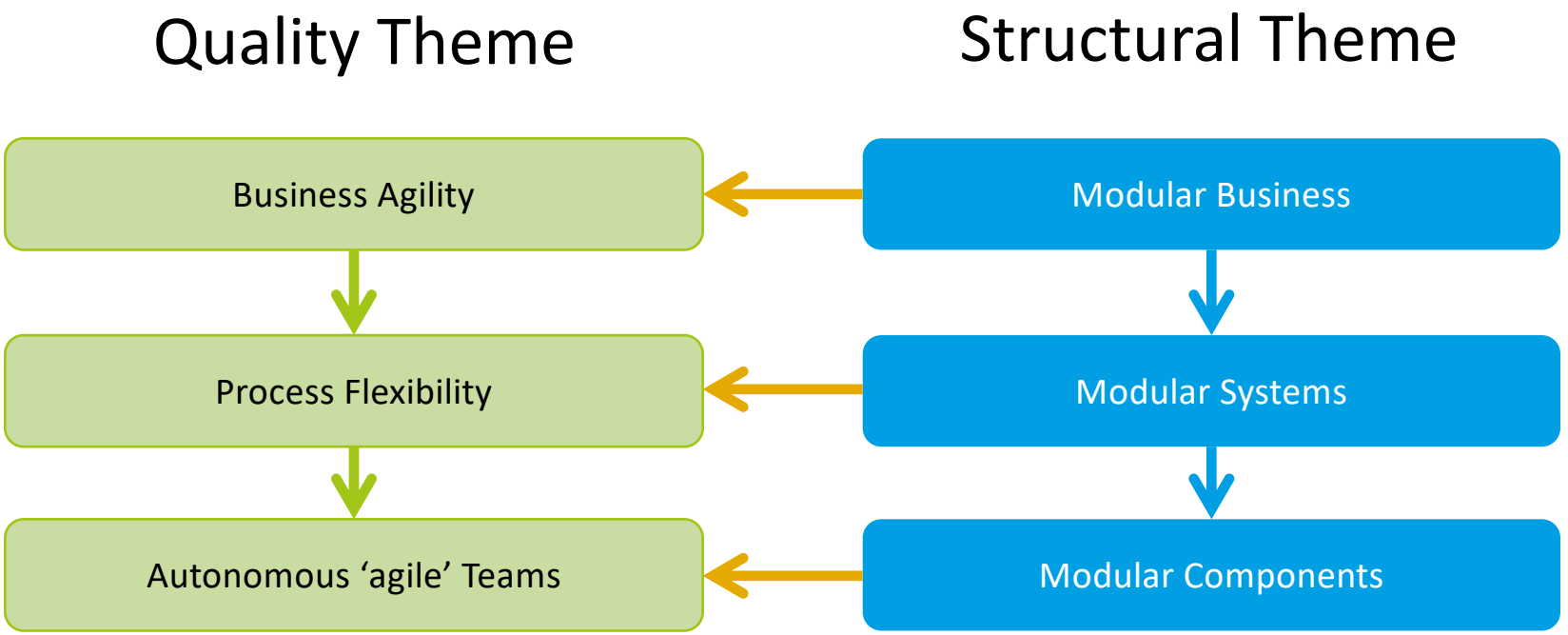
How to scale development – today ?
Architecture, Agile, CI/CD, DevOps

Kim Horn

Version 1.83 **15 January 2019**

Continuous Themes



Definitions

Continuous Integration (CI) simply means developers integrate the mainline several times a day.

Continuous Delivery (CD) means always being able to put the integrated product into production, and place the release decision into the hands of the business.

Continuous Deployment (CDP) is the real value step. Delivered software is automatically deployed into production when automated tests pass. Place decisions into the hands of customer. Continuous deployment implies continuous delivery but not vica versa.

DevOps is the outcome of applying Lean principles to the IT value stream. It builds on Infrastructure as Code (IaaC), CI, CD and CDP.

Evolutionary Architecture (EA) evolving and refining an architecture iteratively and continuously, just in time, to keep teams productive (build out a runway). Progress measured against a fitness function that is a collection of quality goals for the system.

*remaining slides use CD to mean CI/CD/CDP

CD – Principles

There are five principles at the heart of CD:

- Build ***quality*** in;
- Work in ***small*** batches;
- Computers perform ***repetitive*** tasks, people solve problems; automate, ***'jidoka'.***
- Relentlessly pursue ***kaizen***; continuous improvement;
- Everyone is responsible.

CD is derived from the Toyota's TPS* - Continuous Flow manufacturing. These are the 'production line' parts of software development, following on from the up front design activities; they are not the design processes themselves. Lean principles of DevOps also derived from TPS.

Toyota had a separate process, TDP*, for the up front design.

* Toyota Production System & Toyota Design Process

Potential Benefits

- Low Risk Releases – they are small and cheap;
- Faster time to market – means faster feedback: experiment, measure, change.
- Improved quality – automated;
- Lower costs;
- Improved customer satisfaction;
- Improved quality of life and work – no late nights, weekend deployments.

These must trace to Business Drivers (Requirements). For example, a business that does not need speed of release as it has an agreement with customer to release every 4 months, to reduce impact of process change, may not gain a cost/benefit.

Releasing to customer is different to internal release, and has different benefits, issues and requires different architectural tactics.

Foundation Concepts – Dev Ops

Continuous Deployment

Deployment Pipelines

Deployment Automation

Configuration Management

Continuous Integration

Continuous Testing

However – its not all dev ops.

Continuous Deployment

Scientific Method – Experiment, Measure

Deployment Pipelines

Component Pipelines

Evolutionary Architecture

Deployment Automation

Infrastructure as Code

Continuous Ways of Working (Team, Culture)

Configuration Management

Continuous Integration

Continuous Testing

Continuous Delivery – 2 Main Concerns

1. **Architecture Drives CD.** Understand and deliver based on Systemic Qualities; NFRs. Provide the right architecture for:
    1. **The CD Pipeline:** Architect for Build, Test and Deploy for each system platform and environment;
    2. **Enterprise Systems: Architect a Component Based Distributed Organisation;** Based on distributed components that can be independently and continuously architected, created, replaced and deployed in 'small' batches.
    3. **Utilise the cloud** in the most effective way for the organisation. All environments are virtual and built on demand. Utilise the rich set of cloud capabilities.

2. **Culture -** Work teams must support a **continuous** 'way of working':
    1. **Scaling down** (not up) teams and process, with well focused (cohesive) goals;
    2. Reduce the dependencies (coupling) between teams;
    3. Small independent teams can **release 'small' batches** without the overhead of complex communication and complex program management;
    4. Small teams working on small independent components can **reduce and control, and isolate risk**;
    5. Continuous means no discrete projects but ongoing streams of work.

To succeed with CD requires many changes

- **Requires a fundamental strategic change in the organisation and particularly the Requirements and Architectural ‘way of working’;**
- Architecture is required much earlier than traditional projects; as from day one, environments will go into operation;
    - Just in Time (JIT) starts happening on Day 1;
- Architecture needs to evolve incrementally based on smaller modular and more flexible components;
- Smaller components means smaller teams;
- Smaller components are easier and cheaper to replace;
- Loosely coupled components means autonomous, loosely coupled and independent teams;
- Small independent teams improves agility, with faster less overhead ridden delivery;
- Small teams work hand in hand with business more easily;
- Requirements need acceptance tests early on, that trace to unit and other tests, that align to the components.

Quality and Requirements

Web Scale is about Quality

"Operations at Web Scale is the ability to *consistently* create and deploy *reliable software* to an *unreliable platform* that *scales* horizontally."

Jesse Robins

Architecture provides the component model to support the independent creation, and the reliable, automated and continuous deployment of tested systems, allowing the scaling of both the systems and the organisation developing and operating the systems.

A new way to do requirements is required, that is 'Complete' and includes Qualities (NFRs) such as *consistency, reliability and scalability*.

Importance of NFRs (Non-Functional Requirements)

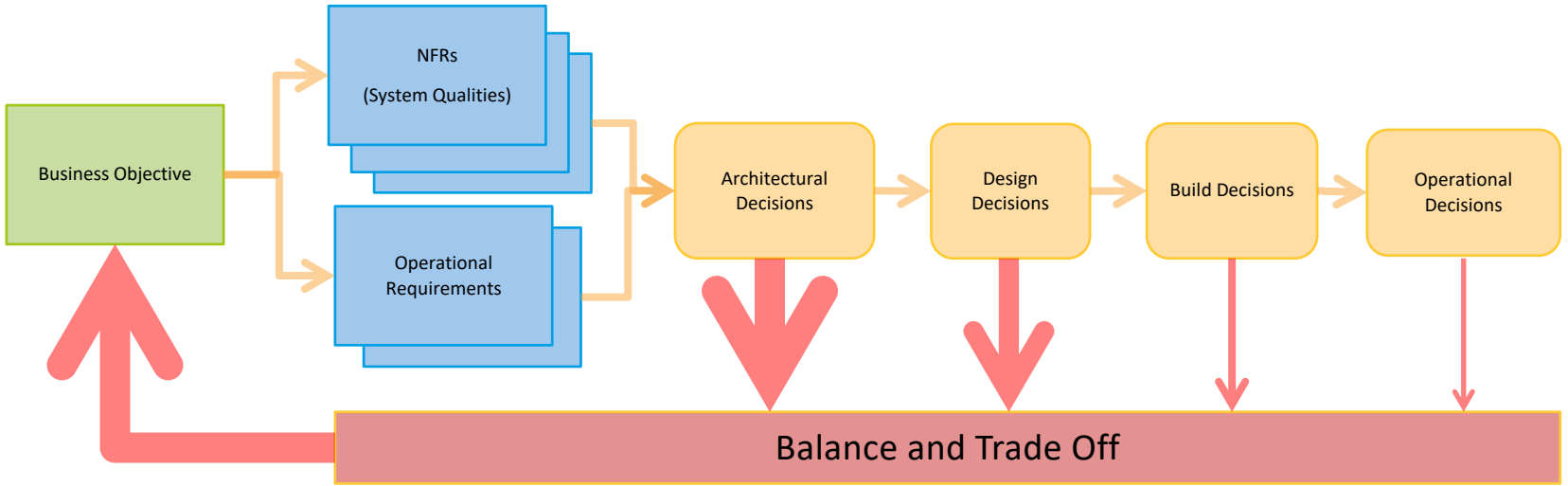
Architecture is driven by Quality, defined by the NFRs, not the functional requirements.

1. Customers ‘feel’ quality, the NFRs, it is the thing they complain about first;
2. The Non-Functional Requirements (NFRs) need to map to the business drivers of the system;
3. NFRs need to be objectively ‘testable’; a component based approach provides a means to effectively ‘isolate’ this quality;
4. Jidoka: Automated Testing of Quality should be strived for;
5. Meeting NFRs at a component level does not mean system will provide these NFRs;
6. A mature approach is required to describe and analyse NFRs;
7. Generic and motherhood statements of NFRs are not good enough, specific testable scenarios are required;
8. The qualities provide the goal for architectural work. In evolutionary architecture they are measured and combined to provide a fitness function, to be maximised, and used to assess progress and decisions.

Non-Functional and Operational Traceability

Traceability from Business Objectives to the Non-Functional (system Qualities) and Functional Operational Requirements and then across the various decisions made down to operations is critical.

- Architectural Decisions should catch the main concerns, but then decisions and issues down the path, in design, build and operations also have impacts.
- Balance and Trade off of Requirements is required, that has an impact on the business objectives.
- It is a cyclic E2E feedback process.



CD implies specific Architectural Qualities - For Example

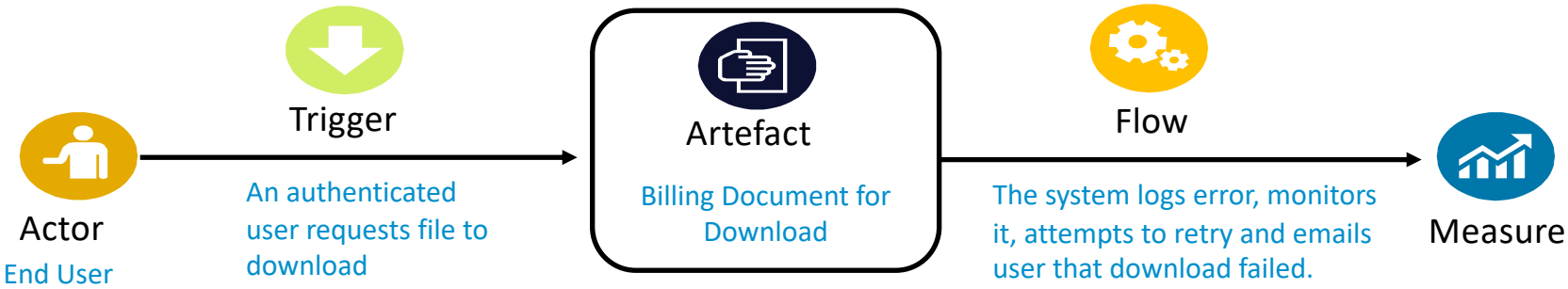
Deployability is concerned with how the executable arrives at the host platform and how it is invoked and configured. Systems may be deployed and delivered continuously or at fixed releases, they may be done while the system is executing.

- ***Push-button Deployability*** is the degree to which a package can be deployed and configured in an automated manner;
- ***Reproducibility*** is the ability for a system to be brought into a good known state, within a specified environment, verified that it's in the intended state, and kept in that intended state on an on-going basis. Provides **replication** for elasticity.
- ***Replaceability*** is the ability to re-connect a replaced failed system or component back into service after it has been reproduced.

