hand, spots tactics that have been effective, and plug them directly into their programs.

In a game, both the coach and the play-stealer will call successful and unsuccessful plays. Here's the difference: only the coach can determine why a play was successful or unsuccessful. Only the coach will be able to figure out how to adjust it. The coach, unlike the play-stealer, understands what the play was designed to accomplish and where it went wrong, and can easily course-correct.

The play-stealer has no idea what's going on. He doesn't understand the difference between something that didn't work and something that played into the other team's strengths.

Tactics provide the "what" and the "how." And just like in football, sometimes that can be enough to get a result. But if you want results no matter how the landscape changes, you must also understand the "why." By understanding the principles that shape your reality, your "why" will more accurately guide your thoughts and actions.

Microservice Architecture Principles

Microservice Architecture Principles

Microservices are:

1. Modelled around a bounded context; the business domain, (Domain Driven Design)
2. Responsible for a single capability;
3. Individually deployable, based on a culture of automation and continuous delivery;
4. The owner of its data;
5. Consumer Driven, (Consumer Driven Contracts)
6. Not open for inspection; Encapsulates and hides all its detail;
7. Easily observed;
8. Easily built, operated, managed and replaced by a small **autonomous** team;
9. A good citizen within their ecosystem;
10. Based on Decentralisation and Isolation of failure;

10 Design Principles for Azure Applications

- **Design for self healing.** In a distributed system, failures happen. Design your application to be self healing when failures occur.
- **Make all things redundant.** Build redundancy into your application, to avoid having single points of failure.
- **Minimize coordination.** Minimize coordination between application services to achieve scalability.
- **Design to scale out.** Design your application so that it can scale horizontally, adding or removing new instances as demand requires.
- **Partition around limits.** Use partitioning to work around database, network, and compute limits.
- **Design for operations.** Design your application so that the operations team has the tools they need.
- **Use managed services.** When possible, use platform as a service (PaaS) rather than infrastructure as a service (IaaS).
- **Use the best data store for the job**. Pick the storage technology that is the best fit for your data and how it will be used.
- **Design for evolution.** All successful applications change over time. An evolutionary design is key for continuous innovation.
- **Build for the needs of business.** Every design decision must be justified by a business requirement.

https://docs.microsoft.com/en-us/azure/architecture/guide/design-principles/

IT Principle Choices

IT Principle related Microservice Choices

1. Degree of Autonomy – most important choice
2. Agility Over Reuse
3. SOA vs Microservices
4. Single Tech Stack
5. Portability Vs Cloud Vendor Lock in
6. Managed Services Over Manage Ourselves

These boxes denote principles impacted

Note that Information slides are added to provide a POV – not one that is endorsed or preferred.

1. Degree of Independence (Autonomy)

How autonomous do you want your teams and services to be?

Only constrained by:

- Corporate Ethical, Moral and legal principles
- Security and Privacy
- Customer SLAs
- Cost not considered by Architectural Fitness Function (Cost is traded off like other qualities Vs primary concern).

What are kinds of teams do we create ?

What are kinds of relationships can teams afford ?

Add on Constraints:

- Standards
- Process
- Language
- Frameworks, Blueprints, Reference Implementation
- Technology stacks / platforms
- Centralised Service Re-Use

How do these get Governed ?

- Centralised
- Decentralised

To maximise autonomy need to delegate decision making and control to teams. Teams own their services.

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

2. Agility over Re-use

Reuse of:

1) Services – layered systems (relates to item 6) – SOA Vs MS.
2) Code and libraries – reusable shared common components
3) Tech – later slide

What are kinds of relationships can teams afford ?

Sharing services and code creates Dependencies, reducing Agility. Although reusing code and libraries would seem to reduce work it can reduce agility, as teams try and align code and project deliverables.

Changes in shared libraries can have cascading implications that stops projects delivering value.

Rule: Only share code that is not exposed from your bounded context. E.g. Platform and Infrastructure.

3. Individually deployable, based on a culture of automation and continuous delivery;
8. Easily built, operated, managed and replaced by a small autonomous team;

3. SOA vs Microservices

SOAs emphasises reuse, centralisation and governance, where as Microservices pulls towards decentralisation, autonomy and agility;

SOA is an Enterprise Capability whereas Microservices are Specific Application components.

For Microservices, **Re-use** is a low Priority, but is often a confused goal;

SOA via an ESB incurs a Canonical data model (to implement re-use) that is often found to create strong dependencies between services but more so between teams; so that agility is completely lost. Teams have to delay work while ESB team has bandwidth. Cross service/team negotiations drag out work.

- ESB/SOA integration should be avoided;
- Light weight Async messaging is alternative.

8. Easily built, operated, managed and replaced by a small autonomous team;

Dave Chappel 2006 on SOA Reuse.

David Chappell observses that the "reuse" concept didn't work too well with object-oriented programming, and isn't working too well with SOA, either.

"I've spent much of the last two years flying around the world talking with people about SOA. What I've heard from virtually all of them is that reuse of business logic is nearly as tough with services as it is with objects. Even Gartner is now saying that you should expect to reuse only a fraction of your services, maybe just 20% of them. Yet if service reuse is so limited, how much value will SOA really provide? If an organization reuses only one in five of its services, why is it building the other four? The one that does get reused had better provide significant value to make up for the cost of the four that sit idle."

Chappell puts it this way: ***"Creating services that can be reused requires predicting the future… how can a service's creators accurately guess what future applications will need?" The 'If-you-build-it-they-will-come' approach is tough to turn into real reuse,"*** he says. Plus, there are few, if any, organizational incentives to build a component that can be reused by other groups.

For reuse to work, Chappell says, companies need a corporate culture conducive to the sharing of innovation across departments, with *"top management support, strong commitment, and diligent effort."* How many companies are fortunate enough to have such visionary management and culture?

4. Single Tech Stack – Reuse of Platform

It is difficult enough to decide on the technology stack for a monolithic application, but now this decision has to be made for every microservice. Your technology stack should include programming language, testing and logging framework, cloud provider, infrastructure, storage, monitoring, etc.

- What seems attractive in theory can become problematic in practice if your services are too heterogeneous. Standardization becomes a problem and there is potential for cowboy behavior. Also, it's harder for people to move between teams if every team is using a completely different stack. **See Bounded Buy**

- A balanced approach where there is a preferred technology stack. If any team wants to override this default stack, they should justify their decision with pros and cons of why a different stack is more suitable for their microservice (Implying a Governance process).

3. Individually deployable, based on a culture of automation and continuous delivery;
8. Easily built, operated, managed and replaced by a small autonomous team;

5. Portability

How much vendor lock in can we tolerate. Do we require platform portability across:

1) Cloud vendors: AWS, Google, Microsoft (one or many)
2) Cloud and in house data center
3) Cloud IaaS and Vendor PaaS, e.g. OpenShift
4) Cloud production BUT build, develop and run on desktop, debug Cloud + desktop

Relates to 13.
Development Env Where ?

if you answer yes to any of the above a portable platform is required.