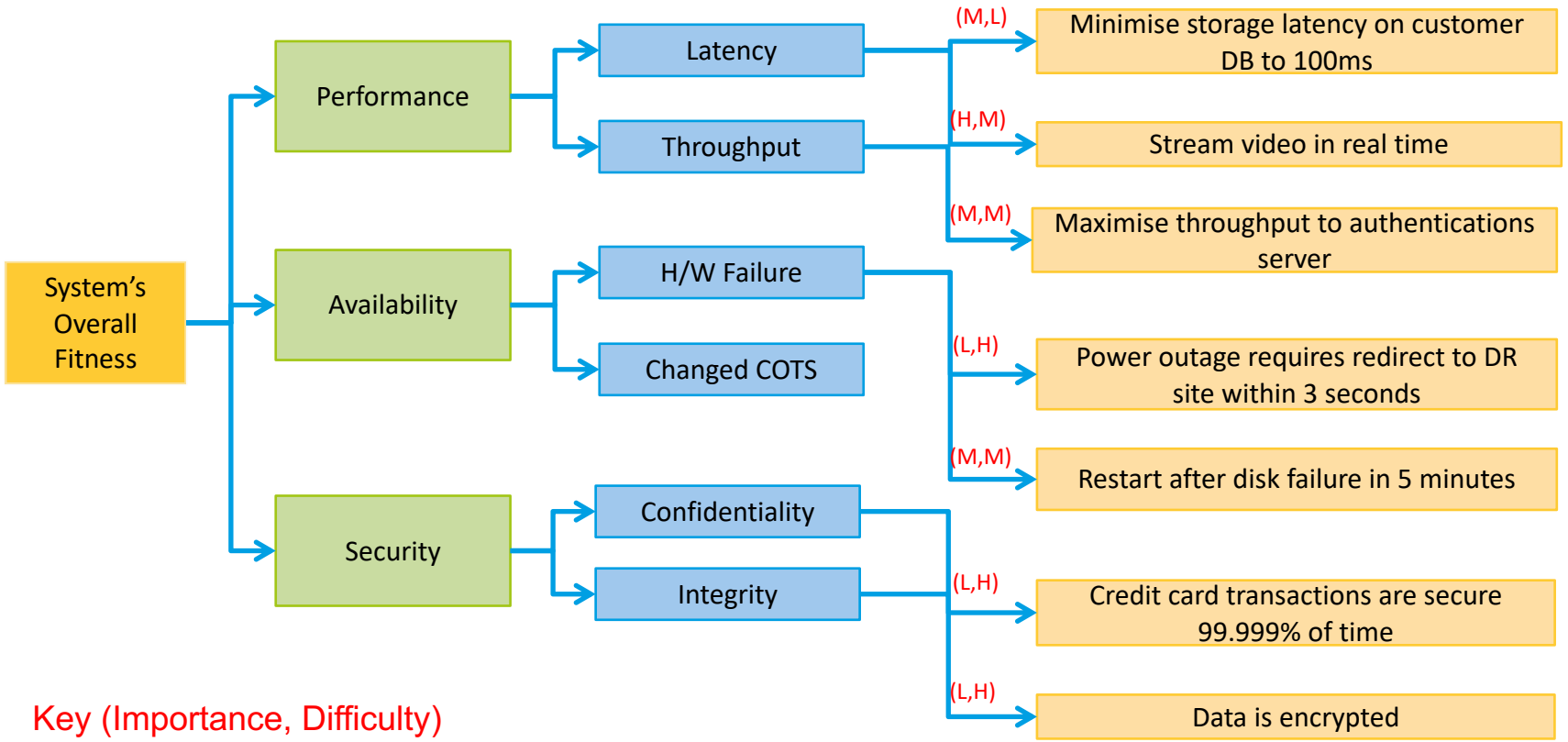
Testability is the degree to which a systems quality can be tested. Systems can be tested by hand, using automated tools and by inspections. A component based and modular architecture and well defined requirements contribute to testability. **Automated testing can occur continuously** throughout the software delivery lifecycle to support continuous delivery, and not just once as during a formal single release test phase.

Approach to NFRs using Scenario Template – Availability Example

A graphical scenario describes unambiguously the specific objectively measureable and testable qualities. Each scenario is based on 7 aspects:

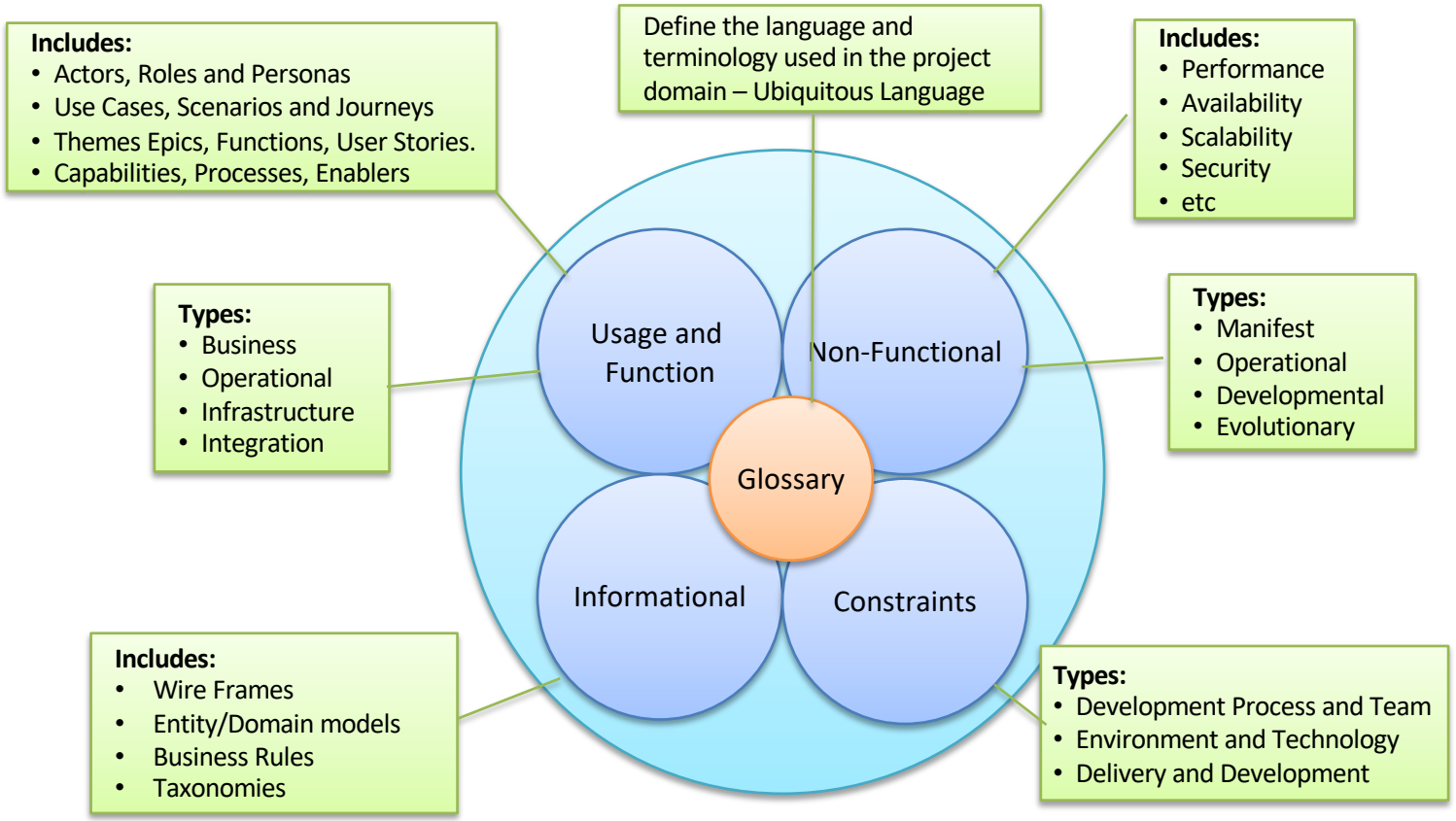


Quality Tree – a way to visualise the Scenario Forest

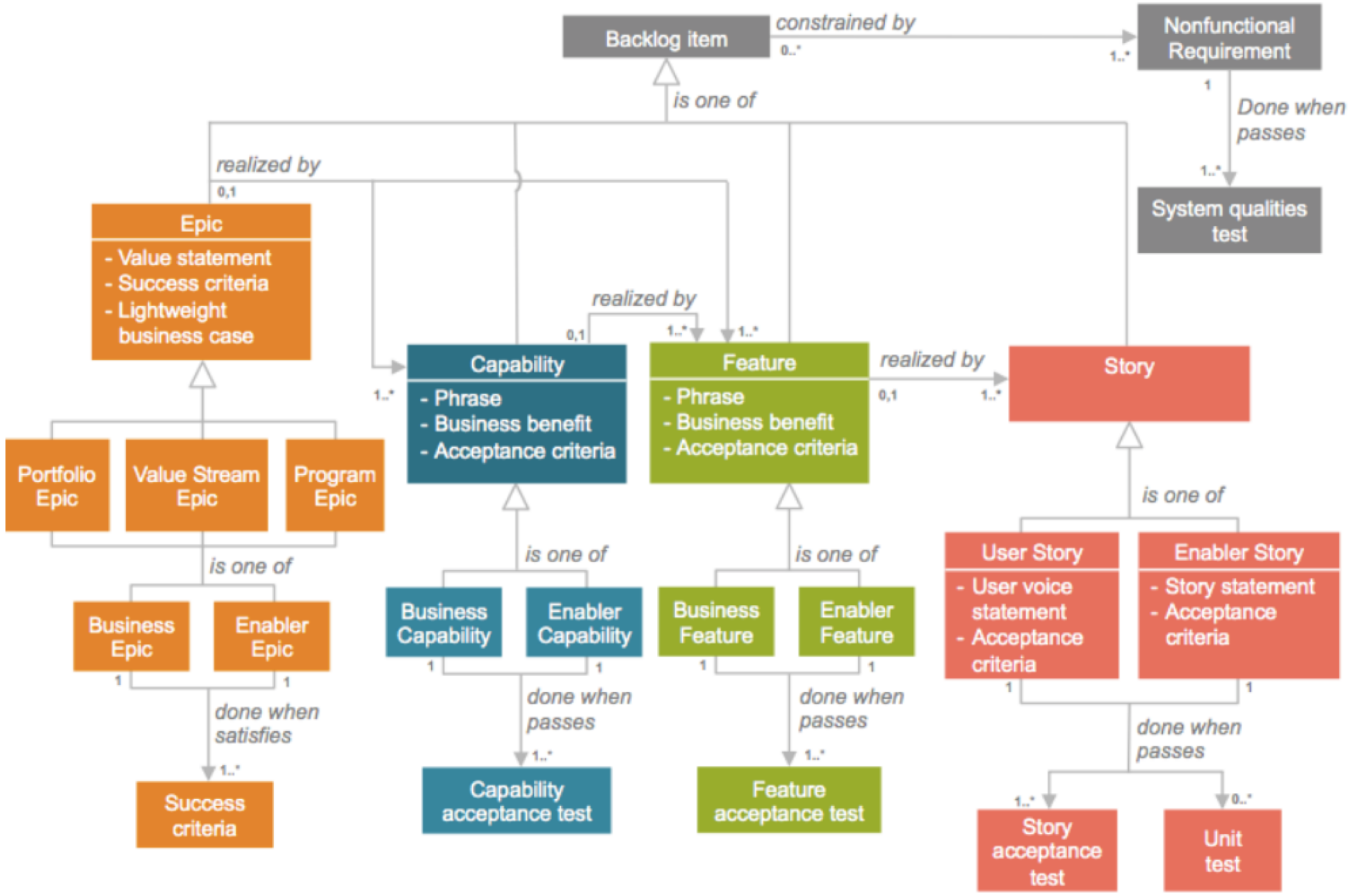


Key (Importance, Difficulty)

‘Complete’ Requirements approach required, not just NFRs



SAFe Requirements and Backlog Model



Unites: Non-Functional, Traditional and Agile Requirements, at various levels of granularity and organization levels, into backlog model for execution and planning.

Decomposition

Decomposition

Architecture gives us intellectual control over the very complex by allowing us to substitute the complex with a set of interacting pieces, each one of which is substantially simpler than the whole.

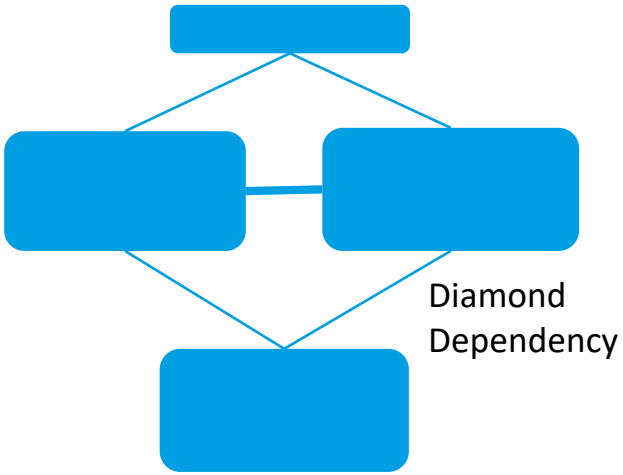
The prudent partitioning of a whole into parts is what allows groups of people—often groups of groups of people separated by organizational, geographical, and even temporal boundaries— to work cooperatively and productively together to solve a much larger problem than any of them would be capable of individually.

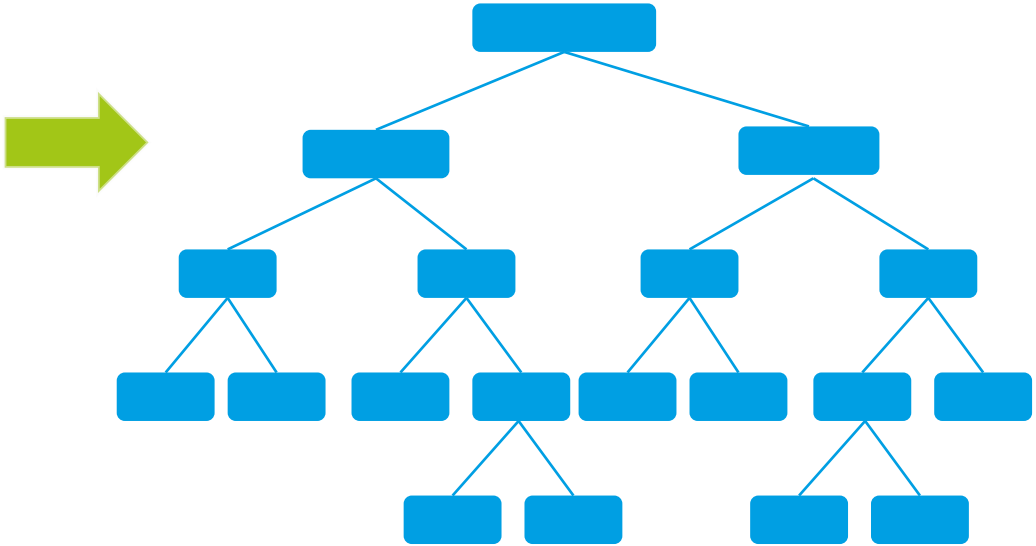
"**Software Architecture Documentation in Practice:"**
Bachmann, Bass et al 2000

CD is completely dependent on Good Quality Decomposition

- Only Components not systems can be deployed Continuously;
- Break monoliths down to small loosely coupled sets of components; These are the basis for small batches of work;
- **Building, deploying and managing small components provides Agility and improves quality;**
- Loose coupling requires a decomposition framework;
- Reliability in distributed component systems requires architecture to manage quality;
- Principles and metrics are required for good quality components;
- Understanding NFRs at a component level is easier than for the whole system. However component quality does not imply system quality;
- **Open source the organisation's components:**
  - Re-use from what's been done before, not as a primary goal of a component;
  - Components used and improved by those that take responsibility.
- Small components mean they can be understood, built and improved by any team, no specialised team required.

Monoliths to Multiple Independent Modules





- Three Monoliths
- Two Large Teams
- Tightly Coupled – dependencies:
  - Diamond
  - Circular
- Long synchronised release time frames

- Loosely Coupled
- Independent Components
- Limited Dependencies
- Multiple Independent Small Teams
- Short Independent releases

Tyranny of Dominant Decomposition

The 'Tyranny of the Dominant Decomposition'* is being forced to decompose a system in a single way at all times (e.g. a single type hierarchy) implies only having a single perspective. Achieving Architectural goals depends on the ability to separate all concerns of importance.

Poor decomposition includes:

- Pure functional decomposition, too early;
- Ad-hoc, having grown 'organically' without any planning or refactoring;
- Excessive dependencies between parts, cyclic complexity;
- Requires complex interrelated set of systems to be deployed;
- Deployment has to be orchestrated (ordered), and is not independent;
- Unnecessarily complicated, poorly encapsulated interfaces;
- Unspecified or ambiguous, as to how systemic qualities are achieved;
- Can't be managed on their own;
- No consideration of future operations, maintenance and support, e.g. can root cause of faults be easily identified.

* Coined by Ossher and Tarr, 1999

Quality Decomposition provides Separation of Concerns (SOC)

Done well, SOC provides a host of crucial quality benefits:

- Additive: rather than invasive, change;
- Maintainability: Low coupling reduces the impact of change in one module affecting another; High cohesion means changes are localised.
- Flexibility and Extensibility: Low coupling means easier to introduce new or changed modules. High cohesion means concerns are localised.
- Reusability: Low coupling makes modules more re-useable.. High Cohesion means the module is well-defined and complete. However, re-use should not be a primary goal of a component, as it creates dependencies;
- Testability: testing can be focused on each module, each concern; faults can be isolated and localised (both run time and test/build time);
- Integratability: Simplified component integration;
- Understandability: Improved comprehension and reduction of complexity;

and the ultimate goal of "faster, safer, cheaper, better" software.

Eight Fallacies of Distributed Systems

Reliability in distributed systems is determined by the weakest component. The number one hot spot for serious system failure is network communication. Unfortunately, architects and designers of distributed systems are often incorrect in their assumptions about network behavior and eight fallacies exist.

These fallacies are more important today with reliance on networked components using REST/HTTP API SOA style services, as there is no protection by the transport. Peter Deutsch and others from Sun listed these eight fallacies.

1. The network is reliable;
2. Latency is zero;
3. Bandwidth is infinite;
4. The network is secure;
5. Topology doesn't change;
6. There is one administrator;
7. Transport cost is zero;
8. The network is homogeneous.

These fallacies mean more development and architectural work is required to achieve quality when the underlying transports do not provide them.

Decomposition, Isolation and Fault Tolerance

- Isolation of components is required to ensure systems are fault tolerant;
- Isolated components are decomposed by how they expose two concerns:
  - their functionality (and others, see tactics later)
  - the impact of failure;
- So when a component fails it does not propagate failure throughout the system, it is contained;
  - Components should **fail fast**, not linger around in a state of death;
  - Components should **fail early**, communicate their failure, so they can be dealt with quickly and not consume resources;
- When components are isolated they can be run on parallel;
- Components that are isolated and running in parallel provide scalability.
- Architect for Failure, implement quality patterns: timeouts, retries, circuit breakers, bulkheads,

- **Components that are isolated can be deployed easily, independently, without complex orchestration, and versioning issues.**

Failing Well – Platforms are Just Unreliable

To provide Reliable Systems, as opposed to just reliable components, requires more than highly reliable platforms.

- Components that can be very easily deployed, replaced quickly with improved versions;
- Components that have infrastructure instantly available to be deployed onto;
- Component that can be quickly made available, that is connected into the network automatically;
- Component that are modular and have a single responsibility;
- Components that are monitored and make aware their failure;
- Components that can be orchestrated automatically to be re-instated and replaced on failure;
- Components can be replaced effortlessly at low cost;

Trying to make Platforms reliable is a never ending and costly battle.

Benefits of Good Components – Example Qualities

- **Configurable:** Different implementations of the same logical component can be interchanged to provide new capabilities without redesigning the whole application.
- **Manageable:** Components can be located in multiple locations. Fixed, replaced or repaired, started or stopped once.
  - Ideally Management interface is separate to the Functional interface;
- **Increased consistency and flexibility:** With components located in multiple places in the system, and their interfaces remaining unchanged, the code in a component, or the executable, can be changed without impacting the whole system, enhancing consistency.
- **Provide single logical location** for:
  - Change and improvement, reducing the dependency between development teams;
  - Testing;
  - Monitoring;
  - Fault isolation – there is only single point of failure, we know where and why;
- **Simplified component integration;** component exposure;
- **Improves system scalability,** easy to parallelise independent components (functions).
- **Improving development scalability;** They can be developed independently, teams do not have extra communication overheads;
- **Improve reliability;**

Decomposition Tactics

Heuristic	Description
Bounded Context	Partition the domain into areas with a common ubiquitous language, around business capabilities
Layering/Tiering	Ordering of concerns; cohesive services within a layer, expose via interface – low coupling.
Distribution	Partitioning by distribution among computational resources, provides scalability;
Exposure	Decide what to expose to other components, via interfaces, gateways. Separate implementation;
Functionality	Decompose into Functional units ,e.g. service groupings with domain and subdomain boundaries;
Generality	How general or specific to make something to make it reusable across projects or products.
Aspectual	Cross-cutting concerns, implemented as aspects, e.g. logging, security.
Coupling & Cohesion	Partition by Reducing Coupling and Increasing Cohesion.
Volatility	Isolate things that are more, verses less likely to change, or things that simply change on different schedules.
Configuration	Systems must support different configurations (for environments, pricing, usability, performance, security, etc.)
Planning & Tracking	Break a big thing apart so that work can be defined in smaller time units against the smaller parts. Dependent components done later.
Work Assignment	Partition by physically-distributed teams, matching skill-sets, security areas (clearance), contractual concerns, component.

Re-Composition Architecture

After decomposing need to re-compose to run the system. Three ways to do this:

1. **Run Time Binding**
   - Independently deploy ‘small’ distributed components, as ‘Micro-services’;
   - These are Choreographed or Orchestrated as an Application. *Choreography is the better approach as it reduced dependencies;*
2. **Build Time Binding**
   - Create independent ‘small’ components in a library;
   - Combine (integrate) at build;
   - Deploy a monolith.
3. **Combination of the above**

Many tactics apply to each of these and are discussed later.

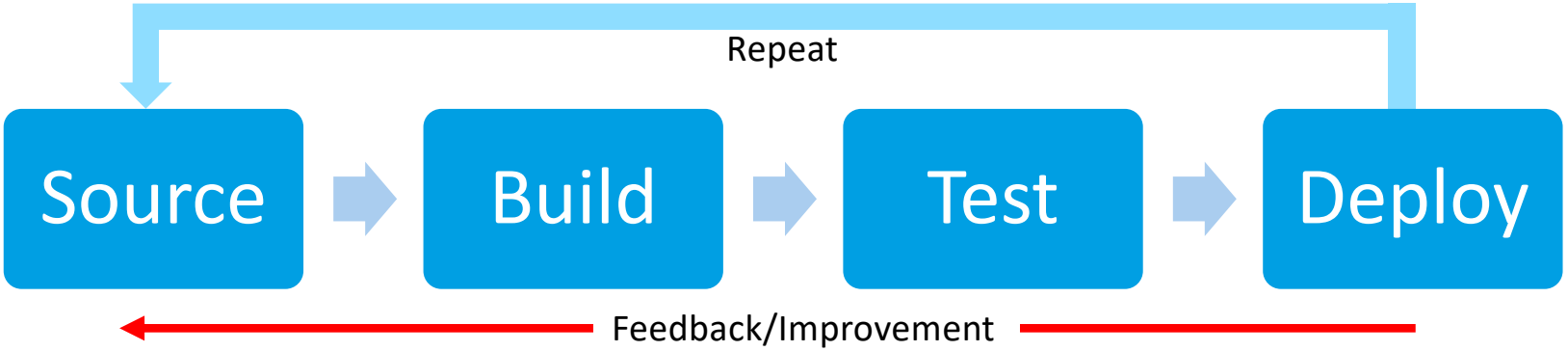
Small runtime components can be chosen, at the last minute, with no **dependencies on single source but from an open selection.**

Deployment Tactics

Major Points

- CD Best Practices;
- The 3 ways of DevOps;
- Infrastructure as Code;
- Immutable Deployments;
- Feature Toggles – do not branch code;
- Blue Green Deployments;

4 Main Phases of CD Release



- Check Source Code into mainline often.
- Peer review new code:
  - Pair Programming

Automated and Integrated build to Main Branch Daily or more often:

- Compile Code
- Unit Tests
- Style Checkers
- Code Metrics
- Package Code
- Create Container Image

Automated Testing:

- Integration
- NFR
- Load
- UI
- Penetration

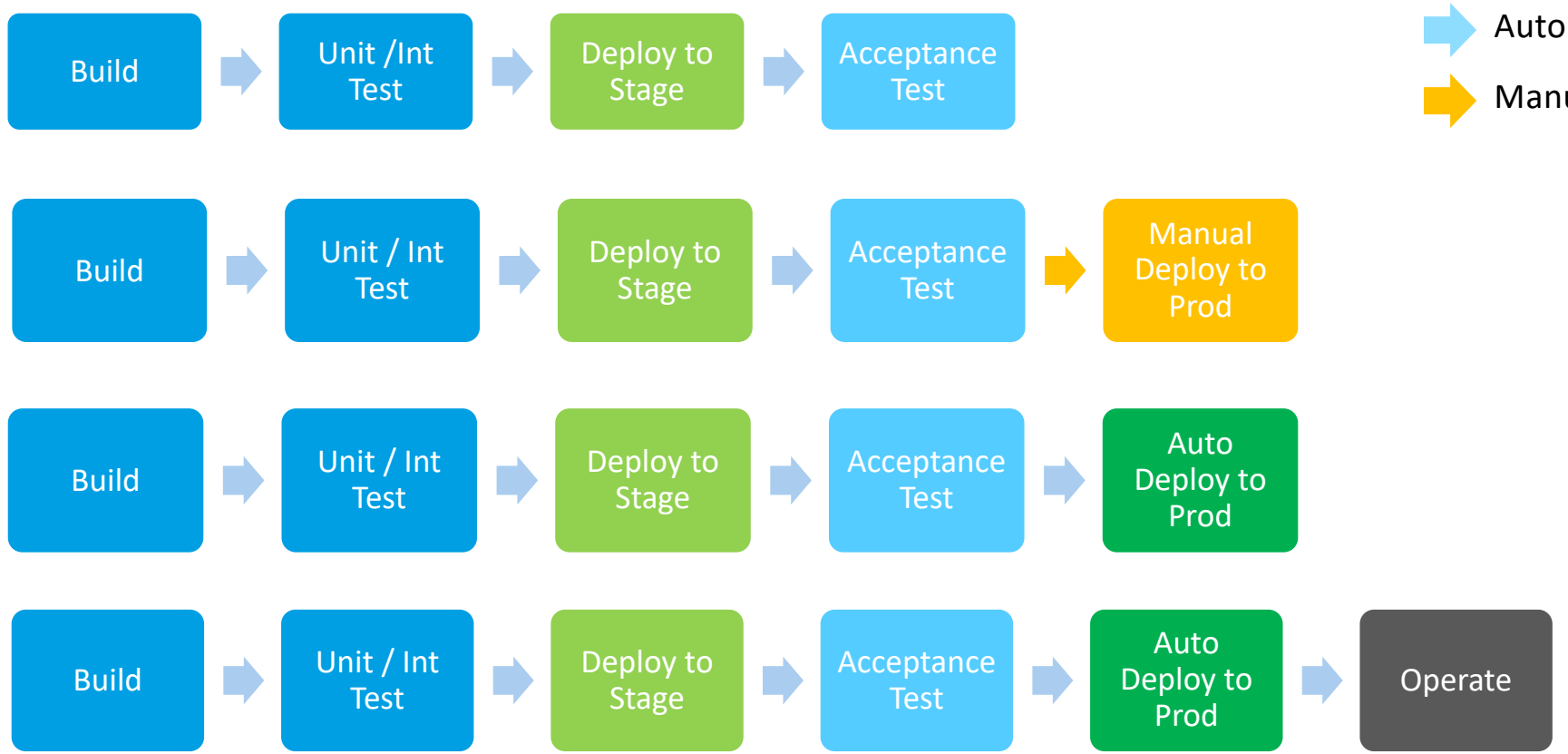
Only code that passes is allowed to be deployed

Automated Immutable Deployment:

- Smoke test environment
- Artefacts are environment agnostic
- Each Environment Configuration pulled

Fail Fast – quick feedback, move failed test to the LEFT !

CI to CD to CDP to DevOps



Continuous Delivery Issues

- When building system's how do we keep deploying into production continuously when parts of the system may not be completed, tested, or ready for the business or customers to have made live?
- How do we do this without branching the code ?
- How do get around deploying the whole system when a small bug fix was made ?
- How do we cope with the complex dependencies in deployment, and not need complex orchestration logic ?
- When decomposing a system into many distributed processes, how do we consistently and reliably distribute configuration to these processes?
- When that configuration needs to change, how do we update that configuration without redeploying all of the processes ?

CD & CI Core Best Practise

Practises:

- Mainline is always deployable, no functional branches (importance of feature toggles)
- Check into Trunk Daily;
- Jidoka: Automatically build, integrate and test;
- Test Functional and Non-Functional Requirements.
- Small independent components remove need for orchestration.

Result:

- Improved Quality; fast feedback about code and integration finds issues early.
- Quality issues can be isolated easily due to small changes;

Requires Architecture supporting:

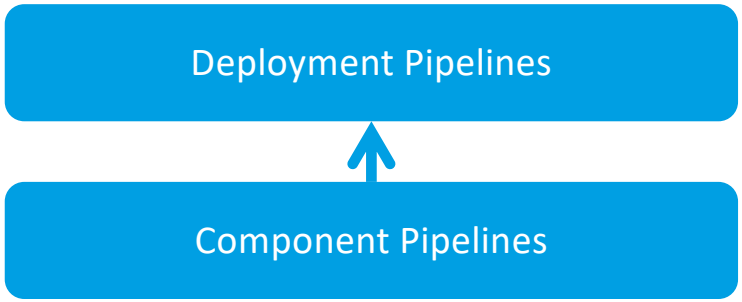
- Small independent units to build and commit – components move issue of branching away;
- Single command to build machines, network, infrastructure and platforms;
- Single command to build applications;
- Single commands to deploy to any environment;
- Components that are independent without deployment dependencies.