Vendor Lock In

Vendor lock in is not a bad strategic choice, as long as you only need to build and deploy in the one vendor. No one can predict what the IT market will be like in 5 years, so lock in is as good a bet as no lock in.

If use cloud vendor solely (cloud lock in) do we fully use their service over Open source or other vendor (service lock in)?

8. Easily built, operated, managed and replaced by a small autonomous team;

Bounded Buy - Thoughtworks

"Out-of-the-box or SaaS solutions tend to aggressively expand their scope to entangle themselves into every part of your business. ***We recommend a strategy to only select vendor products that are modular and decoupled and can be contained within the Bounded Context of a single business capability****."*

- Most organizations that don't have the resources to custom-build their software will select out-of-the-box or SaaS solutions to meet their requirements.
- All too often, however, these solutions tend to aggressively expand their scope to entangle themselves into every part of your business. This blurs integration boundaries and makes change less predictable and slow.

Sticky Clouds – Thoughtworks

“Cloud providers know they’re competing in a tight market, and to succeed, they need to sign up and retain long-term customers. Thus, to stay competitive, they race to add new features,

…

However, once customers sign up, these providers tend to create as sticky a connection as possible with their customers to discourage roaming to another provider. Often this manifests in a strong dependency on their specific suite of services and tools, offering a better developer experience as long as customers stay with them.

Some companies are taken by surprise when the stickiness becomes apparent, often at the time of making a choice to move parts or all of their workloads to another cloud or finding their cloud usage and payments spiraling out of control.

We encourage our clients to use either the run cost as architecture fitness function technique to monitor the cost of operation as an indicator of stickiness or Kubernetes and containers to increase workload portability and reduce the cost of change to another cloud through infrastructure as code.
“

6. Managed Services

Do we build our service, then manage and operate them ourselves

or

prefer outsourced managed services ?

We use other peoples tooling / components only, don't maintain our own.

- It's not our core capability.

IaaS or PaaS – how much do we build and manage ourselves or hand over to vendor ?

Microservice Choices

Items

1. Server Side Vs Client Side Load Balancing
2. API Management Gateway or ELB/ALB
3. Synchronous and Asynchronous (Event) Messaging
4. Platform Chassis
5. Layered or not - reuse
6. Orchestration, Choreography or none
7. Data Base per service
8. Acid Vs Base – CAP What ?
9. Polyglot – Multiple Languages, Platforms, technologies or Just focus on one
10. Lambda and Containers
11. Mono Vs Multi Repos
12. Dependency Mngt
13. Development Env Where ?
14. Horizontal Vs vertical Scaling
15. Versioning / CDC
16. Others…

1. Server Side Vs Client Side Load Balancing / Discovery

Do we use Infrastructure Server Side Load (SSL) balancers or Client Side Load (CLB) balancers ?
Blueprint 1 or Blueprint 2 – see next 2 slides.

- CLB requires a Registry, Registry creates a dependency with services.
- However, its is portable as it decouples app from infrasructure = Software Controlled Infrastructre.
- Allows creative app specific load balancing

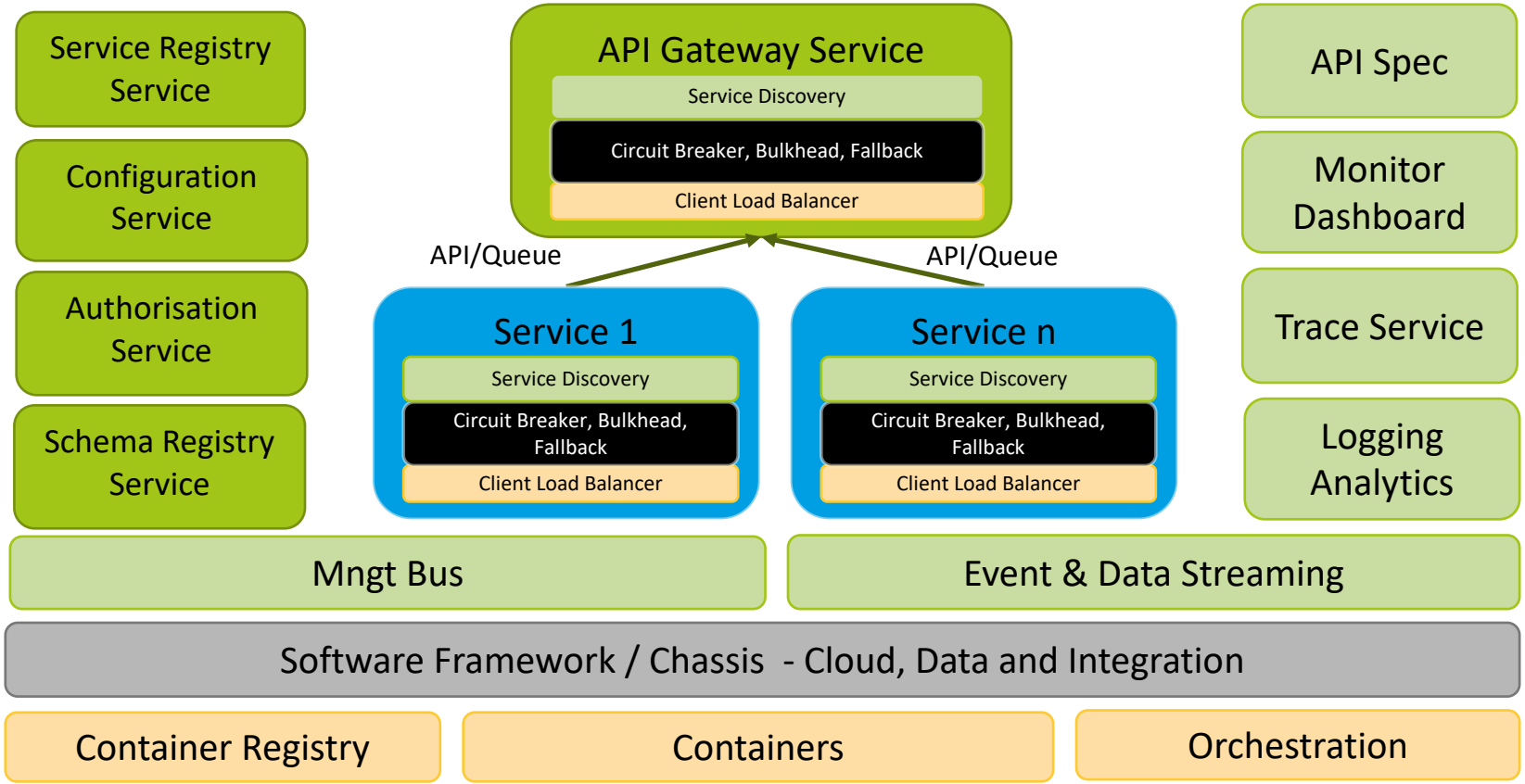
- SSL abstracts Load balancing away BUT is another item that requires configuration, operation and management.