The 3 Ways of a Dev Ops Approach – based on Lean

The 3 primary drivers for DevOps and CI/CD:

1. **Systems Thinking**, as such its the whole system with left to right flow from development to operations to customer. IT applications do not work in isolation, they are linked to business, the market as a whole. Practices:
   - CI, CD, Create environments on demand;
   - Limit WIP;
   - Build safe systems and organisations that are safe to change.
2. **Amplify feedback loops** - a constant flow of fast feedback from right to left. Correct continuously. Practices:
   - '*Andon*' - Stop the Production line when builds or tests fail;
   - '*Kaizen*' or continuous relentless process improvement. Become a learning organisation.
   - Code that can always be deployed
3. **Learn from Failure culture** - amplified feedback allows correction quickly and you can fail fast. Don't be afraid to stuff up, take risks and experiment. Create a culture of innovation that removes the fear of making mistakes or taking risks. Focus on Non functional aspects equally.

With CD 'Operations' Happens Early On



Dynamic components deployed into Environments;

Static components tested, code analysed for quality, integrated.

- CD means the above pipelines need to be architected and delivered earlier in a project than usual;
- Operational concerns become Architecturally Significant, as all development relies on them.
- Environments have to built and configured early;
- Platforms made interoperable early;
- Failing Well and Fast applies to deployment and components;
  - Faster we fail (safely) the faster we fix;

Deployment Pipeline Practices:

- **Immutable Deployments - Only build packages once**. We want to be sure the thing we're deploying are the same thing we've tested throughout the deployment pipeline, so if a deployment fails we can eliminate the packages as the source of the failure. Don't ever patch production as then we don't know what's there.
- **Deploy the same way to every environment**—including development. This way, we test the deployment process many, many times before it gets to production, and again, we can eliminate it as the source of any problems.
- **Deploy the same thing to every environment**— rather then deploying systems built for specific environments, deploy the same system to all environments that pulls their unique configuration (from the environment not the system);
- **Smoke test your deployments**. Have a script that validates all your application's dependencies are available, at the location you have configured your application. Make sure your application is running and available as part of the deployment process.
- **Keep your environments similar**. Although they may differ in hardware configuration, they should have the same version of the operating system and middleware packages, and they should be configured in the same way, e.g. pull from a central config repository. This has become much easier to achieve with modern virtualization and container technology.

Infrastructure as Code

- Pipelines not only support applications and code changes but Infrastructure.
- Both Infrastructure (VMs, Network, etc) and Application Code are under Software Configuration Management;
  - Configure rather than build environments.
- Build infrastructure ‘Just In Time’, pull it down ‘Just in Time’;
- Infrastructure managed via APIs;
- Architect core services to build and manage the full stack;
- Architect to support Abstract, Virtualised Infrastructure:
  - Abstract away the infrastructure;

Infrastructure as Code enables automated provisioning for deployment, only when resources are needed, enabling Web Scale Reliability.

Feature Toggles

Are configurable switches to turn features on or off within a released environment.

- Decouple Deployment and Feature Release in CD.

Goal is that mainline is always deployable and feature branches are not required. Various types of Toggles:

- **Release toggle:** Hide partly built features, that take longer than one release cycle, and keep everyone working on the mainline.
- **Experiment toggles** to activate real time testing, e.g. A/B testing;
  - A canary launch is rolling features out to a small number of users to assess the reaction of the overall system.
- **Ops Toggles** to provide controls for operations staff;
- **Permissioning Toggles** to control access of features for different subsets of users.
- **Issues:** *who pays to remove toggles, many features need feature management tools and the logic gets complicated*

Branching is not CI/CD

"Perhaps the most important practice that makes continuous integration possible is that everybody checks into mainline at least once a day…. If you're not doing that, you're not doing continuous integration."

Jez Humble

Not doing branching at all is just too restrictive, its permissible to branch and merge within the day.
Long term branches should no be allowed, as they break the trunk with merge, and can't be deployed.

Longer lived need for branching should be tackled by:

- Feature Toggles – hide the changes;
- Break the change into smaller pieces, each releasable;
- Components – isolate the change to small components that can change fast, and provide a consistent API.

Blue Green - Deployment Architecture

Symptom of Illness: Deployment outside Business Hours

Principal: Separate Release from Deployment

- The blue-green deployment approach addresses this by ensuring you have two production environments, as identical as possible.
- At any time one of them, e.g. blue, is live.
- As you prepare a new release of your software you do your final stage of testing in the green environment.
- Once the software is working in the green environment, you switch the router so that all incoming requests go to the green environment - the blue one is now idle.
- This switch can be gradual; maybe via Canary testing;
- And so on…..

Modular Business

Major Points

- As with IT systems the business itself can be treated as a system assembled from components;
- **Business Components provide Business Agility;**
- **IT Components provide IT Agility;**
- Each component of the business exists in its own domain;
- Each domain is self contained and ‘Bounded in Context’;
- DDD – Domain Driven Design;

4 Stages of Enterprise Maturity*

1. **Business Silos:** Applications built by local business units that are independent of other applications in the business.

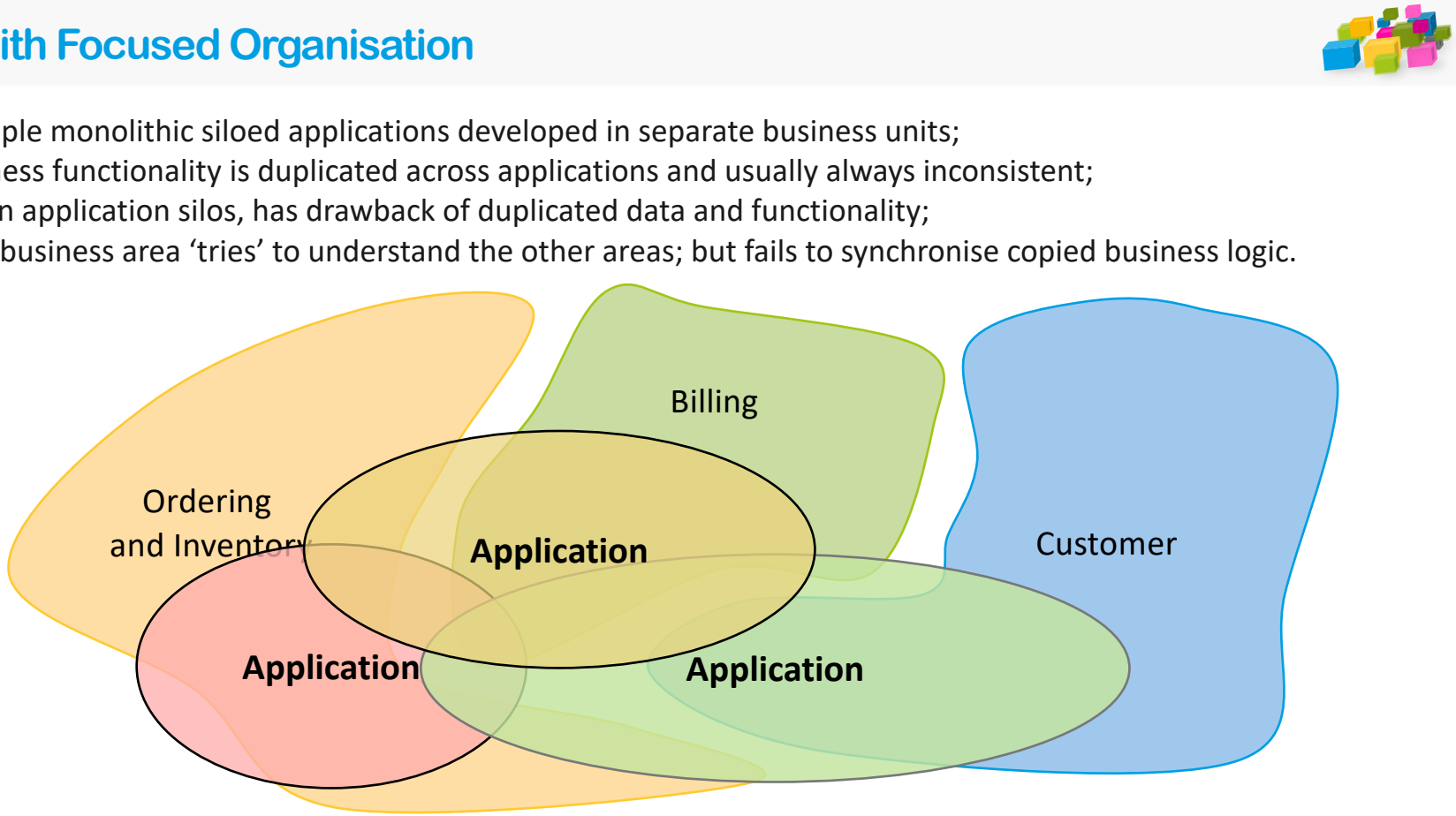
2. **Standardised Technology:** Employ a standard set of infrastructure across business units and applications. Decrease the number of platforms managed.

3. **Optimised Core:** Move from a local view of data and applications to an enterprise view. Embrace the principle that standardisation enables innovation.

4. **Business Modularity:** Business Services that can be swapped in or out, re-used, and provisioned internally or externally. Enables strategic agility.

* Wells et al, Enterprise Architecture as Strategy

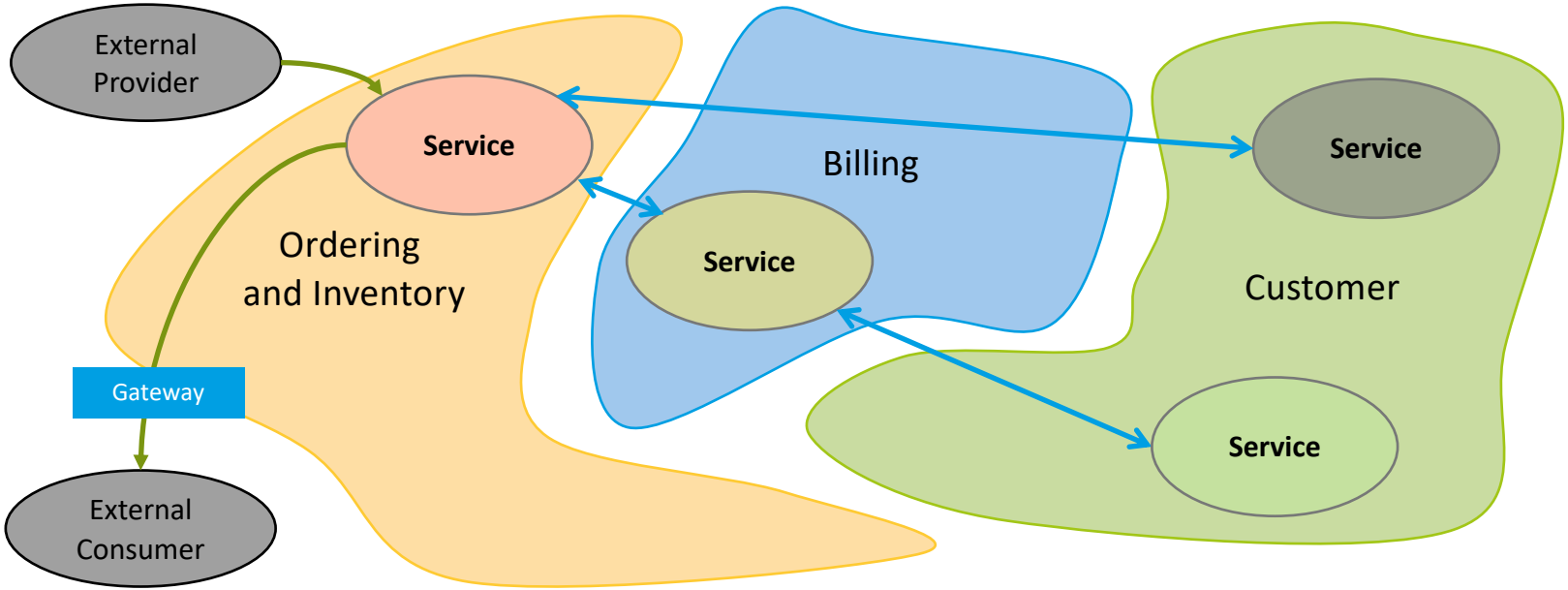
Monolith Focused Organisation



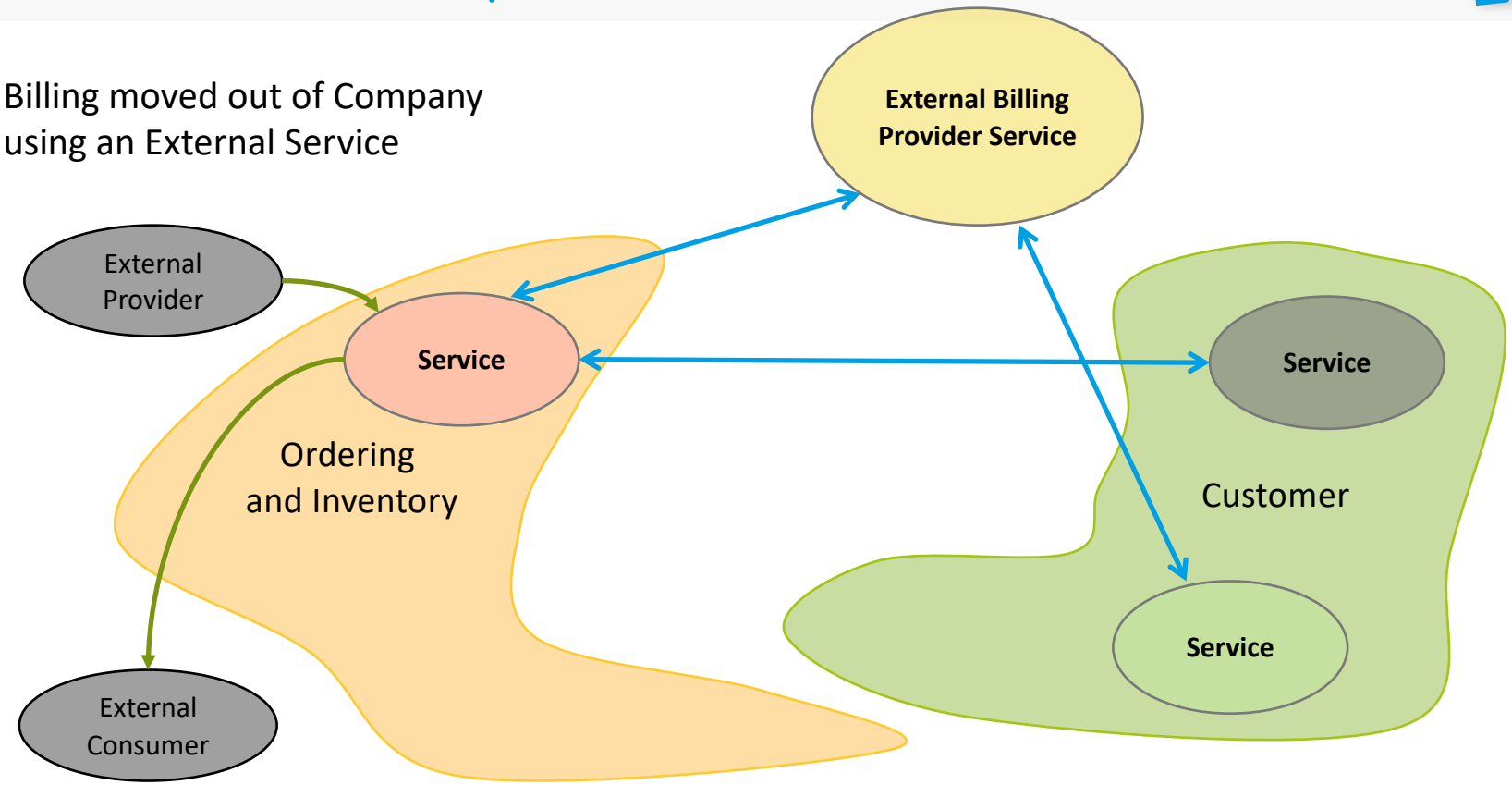
- Multiple monolithic siloed applications developed in separate business units;
- Business functionality is duplicated across applications and usually always inconsistent;
- EAI on application silos, has drawback of duplicated data and functionality;
- Each business area ‘tries’ to understand the other areas; but fails to synchronise copied business logic.

Service Oriented Organisation

- Each Business's Capability is offered as Services.
- Services can be exposed externally and internally.
- Exposing our services may require a Gateway to protect us.



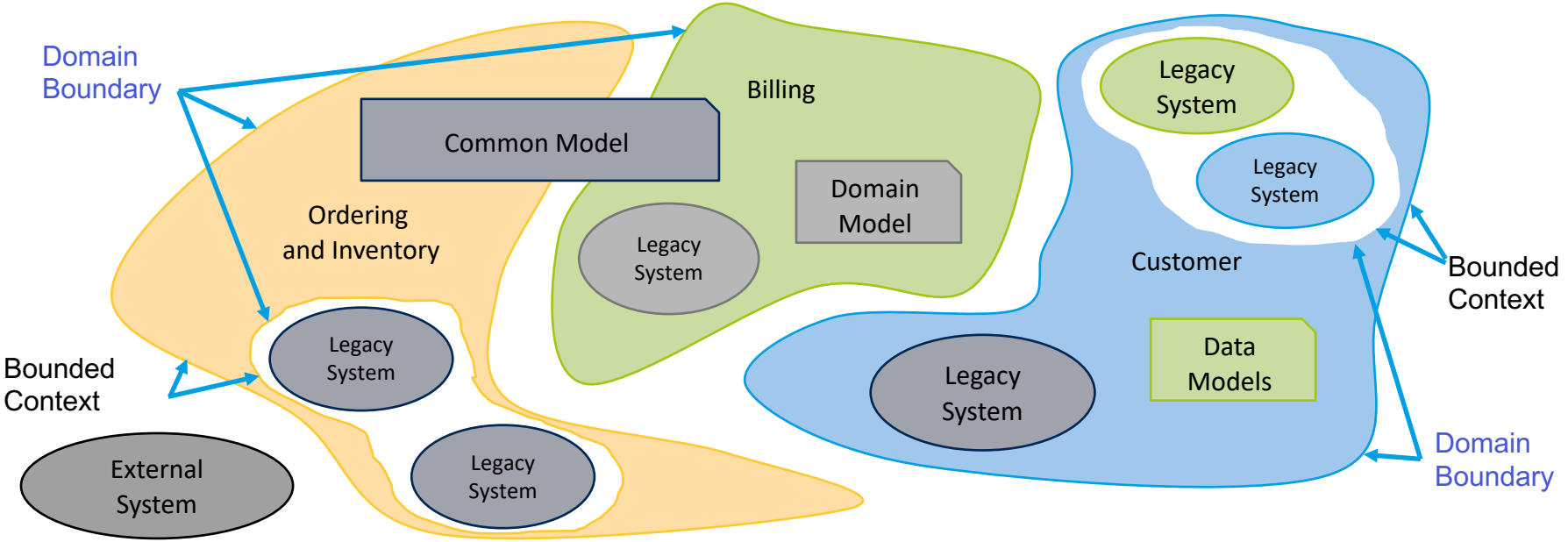
A Modular Business Example



Enterprise has Multiple Domains – Domain Driven Design(DDD)

Each domain may be defined by its own **Ubiquitous Language and Unique Domain model.**

- Some domains may share similar information, data entities and their definitions.
- Other domains may have a different representations, definitions and requirements for entities.
- Legacy system, or external systems, most likely will have different models.



DDD and Bounded Context

"A specific responsibility enforced by explicit boundaries. Each bounded context has an explicit interface, where it decides what models to share with other contexts.

Building Microservices: Designing Fine-Grained Systems
Sam Newman

Domain Boundaries

Systems that share domain models exist within a ***bounded context***. The boundary delimits the applicability of a particular model.

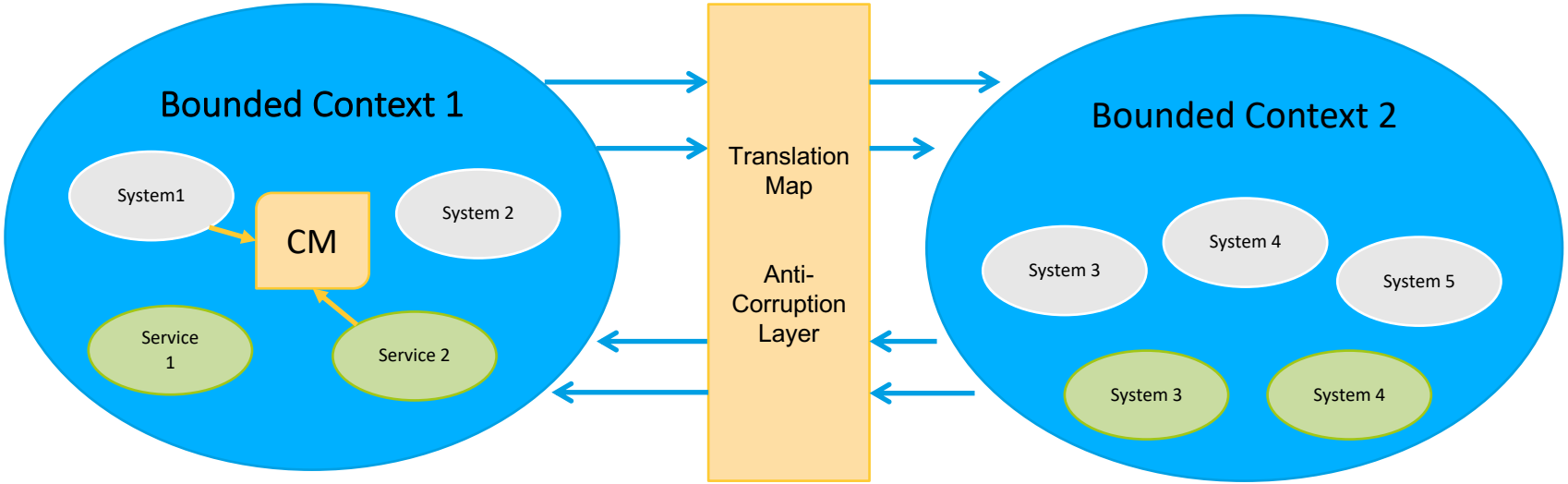
In contrast a large monolithic project may be required to work across multiple domain models.

- Developers, when they combine distinct models produce systems that are buggy and difficult to maintain. The domain logic is never 'copied' correctly;
- Working directly with multiple low level API representations, across bounded contexts, can produce poor software, as data is converted locally in an add hoc manner;
- Legacy systems integration can produce even more disparity and model conversion issues.

- **Total unification of the domain model, across an enterprise, for a large system will not be feasible or cost-effective. Enterprise Canonical Models compromise all bounded contexts.**

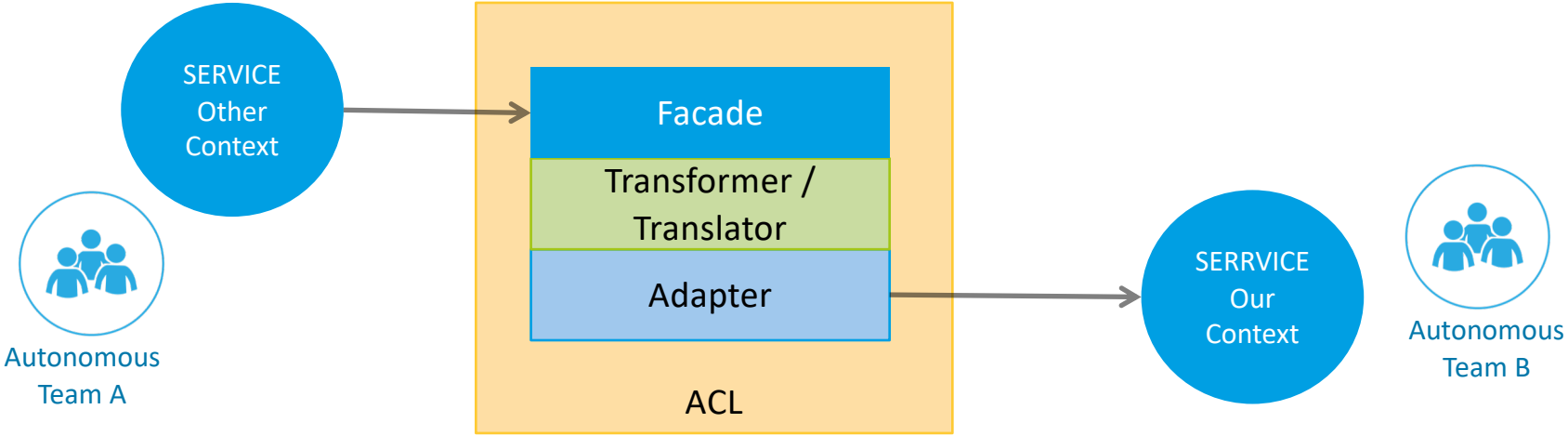
Instead divide up a large system into smaller systems each within an Explicit Bounded Context, with each model protected and isolated;

Tenet: Keep Boundaries Explicit



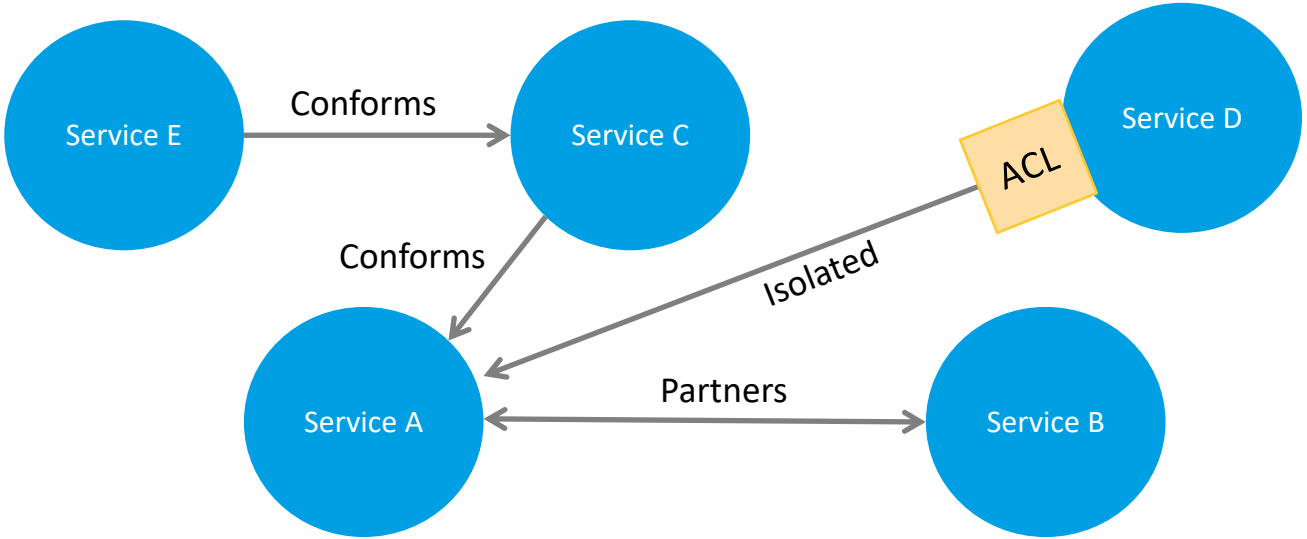
- Each Bounded Context has its own **Ubiquitous Language,** the unique language of the team;
- Each Context has its own domain model, a **Model in Context.** Maybe defined by a Canonical Model (CM).
- Data Conversion, per service, is a low level and expensive mechanism to enhance communication between contexts;
- A **Single Translation Map, across domain services,** provides a well defined relationship between contexts.
- Each Bounded context can implement its own Anti Corruption Layer to reduces corruption.
- Encapsulation: Outside a Bounded Context there is no knowledge about what is behind the boundary.
- Enterprise Canonical Models are not realistic, and become increasingly harder to manage.

Anti-Corruption Layer Components – Isolate Teams



- The *Façade* exposes only the functionality the context will see. It is important to understand that the facade should be defined in terms of the other context's model elements;
- An *Adapter* is placed to modify the interface of the other context and adapt it to the interface expected by our context;
- *Transformer Translator* maps to the elements of our context that are exposed by the facade to the other context. Isolating the two domain models;
- An API Gateway provided one mechanism to do this.

Context Map – Understanding Relationships



- Team E *conforms* to the Service from C, it has no choice, it has to use these services as is, as is not open to negotiation or discussion with its consumers;
- E and C are dependent on A's services. Autonomy is lost. When A changes it forces the conformists to change.
- However D uses an ACL to *isolate* itself, it cannot afford to be impacted by others;
- A and B are *partners* and work closely together; they are OK to live and die together.

Domain Driven Design Building Blocks

- **Services** – operation that do not sit within an Entity; Services have a specific context;
- **Entities** – an object defined by its identity and continuity;
- **Aggregates** – a collection of objects that are bound together;
- **Domain Events** – happening of interest in the domain, can be used to decouple system for high performance;
- **Value Objects** – an object with no identity, still has attributes as per entity;
- **Repositories** – save and retrieve entities and aggregates;
- **Factories** – a mechanism to create objects that decouples the implementation.