Discovery: The set of service instances in a micro services world changes dynamically due to scaling and upgrades.
Services have dynamic network locations, so you need a way for new service instances to be discovered.
How to locate a service: DNS (R53) , Named Service (Spring Cloud)

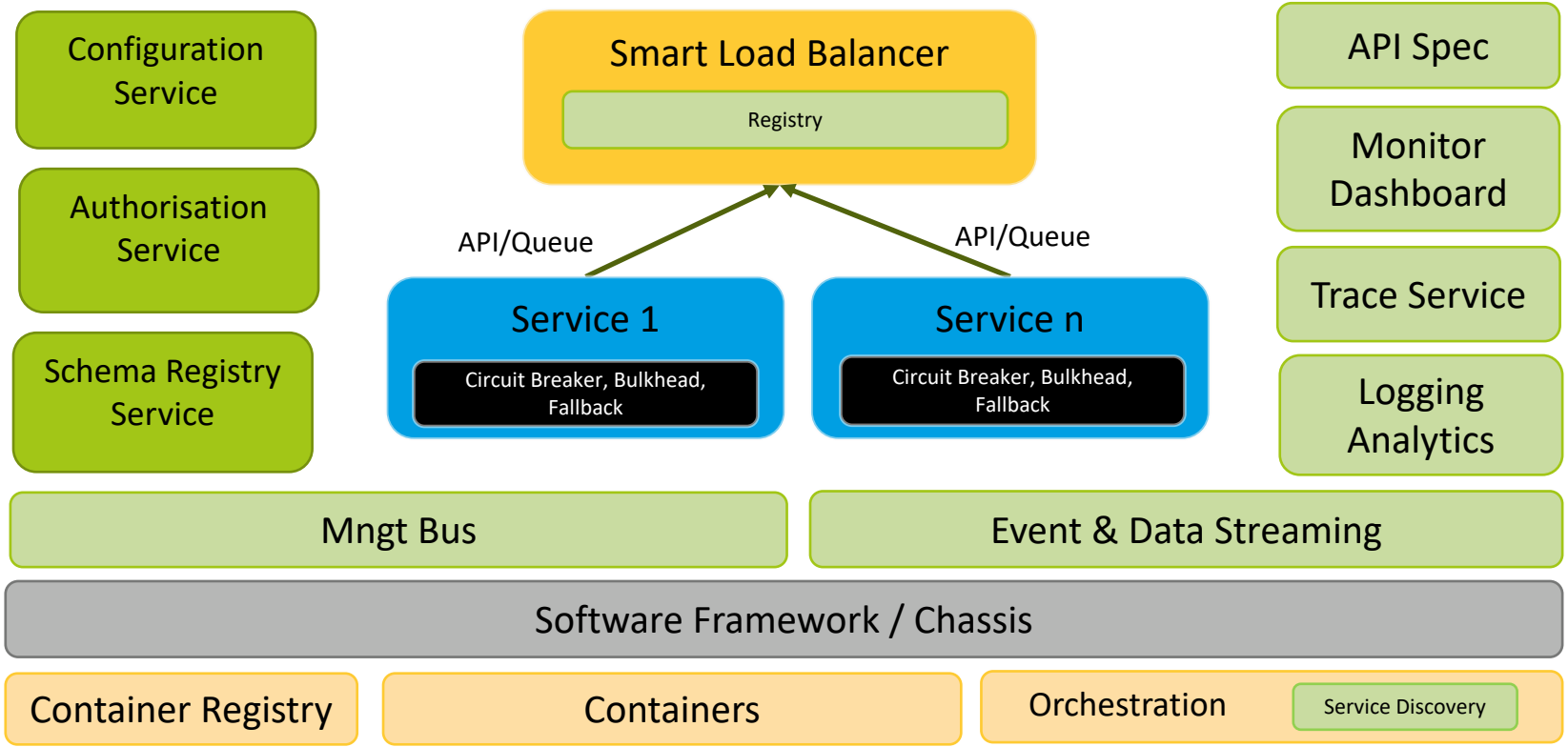
- Portable Registry Component (Eureka, Consul Zookeeper…)
- ELB per Service – we all know this is bad
- ALBs – latency ?
- R53 with ECS Discovery – lower latency, CName per Service

Relates to 12.
Dependency mngt

Distributed Cloud Blueprint 1 – Client Side Load Balancing + Portable



Blueprint 2 - Server Side Load Balancing – AWS:ALB-ECS, Kubernetes



2. API Management/Gateway or ELB/ALB

How do we manage the boundary on our bounded context, and route from URLs (Paths) to IP:Port ?
How do we handle cross cutting concerns like, Authorisation, Logging, Tracing, Throttling ?
Blueprint 1 or Blueprint 2 – see past2 slides.

Relates to 12. Dependency mngt

- Use an API Gateway ?
- Do we need API Management ?
- Can we just use an ALB ?

Can we handle HTTP, REST, SOAP, Websockets, SSE, Events….

How fat do we make the API Gateway ?

- Recommendation is to put no business or domain logic at all

3. Synchronous and Asynchronous (Events)

HTTP calls create strong dependencies between services, and create massive overhead attempting to make them fault-tolerant……..

"Using synchronous communications like REST, "from a technical perspective, we've just built a monolith. It doesn't matter that all these things are separate services, the system is still behaving like a monolith,"

James Roper (lightBend)

- Alternative like 'Play' only have Asynchronous messaging, using an event model is the way to go for persistent connections - through Comet, long-polling and WebSockets.
- Use Kafka (Asynchronous Stream) for Event communication / AWS SQS, SNS / Firebase / PubNub…..
- Event Sourcing and Immutable Data.
- Most cases need a combination of Sync/Async.
- Async should be used for mngt data so that services do not block waiting to write to an operational service.

Asynchronous Events to client via API Gateway

A bounded context will not only manage requests with responses, and react to received messages and events but also product messages and events for consumers asynchronously. Using an API Gateway to proxy all of these provides a consistent approach.

Having an API Gateway to Proxy an application services to be complete, also needs to consider proxying the push of asynchronous messages or events. Client to Server messaging can be handled by an API , or XMLHttpRequest, **but the converse is not.**

Edge proxy also supports:

- Server-Sent-Events (SSE) is an HTTP 5 standard that allows a web application to handle a unidirectional event stream and receive updates whenever server emits data. Websockets is also HTML 5 standard that connections can both send data to the browser and receive data from the browser. SSEs are sent over traditional HTTP. That means they do not require a special protocol or server implementation to get working.
- WebSockets on the other hand, require full-duplex connections and new Web Socket servers to handle the protocol. In addition, Server-Sent Events have a variety of features that WebSockets lack by design such as automatic reconnection, event IDs, and the ability to send arbitrary events.