Building domain models and context maps is not easy.

Services
Big, Mono and Micro

Major Points

- SOA vs Microservices (Re-use vs Agility);
- **Share as Little as Possible** Architecture – The Myth of Re-use;
- How ‘Micro’ is a MicroService;
- **Consumer Driven Contracts -** The customers specify the services they need, not the providers.
- **Services can be grouped and exposed internally and externally via Gateways.**
- **Independent, decoupled services (components0 at build and run time.**

SOA and ESBs

SOA is an architectural style adopted by many 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 mediate business processes orchestration and integration.

"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;
- A shared monolithic data base that created coupling;
- Re-use was based on predicting future possible 'wants' not driven by real current consumer 'need';
- 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.

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).

The Re-use Myth - The focus for microservices is agile 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 upfront is expensive; it is waste. While it may appear to 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.

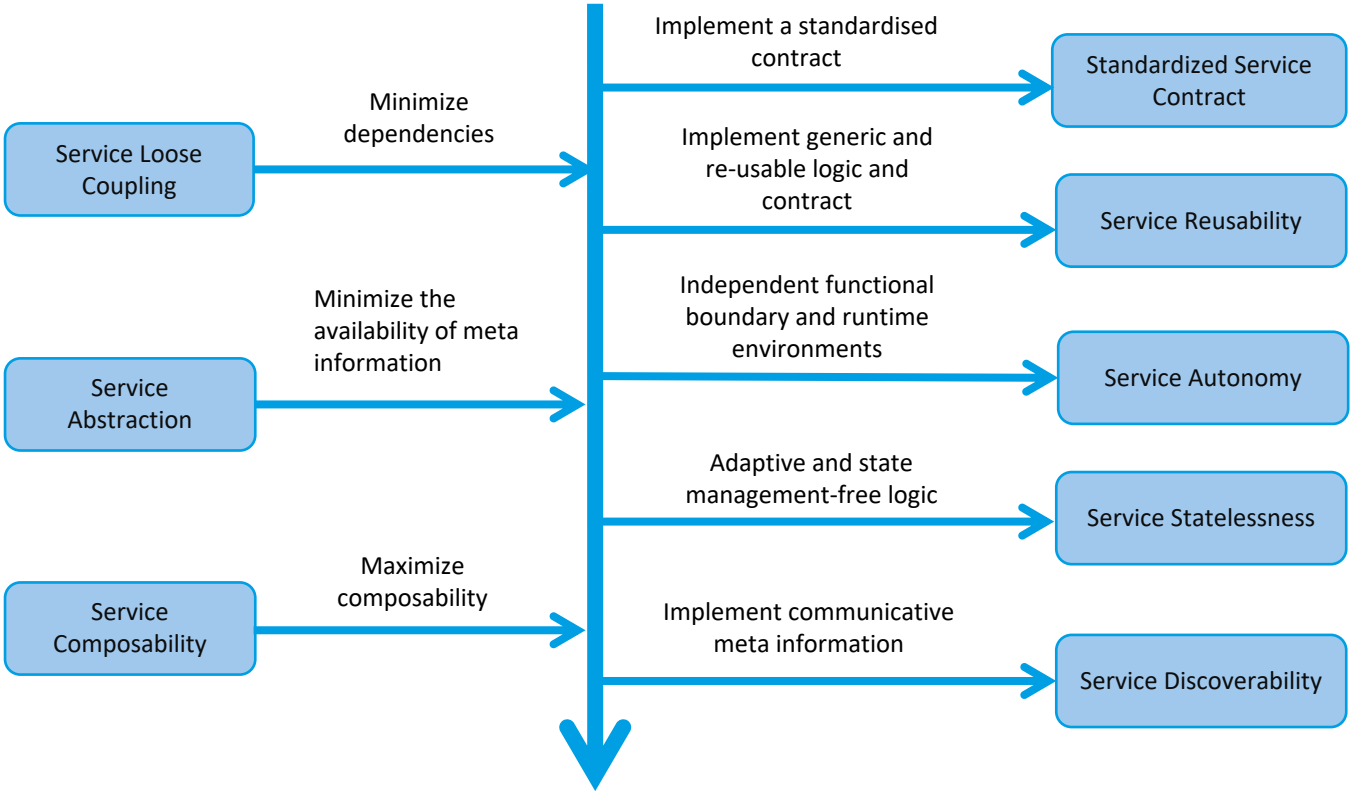
Its about the Prime Driver of an Architectural Style.

The **Microservice Architecture Style** is about the decomposition of an application for agility and isolation. **Share as little as possible.**

The **Service Oriented Architecture Style (SOA)** is about decomposition of the enterprise for re-use. **Share as much as possible.**

- The concept of a *share-as-much-as-possible* architecture appears to solves issues associated with the duplication of business functionality but leads to tightly coupled components, applications and increases the overall risk associated with change.
- **Using both styles is fine but means we have to be aware of the tradeoffs. How Context Relationships effect Autonomy.**
- **Re-use sounds ideal but can be costly and severely reduce agility.**

Eight Principles of SOA (Thomas Earl)



4 Tenets of SOA (Don Box)

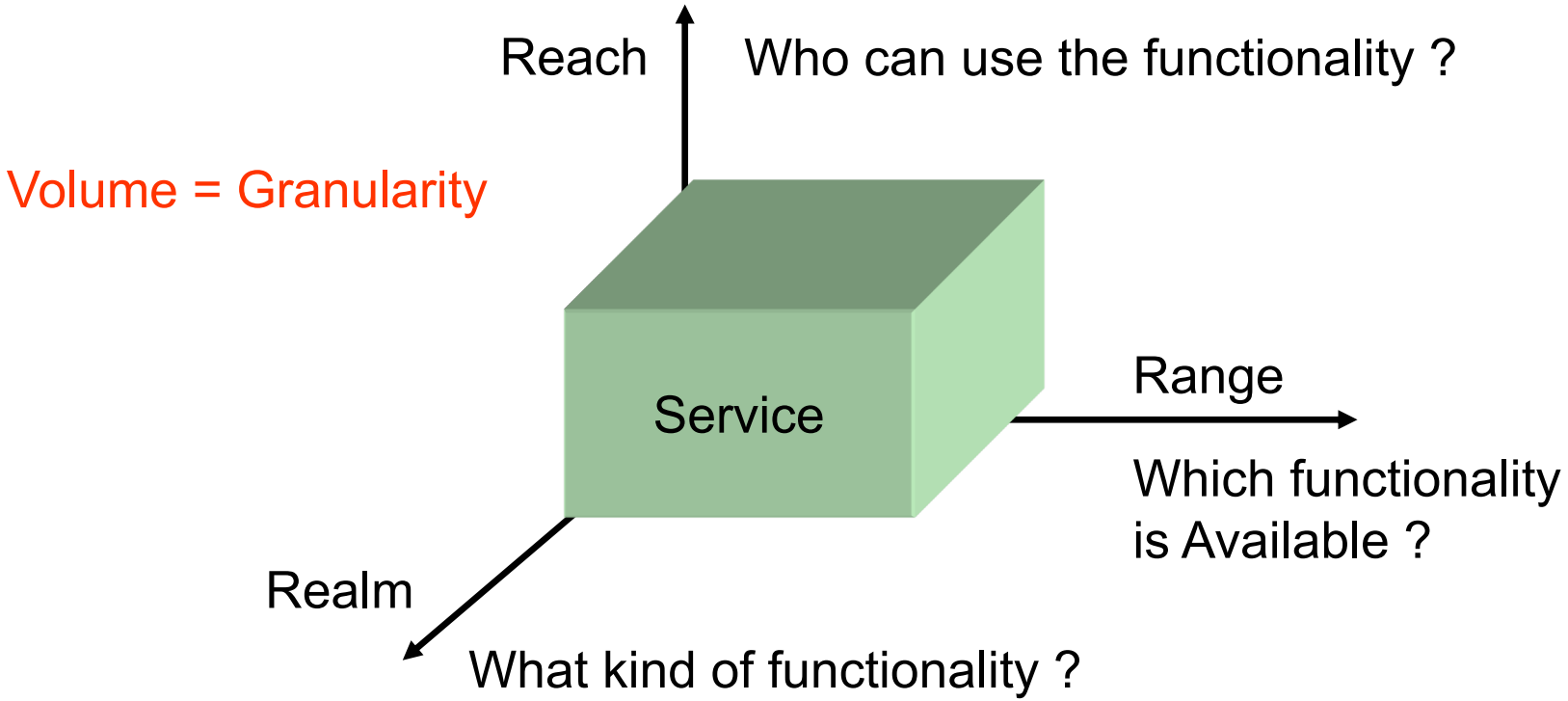
1) **Boundaries are Explicit.** A boundary represents the border between a service's public interface and its internal, private implementation. The interface consists of public processes and public data representations. The public process is the entry point into the service while the public data representation represents the messages used by the process;
2) **Services are Autonomous.** Each service has its own implementation, deployment, and operational environment;
3) **Services Share Schema and contract, not class.** Schema describes the format and the content of the messages while contract describes message sequences allowed in and out of the service. What is not known is how implementation takes place;
4) **Service Compatibility is determined based on policy**. Formal criteria exist for getting "service from the service." The criteria are located in an English document that outlines the rules for using the service.

‘Microservices’

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.”

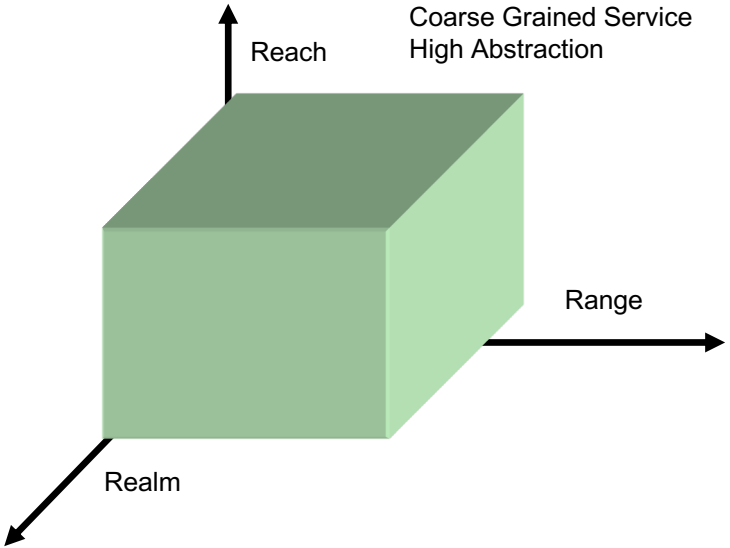
Martin Fowler, March 2014

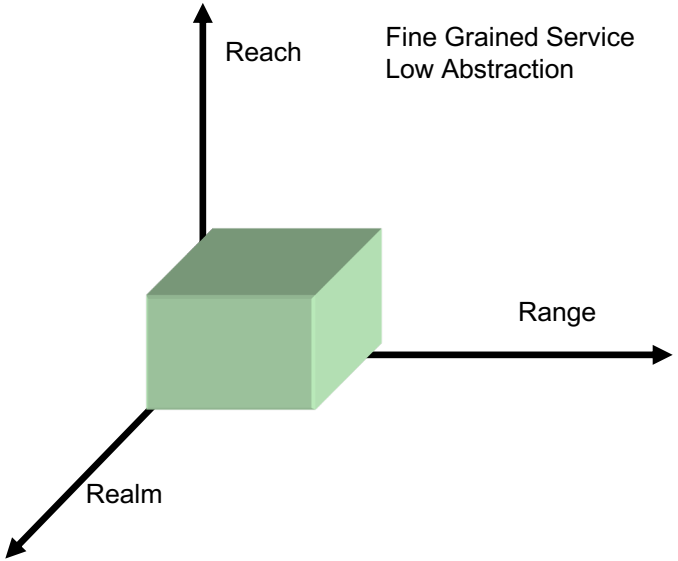
3R's - Realm, Range and Reach



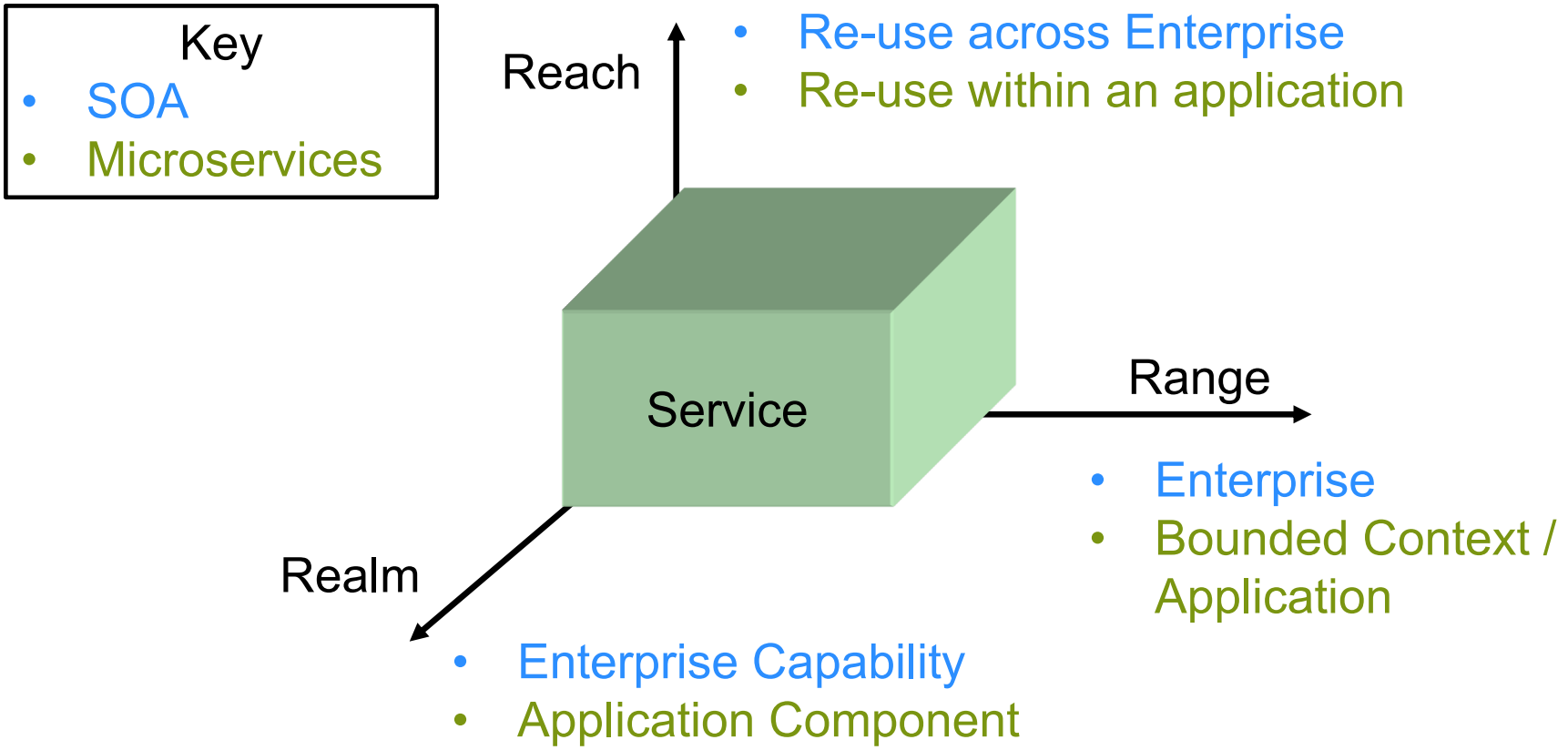
What is the size of a Small MicroService ?