4. Using a Chassis Framework Vs IaaS or Roll your own

When you start the development of an application you often spend a significant amount of time putting in place the mechanisms to handle cross-cutting concerns. Examples of cross-cutting concern include:

- **Externalized configuration** - includes credentials, and network locations of external services such as databases and message brokers
- **Logging** - configuring of a logging framework such as log4j or logback
- **Health checks** - a url that a monitoring service can "ping" to determine the health of the application
- **Metrics** - measurements that provide insight into what the application is doing and how it is performing
- **Distributed tracing** - instrument services with code that assigns each external request an unique identifier that is passed between services.
- **Integration mechanisms** e.g. to call REST, DBs, Message brokers etc. Includes Circuit Breakers etc.

Forces

- Creating a new microservice should be fast and easy; quick to start
- When creating a microservice you must handle cross-cutting concerns. There are also cross-cutting concerns that are specific to the technologies that the microservices uses.

Example: Spring Boot + Spring Cloud

Open Source Application Frameworks

Using proprietary Application Frameworks vs Open Source is another kind of lock in. That is we can be locked into a cloud vendor and then locked in again with a proprietary frameworks in the cloud.

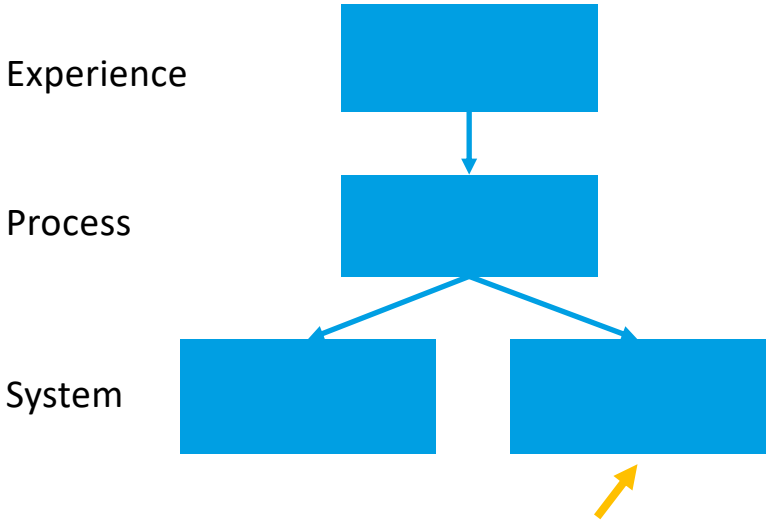
Examples are Open Shift or AWS Lambda.

There are many benefits to open source, and these need to be carefully considered against vendor provided software.

5. Layered or Not (Team Dependencies)

Good ?

Bad – Reduced Agility ?



For example, a bottom level service functioning as a data access layer (ORM) to expose tables as services; seems highly reusable. However this creates an artificial physical layer managed by a horizontal team, which can cause delivery dependencies.

8. Easily built, operated, managed and replaced by a small autonomous team;

Relates to 12. Dependency mngt ?

Layered or Not - Thoughtworks

“We've observed a number of organizations who've adopted a **layered microservices architecture**, which in some ways is a contradiction in terms. These organizations have fallen back to arranging services primarily according to a technical role, for example, experience APIs, process APIs or system APIs.

It's too easy for technology teams to be assigned by layer, so delivering any valuable business change requires slow and expensive coordination between multiple teams. We caution against the effects of this layering and recommend arranging services and teams primarily according to business capability.”

6. Orchestration Vs Choreography Vs Nothing.

Should a service know who their collaborators are, or should they simply report "what happened" and trust the appropriate service(s) to respond appropriately?

- Sam Newman calls this orchestration vs choreography.

Orchestration (BPM/Workflow/Lambda StepFunctions) creates dependencies, and also creates layered Microservices.

- Should you use an orchestration ‘product’, centralised server or create special services (backend for Front ends)

Vs

Choreography using Asynchronous (events) messaging result in a highly decoupled architecture, however not applicable in all cases.

Neither:

- Build Independent Vertical solutions.
- Let the client manage the flow, state.

8. Easily built, operated, managed and replaced by a small autonomous team;

7. Database per Service

Each service is completely independent with its own Database . This is consistent with an extreme microservices architecture where services share nothing and are completely decoupled.

- Can we create a separate DB for each service or are they dependent e.g. the Mongo Monolith ?
- Do we 'compensate' failed transactions ?
- Are there cases where we must have transactions across services ?
  - Do we share DBs for these services ?

Degree:

- Share DB, data and schema;
- Share infrastructure but separate DBs, schema - no shared data
- Separate infrastructure

Companies that have large shared NoSQL DBs find that due to small Data footprint of a MS, they can just use SQL.

Immutable data stores VS Incremental (SQL & NoSQL). Fully incremental DBMS have significant problems.

4. The owner of its data

8. ACID Vs BASE – Eventual Consistency

You can only have two of CAP, which 2 for what ?

- CA / AP / CP

- What is the impact of network Partitions ?
  - Sacrifice Availability or Consistency.

- Are we OK with Eventual Consistency of our core data, for what components ?
  - Business data
  - Operational (Registries)
  - Logs
  - …..

9. Polyglot - Multiple Languages, Platforms, technologies.

Microserivces are polyglot and encapsulated, the API forms the edge of the bounded context, and what inside should not matter.

They could be built in any language a team likes. However, do you really need to support multiple languages or does the company support a default, such as Java ?

Benefits of Microservices is that Technical implementation is irrelevant (encapsulated);

- Allows flexible use of languages, frameworks, and databases;
- Innovation - Latest technologies can be adopted quickly;

6. Not open for inspection;
Encapsulates and hides all its detail;

10. Lambda and Containers.

Containers are our preferred vehicle for Microservices ?

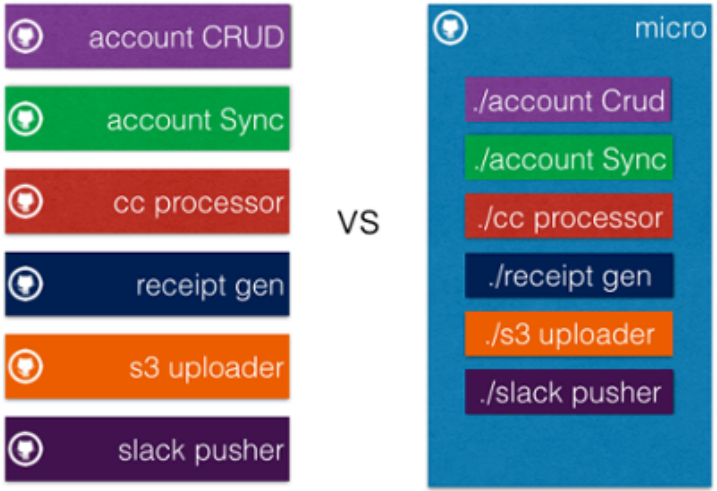
- Portable
- How to check security ?
- Consistent Image Build; OS version and JVM build., etc…. ?
- Phoenix servers - Immutable –don't allow people to fiddle.

Do we need to Support Lambda ?

- No agreement that you can build vertical apps with Lambda ?
- Extreme Lock In
- Non Portable- can't develop on desktop.

11. Mono Repo Vs Multiple

There are several ways you can organize your codebase. You can create a repository for each service, you can create a single "mono repo" for all services and have a folder for each service.



Multiple Repositories

- **Clear ownership:** Since the codebase mimics the architecture, a small team can own and independently develop and deploy the full stack of a microservice.
- **Better scale** Smaller codebases are easier to manage and lead to fewer instances of "merge hell". Teams do not need to co-ordinate with other teams leading to faster execution.
- **Narrow clones** Most source control providers including git do not support cloning parts of a repository. For large codebases, clones, pulls, and pushes take too much time, which is inefficient.

Real Example at Shippable, run a fairly big, distributed organization spread out across multiple time zones. They broke down a 5-tier platform into microservices, 50+ services, each with its own code repository. And then the chaos started.

It was challenging to enforce standardization of code across loosely-coupled repositories. Code became very complex to read and nobody understood the platform end to end. This caused code reviews for pull requests to be ineffective -- someone working on the same service lacked understanding of the the bigger picture, and anyone not working on the service had no context on it. Duplicating effort across teams since there were no shared components.

Ironically, tightly coupling teams with services and repos was causing most of our problems.

They wanted to build one team with knowledge across services as opposed to several teams with localized knowledge.

Mono Repository

Services are developed and deployed independently, but the code for all services lives in one repository.

Improvements:

- **Better developer testing:** Developers can easily run the entire platform on their machine and this helps them understand all services and how they work together. This has led our developers to find more bugs locally before even *sending* a pull request.
- **Reduced code complexity**: Senior engineers can easily enforce standardization across all services since it is easy to keep track of pull requests and changes happening across the repository.
- **Effective code reviews:** Most developers now understand the end to end platform leading to more bugs being identified and fixed at the code review stage.
- **Sharing of common components**: Developers have a view of what is happening across all services and can effectively carve out common components. Over a few weeks, we actually found that the code for each microservice became smaller, as a lot of common functionality was identified and shared across services.
- **Easy refactoring:** Any time we want to rename something, refactoring is as simple as running a grep command. Restructuring is also easier as everything is neatly in one place and easier to understand.

The Outcomes at Shipable

Productivity has increased at least 5x.

The best thing about our move to a mono repo is that we didn't give up any of the advantages of the microservices architecture.

We believe mono repos are the right choice for teams that want to ship code faster.

There are concerns that this doesn't scale well, but these are largely unfounded. Companies like Twitter, Google, Facebook run massive monolithic repos with 1000s of developers.

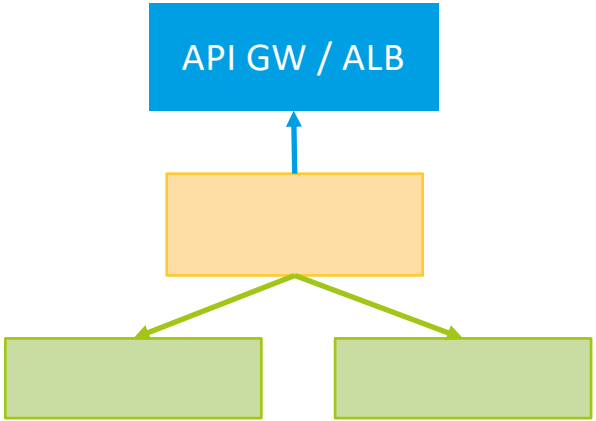
The only thing you really give up with a mono repo is the ability to shut off developers from code they don’t contribute to. There should be no reason to do this in a healthy organization with the right hiring practices. Unless you're paranoid ... or named Apple.

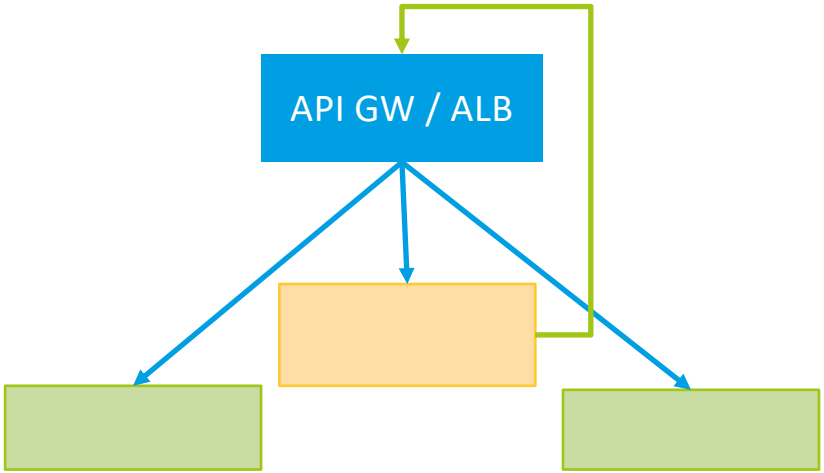
12 Dependency Mngt

Do we allow Microservices to call other microservces directly ?

- Implies client side load balancing

Do all calls go via API Gateway or ALB, (a Fat Gateway), who manages Operational/QOS concerns.





Relates to 5.
Layered or Flat

- More and more complex dependencies
- Unclear who is protecting what e.g. Circuit Breakers.
- Reduced reliability, availability (more parts to break).
- More network hops