Service size is not specified by SOA but is a result of architectural QOS balance, e.g. Granular, 'larger' services preserve bandwidth, reduce chatter but provide higher abstraction. Example, *addCustomerAddress, is* a *fine-grained* service, whereas *addCustomer is coarse-grained* .





SOA Vs Microservice - Realm, Range and Reach



Static Contracts

As a Provider of a Service we can give our consumers a contract of what we expose:

- A description of the business capability, the semantics;
- Interfaces, the syntax;
- Policy, more semantics;
- Schemas;
- Assumed NFRs as a SLA (as distinct to our SLOs).

These contracts are Closed, Complete, Bounded, Immutable for the time of a contract.

- Versioning attempts to allow for change and maintain contracts; but versioning is expensive, and becomes a bottleneck;
- How do we really know what consumers need and expect, and how do we test these hypotheses given the need to change, reducing the cost of maintaining version ?

Consumer Driven Contracts supports CD

Each Service Consumer supplies the Provider with an expectation of the service as test cases. The user EXPLICITLY provides what's to be tested.

The Provider manages a test suite across all the consumers.

- Allows Provider to make changes independent of Consumer;
- As a result providers can automatically check when change violate the contract with a consumer;
- **E2E Integration testing is no longer required, as each integration point can be tested and deployed independently.**
- Reduces the issue with versioning;

Issue: is it is up to consumers to understand their needs and requirements and be able to communicate this to the provider via quality test cases.

Service are not designed up front for re-use, guessing the wants of potential consumers. Real consumers drive contract generalisation, iteratively, based on specific test cases.

Microservices introduce new Issues

- When services fail how do we stop those faults cascading through the system.
- Processes will come and go and change their locations, e.g. with faults. How do we determine the locations of the processes that my process needs to collaborate with?
- When processes come and go, and we know the locations, how do we choose which process instance to communicate with next?
- In the span of time between choosing a process instance and communicating with that instance, the instance suffers a fault. How do we prevent that fault from cascading into a failure?
- How do we maintain visibility into the composite behavior of the system that emerges from the evolving topology of autonomous services so that we can make adjustments?

Example Tactics - Stability Patterns

These patterns address the ‘Fallacies of Distributed Systems’ by preventing cascading failures:

Prototypes and GUIDs;

- Timeouts;
- Retries;
- Gateway;
- Circuit Breaker - detects failures and prevents the application from trying to perform an action that is doomed to fail;
- Client Side Load Balancer;
- Handshaking;
- Bulkheads – isolate faults so they don’t take down whole system;
- Non-Blocking Asynchronous IO, Futures, Promises;
- Supervisor Monitor.

12 Factor Application Principles

1. Codebase – One codebase tracked in revision control, many deploys;
2. Dependencies – Explicitly declare and isolate dependencies;
3. Configuration – Store configurations in the environment;
4. Backing Services – Treat backing services as attached resources;
5. Build, release, run – Strictly separate build and run stages;
6. Processes – Execute the app as one or more stateless processes;
7. Port binding – Export services via port binding;
8. Concurrency – Scale out via the process model;
9. Disposability – Maximize robustness with fast startup and graceful shutdown;
10. Dev/Prod Parity – Keep development, staging, and production as similar as possible
11. Logs – Treat logs as event streams;
12. Admin processes – Run admin/management tasks as one-off processes;

https://12factor.net

Evolutionary Architecture

Major Points

Complex Systems have Unknown Unknowns.

- And so there is no predictable path from the current state to a better state.
- Architect for change;
- Focus on Skinny systems;
- Evolve architecture continuously – test quality, see its ‘fitness’ improve;
- Manage Systems not projects;
- Requirements need to be ‘right’, that is provide acceptance tests early on;
- The Qualities provide the objective fitness function for each evolution.
- Architecture builds out a runway that allows development to continue, unimpeded; the size of the runway depends on the landscape;

Fight Stability – Provide Flexibility

Should system stability be the goal, fighting change,

OR

Foster (accept) change and provide the structure (architecture) and processes to support it

In reality architectural change is permanent; new technologies, new products, and business change is continual.

- So rather than try to create stable architectures create ones that are flexible, agile, can evolve support continuous change.
- This applies to trying to make stable reliable platforms, its not possible;
- **Up front** make applications and systems flexible:
  - This requires a new priority for architecture.
  - Emphasis on qualities that are not visible to stakeholders.

Continuously Evolve the Architecture

- Continually evolve and refine an architecture; incrementally build and improve, delivering transition architectures, improving fitness, while:
  - uncovering and addressing architectural issues during development.
  - work ‘just in time’, build out just enough runway;
    - Analyse, measure and document architectural decisions;
    - validate those decisions;
    - ensure they are well communicated.
- Implies work on product systems (continual flow) not projects (interrupted flow);
- Sometimes a ‘Skinny System’ is necessary up front – but then let it evolve. The focus is only on what is Architecturally Significant.
- Evolution is directed by a ‘fitness function’ that is defined by systemic qualities that are objectively testable.
  - Architectural choices are evaluated (measured) against the fitness function.
  - As systems evolves fitness improves.

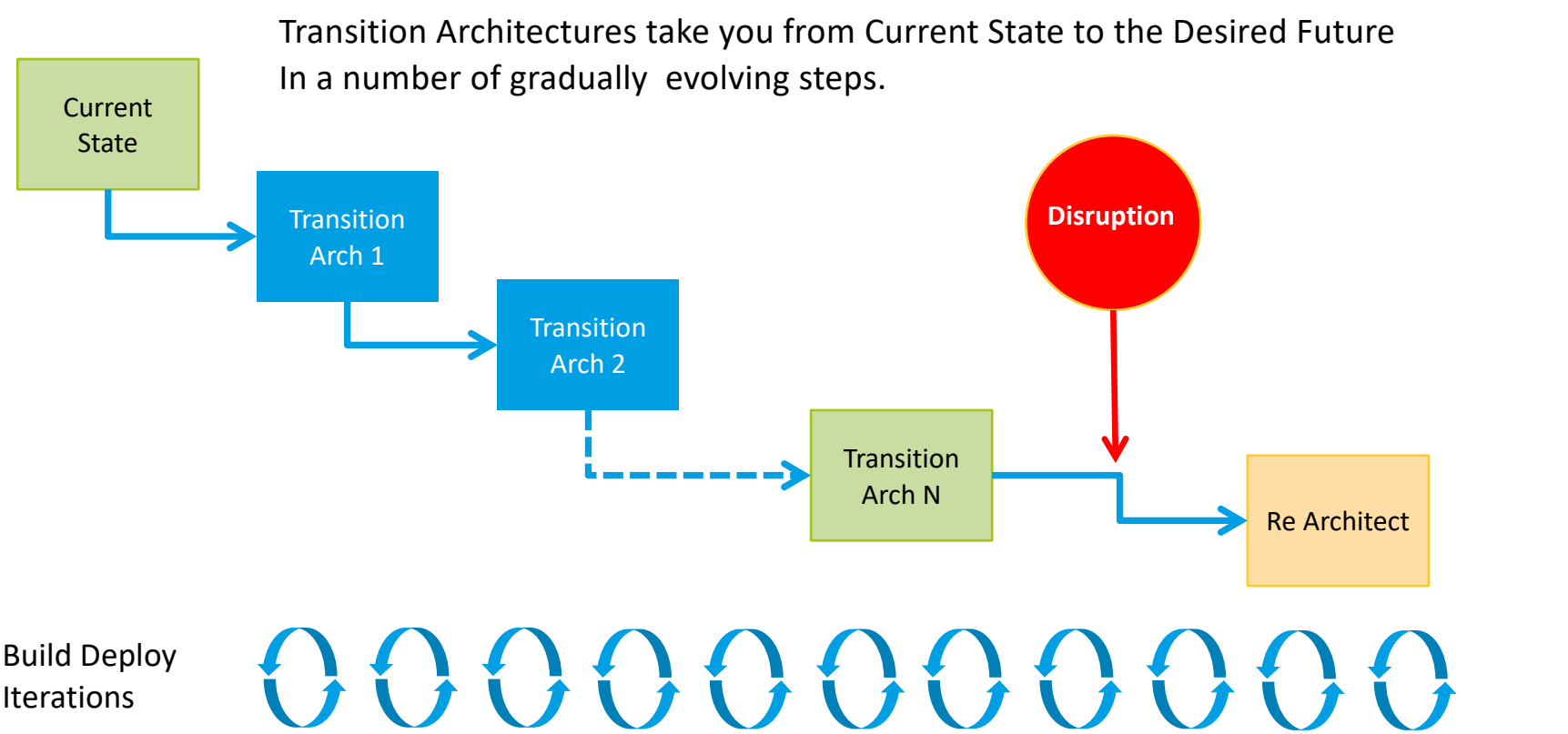
Continuously Architect or Re-Architect

As Architectural change is permanently happening, architecture is a continual process; the architecture evolves, its fitness evolving to meet customer needs.

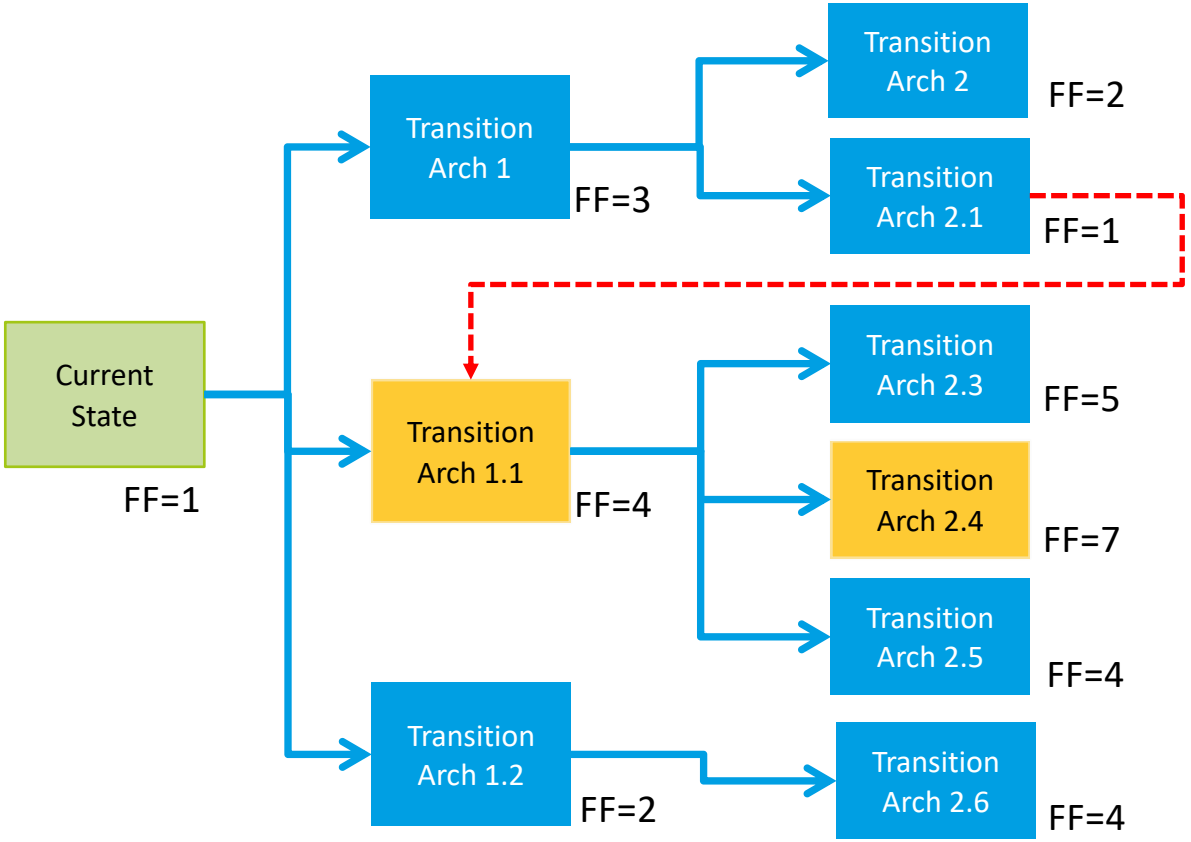
However, **the world is unstable;** disruptive change cannot be predicted, and making systems open to be evolved continuously or truly flexible has limits:

- Making a system endlessly flexible is very expensive;
- The reality is that all organisations should consider re-architecting from scratch every few years:
  - Amazon, Twitter, eBay have all gone through a series of major changes in their core architectures.
  - Each change was disruptive in the context of the prior architecture.
  - Be open to major disruptive re-architecture of systems;

Deliver a Series of Transitions Until you need to Re-Architect



Many ways to Evolve – what is the best path ?



Choose the path that maximises the fitness function (FF).

Some decisions may take you backward and deliver poorer quality down the line. You may need to backtrack.

Realise you cannot see too far into the future and so its greedy search – take the next path that improves things.

FF may also change over time.

Evolutionary Architecture

- **Improves quality and productivity** by reducing the need to make time-consuming, error-prone fixes to late-detected problems that result from architectural flaws. This is possible because the architecture is validated early, and key architectural problems are corrected before the majority of development is done. Architectural runway precedes development;

- **Reduces time to market** by focusing on reuse. It improves the consistency and maintainability of the system by incorporating lessons learned from development back into the architecture and applying those lessons to the rest of development.

- **Improves predictability** by identifying and implementing the highest-risk technical areas first. It improves the team's responsiveness to change by shortening the architectural cycle and minimizing time wasted in architectural scrap and rework when changes arise.

- **Reduces technical risk** without requiring significant up-front architectural effort.