13. Development Env Where ?

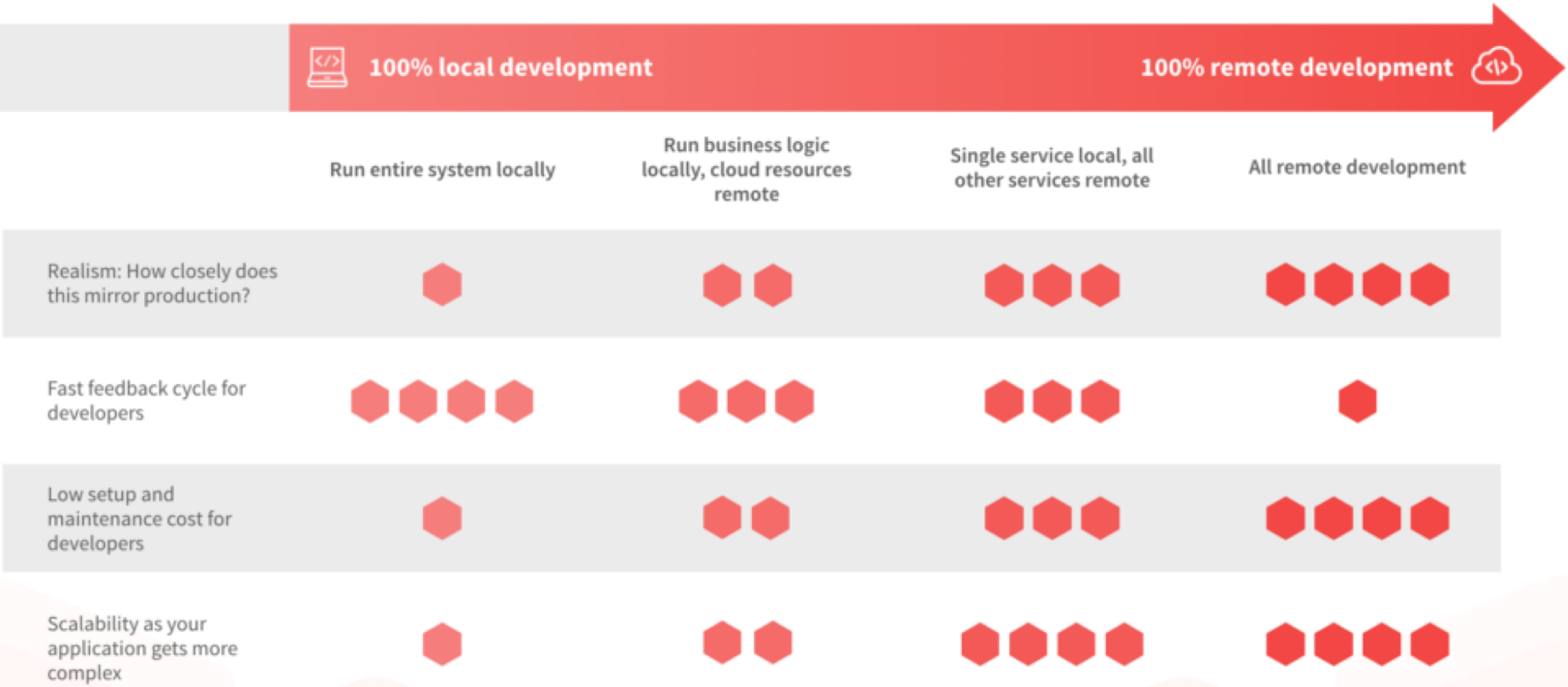
1. **Spin Up the Full System Locally,** provides fast local development;
2. **Spin Up the Full System in the Cloud,** realistic;
3. **Spin Up All Business Logic Locally, Route Cloud Services to Laptop** – things that don't run on laptop e.g. RDS, SQS, Kinesis;
4. **Make Local Code for Single Service Available in Remote Cluster.** Proxy/VPN your service into the remote cluster.

- Relates to 'Portability Vs Cloud Vendor Lock in' Choice.

Decision Aspects:

- How close does environment mirror production ?
- How fast (and isolated) is feedback cycle for developers ?
- Set up , ops and maintenance costs ? E.g. VPN (cloud to local) is difficult/costly.
- Scalability of the environment ? E.g. only run about 10 containers on laptop.

Development Env Where ?



14. Horizontal VS Vertical Scaling

Reactive programming is non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale vertically (i.e. within the JVM) rather than horizontally (i.e. through clustering).

Cloud Vendors allow VMs to scale up (buy bigger VM) – vertical.

Cloud Vendor Horizontal Scaling Issues: Providers have a set offering of instance sizes, you must choose a size large enough to provide the amount of RAM your application needs, which means you will often have CPU resources that are sitting idle. As a result, you end up buying expensive pieces of hardware and using only half (or less than half) of them.

A focused, flexible infrastructure is also a more portable cluster. During the AWS outage on February 28, 2017, the entire us-east-1 regional data center lost the ability to spawn new instances. For companies relying on horizontal instance scaling, this was crippling. Furthermore, the same outage caused cascade failures of several key AWS services.

15. Versioning / CDC

As services evolve the contracts will change – How do we balance up front analysis DDD, with allowing things to evolve.

How to we handle version changes ?

How do we understand the impact on Consumers, when we change ?

How do we see the impact of new consumers requirements on other consumers ?

How do we reduce Integration testing with changes ?

Consumer Driven Contracts help do this.

- What CDC tools do we choose ? PACT or Spring

Other concerns:

- URL includes version
- Semantic Versioning
- Backward compatibility until …
- Process to deal with breaking changes; Customer negotiation.

16. Others

1. Runtime Container Orchestration (Docker Swarm/Compose, Kubernites, Mesos, AWS ECS, etc…..)
2. Deployment CI/CD tools …..
3. Decouple Deployment and Release (Green / Blue)

Examples

Example - CircleCI Move to Microservices

1) **Multiple Languages:** we said to our engineers, “We’re a Clojure shop. It’s not an option for you to decide what language or stack you’re going to use. We all know Clojure, and it has treated us well.”
2) **DBs:** Were a MongoDB shop, but moved back to PostgresSQL, as Microservices have simple DBs, no more monolithic data bases. Now have many independent Postgres DBs.
3) **Re-Use:** we learned was that we could use shared components for infrastructure concerns.
   - The key is to find the areas where the value created by shared components outweighs the overhead.
   - Moved the decision out of the teams into Guilds, make it someone's job, see Spotify model for Guilds (https://blog.crisp.se/wp content/uploads/2012/11/SpotifyScaling.pdf)

Example– TRIFORK Spring I/O 2018

Software Platform Layer:

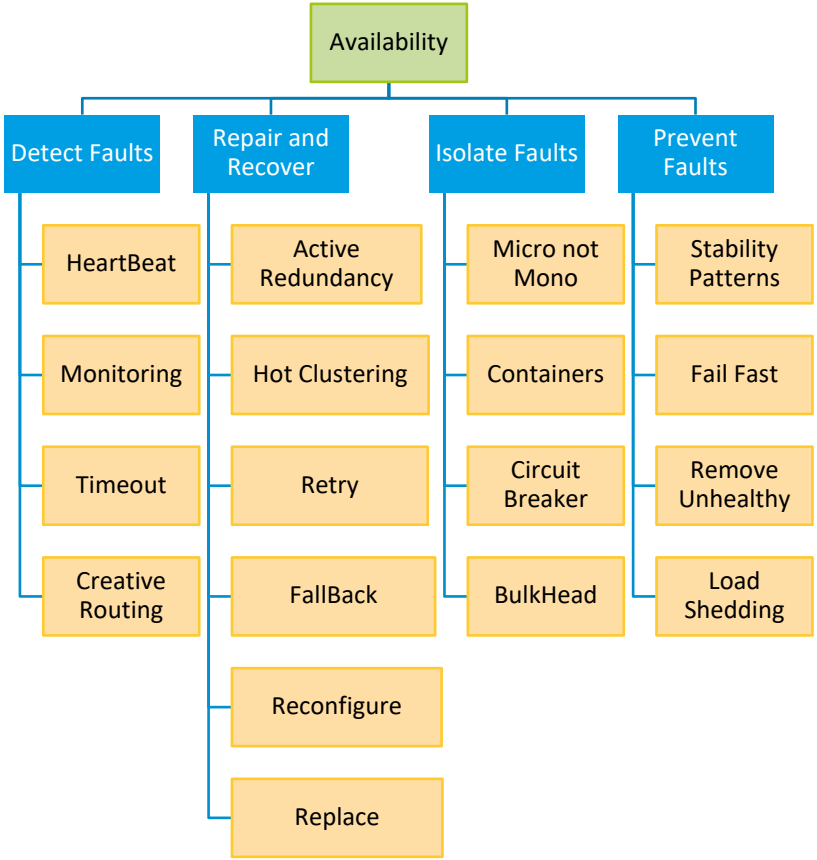
Use Spring Boot + Spring Cloud:

- Zuul used to proxy requests
- Ribbon (no CLB yet)
- Consul for Service Discovery (not Eureka, not R53 or not ECS)
- Use Server Side ALBs now, but want to move to Ribbon Client-Side
  - Internal Load Balancer adds too much latency, better to use R53 or Ribbon.

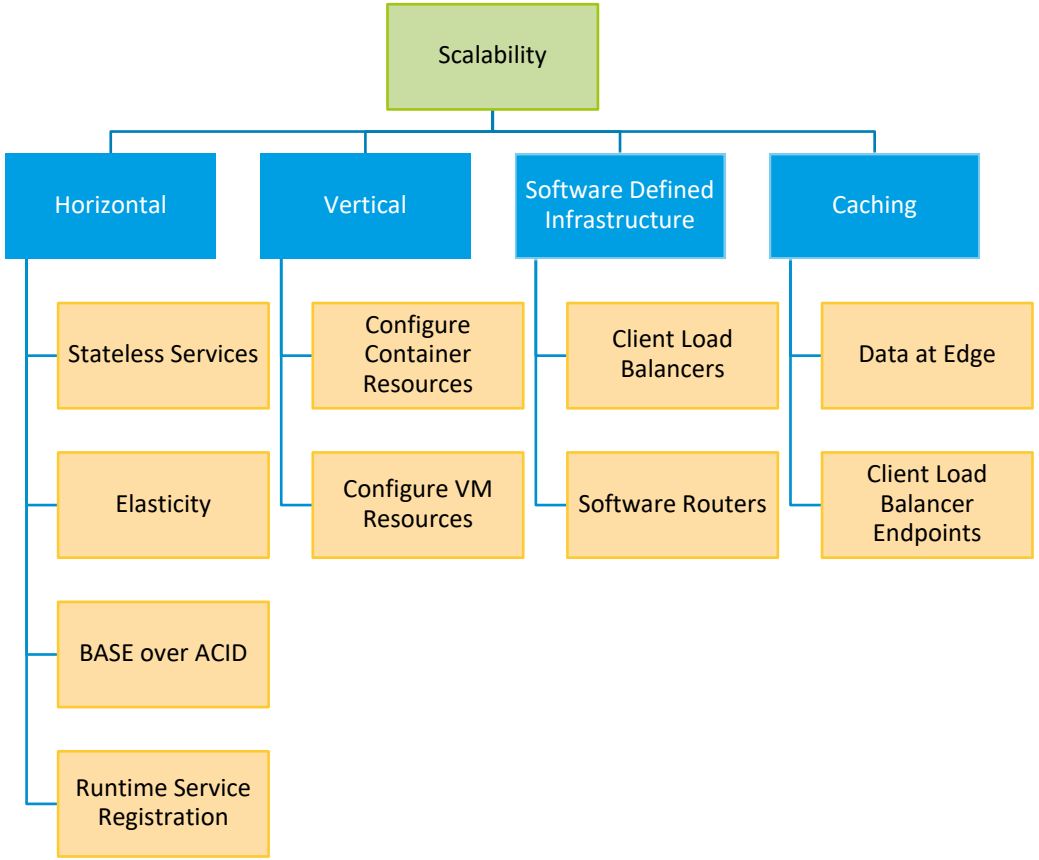
- Infrastructure Layer:
  - AWS ECS
  - Docker
  - Use Docker Health Checks, on ECS

Appendix Choices and Tactics

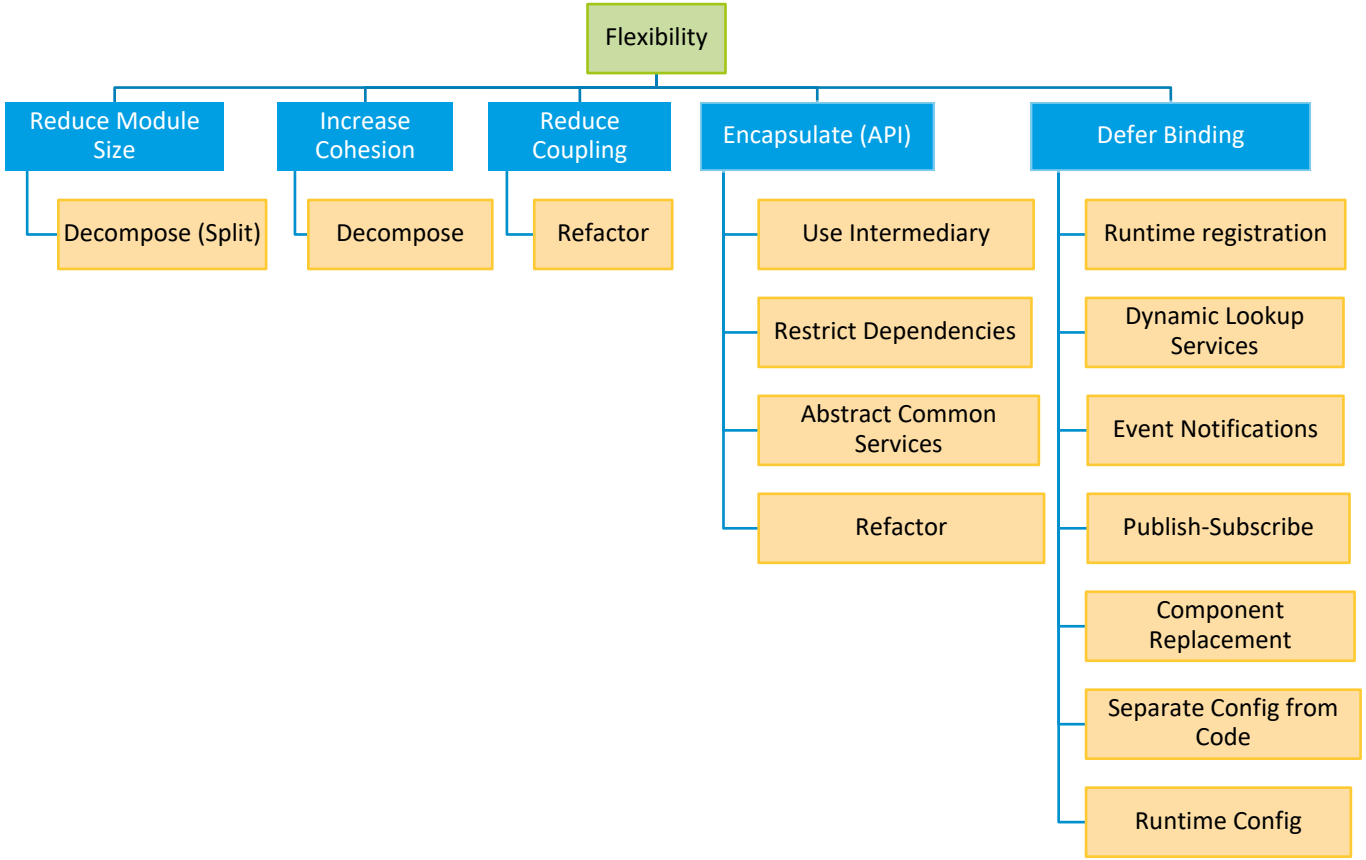
Availability Tactics



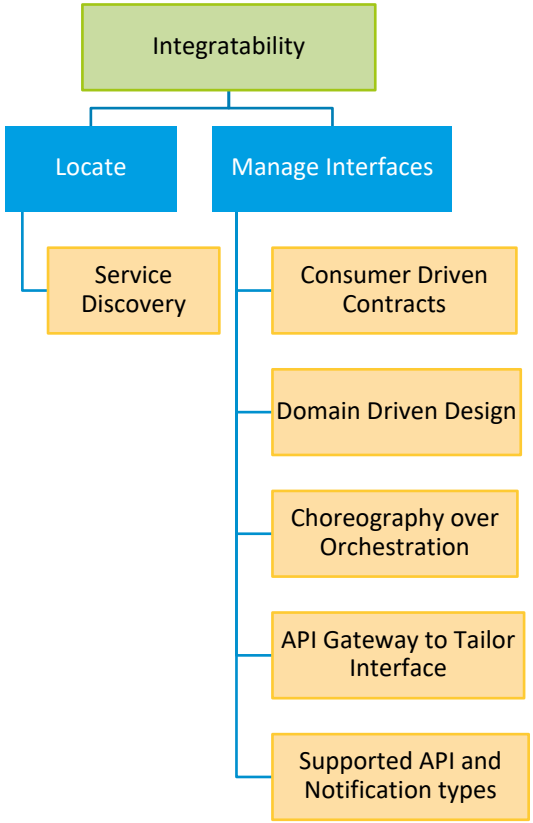
Scalability Tactics



Flexibility Tactics



Integratability Tactics



Appendix

Blueprints = Reference Implementation

Adapting EA for Agile At Scale - CEB

"As we've done more Agile, we've realized that Architecture is something that we teach, not something that we own".

- CEB

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.

Reference Implementation Benefits

Provides Solution Architecture down to code with Architectural Decisions made, tactics and patterns employed, their justifications, research done, tradeoff made explicit and QOS levels it will provide:

- 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;

Example - Goals of a Specific Companies 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,** e,g, Netflix, Facebook, Amazon, Twitter;

Example – Continued.

- Accept that both **networks and platforms are unreliable** and provide graceful mechanisms to prevent faults from cascading into complete system failure.
- 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;

Appendix Principles

WHY

Architecture Principles are intended to guide the organisation in delivering business imperatives in the most efficient and effective manner through Information Technology

- A principle is a statement of intent for use of IT
- It describes preferred practices to be followed when implementing new or upgraded systems
- It is a “soft policy” or a “fat standard”
- It is a foundation to build the enterprise architecture
- It supports:
  - Business capabilities and strategies
  - IT strategies
  - Vision of the IT environment
  - Architecture drivers

62

Definition of Principles

- The principles are defined in the following format:

ID – Name of Principle

IT Services Architecture

Statement	The Statement succinctly and unambiguously describes the principle.
Rationale	The Rationale highlights the business benefits of adhering to the principle. It describes the relationship to other principles, and the intentions regarding a balanced interpretation. It describes situations where one principle would be given precedence or carry more weight than another for making a decision.
Implications	The Implications highlight the requirements, both for the business and IT, in carrying out the principle in terms of resources, costs and activities.

63

Format & Levels

- Each Principle is supported by:
  - Title, short description or catch-phrase
  - Principle, statement of the principle
  - Rationale, the reason for being included
  - Implications, the effect it will have on future IT related decisions (people, processes, architecture, technology)

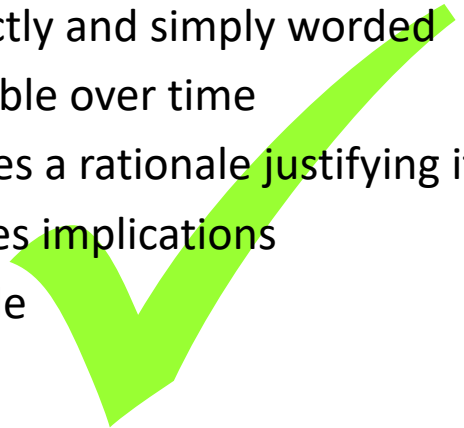
- Principles are defined at multiple levels in line with the Enterprise Architecture Framework
  - Business Principles
  - Guiding IT Principles
  - Architecture Principles
    - Business Architecture Principles
    - Application Architecture Principles
    - Information Architecture Principles
    - Integration Architecture Principles
    - Technology Architecture Principles
    - Security Architecture Principles

64

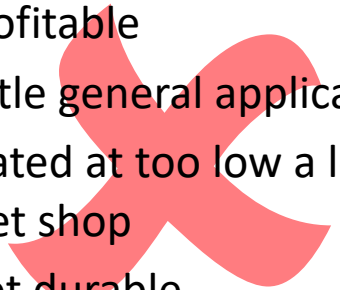
Principle clearly communicate a durable idea

A good quality Architecture Principle can clearly communicate a durable idea

- Recommends a preferred course of action
- Is directly and simply worded
- Is durable over time
- Provides a rationale justifying its use
- Outlines implications
- Testable



- A “motherhood” e.g. easy to use
- A general business statement e.g. profitable
- Little general applicability
- Stated at too low a level e.g. we’re a .net shop
- Not durable
- More than an opinion e.g. because “I say so”



65

Principle Vs Policy

The main difference between **principle** and **policy** is that a **principle** is a rule that has to be followed while a **policy** is a guideline that can be adopted. **Principles** and **policies** are obligatory elements in the proper management of a legal system, a government or even an organization.