Remove Architecture Change Impediments

- Teams should be able to autonomously manage their architecture, need to remove top down hierarchical management as this introduces delays and compromises, and is driven by lack of trust;
- Reduce decisions going through a centralised Architecture Review Board (ARB);
- Instead Architecture is driven by well formulated, documented and rationalised Principles;
- Move Architecture Governance to the teams.
- Lean architecture minimally documents architectural decisions, and how they relate to principles;

- Only significant architecture change that violate principles needs to go through ARB;
- All new solutions should check list the principles. If check list is all ticked then solution just goes ahead;

- ARB deals with change of principles, not solutions.

Evolutionary Architecture Practice Principles

Perform architecture work "just in time", but build out the runway, identify and discuss architectural issues with your team, and then prioritize architectural work with any other work.

- Identity the fitness function early, it too will change over time;
- Refine the architecture incrementally from a base (current) definition;
- Defer architectural issues to handle them "just in time" enables the architecture to *evolve over time*. Decide at the 'Last Responsible Moment'; leave options open;
- Base your initial priorities on mitigating technical risk rather than creating value. Provides an initial **Skinny system architecture** definition that is then refined;

Document key architectural decisions and outstanding issues (technical debt). This is key knowledge. Base decision on improving the fitness function. Decisions are a lean minimum documentation artefact.

Implement and test key capabilities as a way to address architectural issues. Resolving architectural issues typically requires not only architectural brainstorming, but also associated prototyping.

- Prototypes (Spikes) are small throw array builds that validate the assumptions behind the architecture. Resist the urge to productise prototypes;
- Measureable progress against fitness function (quality).

The Right kind of Requirements for CI and Evolutionary Approaches

Functional Requirements

Traditional 'The system shall….' business requirements are not good enough. Approaches that derive tests early on are required, and include:

- Use Cases – these can drive, trace to, and manage tests. These can also group User Stories.
- User Stories – these accompanied with Acceptance Tests also drive testing;
- The business can check when requirements are delivered when their tests or acceptance criteria are met;
- Requirements that are matched with automated test can be used immediately in Evolutionary and Incremental development.

Non – Functional Requirements

- Again generic, untestable or un-measureable statements are just not good enough;
- These requirements need to be stated in a way that are objective and can be tested automatically.
- NFR Quality Scenarios provide specific testable statements of these requirements, that can be measured and tested, to allow the systems fitness to be gauged at any time.

Continuous Architecture* 'Book' Principles

1. Architect Products – not Projects;
2. Focus on Quality Attributes – not on Functional Requirements;
3. Delay design decisions until they are absolutely necessary;
4. Architect for Change – Leverage "The Power of Small";
5. Architect for Build, Test and Deploy;
6. Model the organization of your teams after the design of the system you are working on;

by Pierre Pureur, Murat Erder

* An alternate view to Evolutionary Architecture, that supports the basic premises

Document Decisions

"In general, 'the code is the design' is a good rule of thumb.

- But its not a law or proven principle.

- "Patterns were created out of an understanding that code sometimes does not stand alone."
- "Code won't reveal everything about how a system will work"
- "we go to the code for *what* and *how*, but only the authors or their documentation tell us *why*"
- "Good Documentation is the roadmap that provides context".

"Lean Architecture" – Coplien

Culture People Process

"Everybody, all together, from early on"

"Lean Architecture" – Coplien

Major Points

- Remove bottlenecks (organisational and structural);
- Autonomy, mastery, purpose;
- Process will change over time, there is no one right method/process;
- Practices over Process; where the team owns its practices;
- Steams of product work over Projects;
- Inverse Conways Law - Organise team structured by the bounded contexts and architecture, team owns a complete context, not component;
- Coupling and Cohesion applies to Teams.
  - Use the architecture to decouple teams, make them autonomous;
- **Do not scale teams** – they should be small, distributed and independent, based on a distributed 'functional' development streams.
- **Bounded Context** is used to create Autonomous Teams, and the relationships are made clear by a Context map.
- Innovate rather than Predict (manage); teams should fail well. Everyone Innovates

Remove Bottlenecks; Reduce Hierarchy and Shared Service

- Layers of hierarchical management and centralised governance result in bottlenecks.
  - An Architecture Review Board that 'Signs Off' work is often a bottleneck;
  - Decisions that have to flow up to CIO for 'Sign Off' create bottlenecks;
  - Knowledge at the higher layers often does not filter down, teams left in the dark;
  - Teams feel less empowered or less trusted to make decisions;
- Teams managing and operating Shared Enterprise resources become a bottleneck as they must plan and schedule **their** work, understand others requirements and allocate new resources. Rarely can they explain what they own, as its out of context:
  - Customers may need to put work on hold;
  - Customers are forced to compromise on what they get, its not what they need;
  - Customers may need to duplicate work items, e.g. special requirements and contracts to hand off to the central team;
  - The services team must also compromise their services across all customers;
- Shared services create dependencies to reusable assets; bottlenecks;
- Requirements whose delivery cannot be objectively tested, and requires human QA, creates bottlenecks;
- Hierarchical structures introduce dependencies.

Autonomy, Mastery, Purpose (David Pink)

Successful Teams and individuals work are driven by:

- ***Autonomy***, or the desire to be self-directed;
- ***Mastery***, or the itch to keep improving at something that's important to us;
- ***Purpose***, the sense that what we do produces something transcendent or serves something meaningful beyond than ourselves.

Small, autonomous teams, that self organise, that are the masters of their own fate, with a shared purpose aligned with their business. Innovation is primary to purpose.

To make teams autonomous requires the right architecture and practices.

Processes need to be Flexible and Evolve

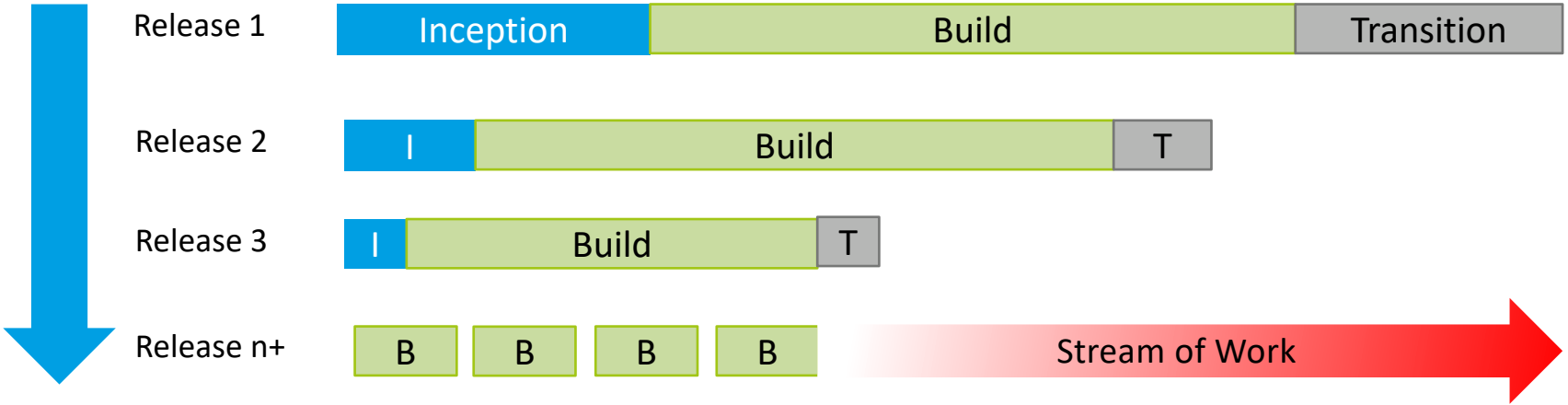
- Some work need to be more up-front, governed and formal, other can be more evolutionary and continuous;
- Important to balance ‘up-front’ Vs evolutionary architecture;
  - Emergent design Vs Intentional;
- The structure of work will change over time;
  - Initially they may be formal;
  - Over time less formality is required;
- The *‘Practices’* used, change over time, improve, mature, and fit better;
- Governance, projects, teams and portfolio management needs to adapt as the system matures, and the type of delivery changes;
- Projects create discontinuities and reduce ownership and accountability to the ‘product’, consider continuous streams of work owned by a small team.
- Team owns a full slice from UI to Infrastructure;
- Creating infrastructure or components teams introduces process dependencies; mismatches and bottlenecks;

However, some work can't be Continuous

- Some situations require up-front focus:
  - Programs with a dependency on large software or infrastructure purchases or choices;
  - Programs that require new platforms, complex product, technical research and investigation;
  - Greensfield projects;
  - Programs with complex security, vendor and delivery relationships;
- These projects require a more intentional up front architecture, that may drive the WBS into realiseable projects that evolve into streams of continuous work.
- **In these situation, building only the simplest architecture is not possible,** e.g. buying a product means you buy an established architecture with an often complex set of infrastructure, interfacing and operational concerns;
- As a result, a continuum, of program/project types is required.

Continuous Ways of Working

- Evolving Work Types – SDLC is not fixed but flexible;
- New Work requires a more traditional model, with some governance, controlled phases, and more upfront activity;
  - Some formality to get things going;
- With successive releases the phases start to disappear;
- Until finally work is shorter continuous stream based iterations.



Conway's Law and its Inverse

"Organisations which design systems .. Are constrained to produce designs which are copies of the communication structures of these organisations."

Conway

"Quality is strongly affected by organisation structure"

Brooks

Architecture provides a way around Conway's Law, by providing structures alternate to the organisation; this is **Inverse Conway's Law**.

- Goal is to make communication effective; not bureaucratic and hierarchical;
- Use decomposition around architecture to organise and structure distributed teams;
- **Use Bounded Context to enable Autonomy.**
- **Use Context Map to understand Autonomy.**

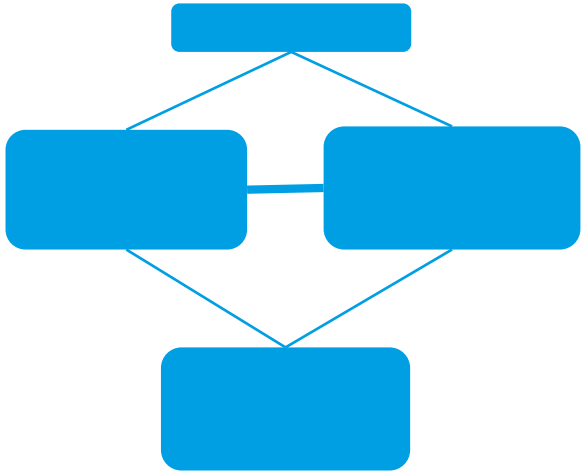
} Microservices

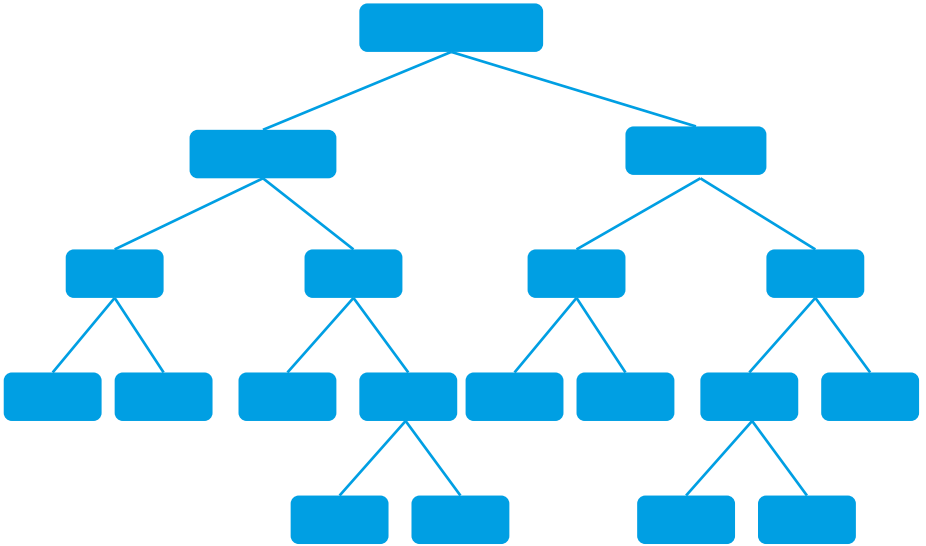
Bounded Contexts define Team Boundaries

"Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside."

Eric Evans

Scale with Independent Distributed Teams – not scaled processes

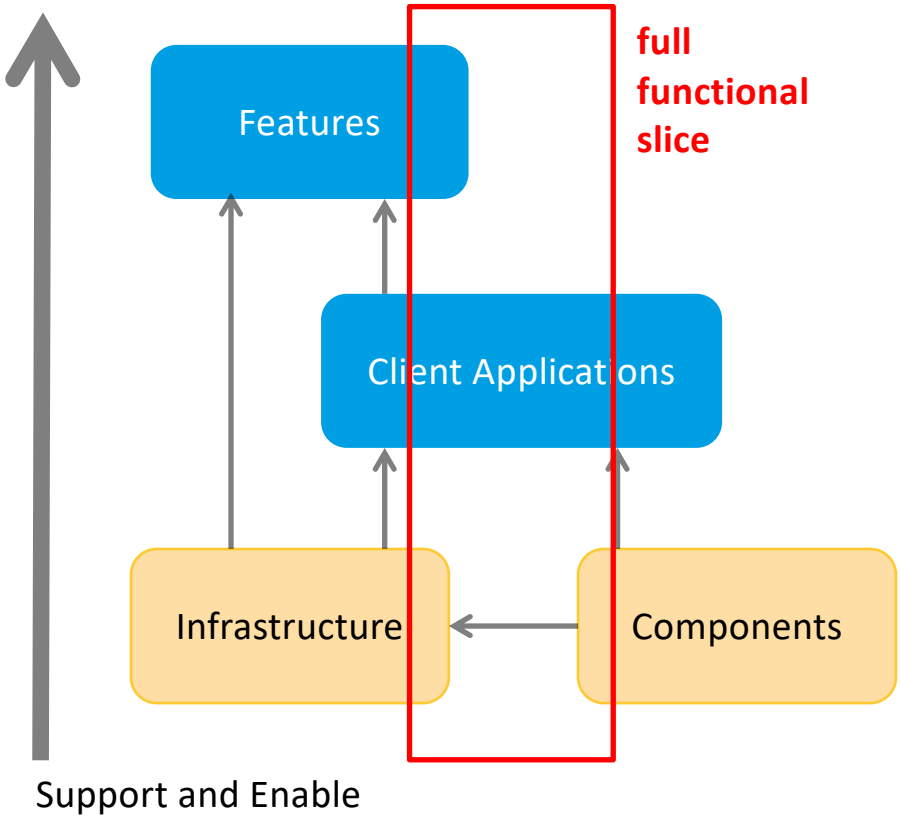




- Large Components;
- Large Teams – coupled;
- Complex Processes;
- More complex communications, more meeting, less work, Committee based decisions;
- Hard to Scale;
- Not Agile.

- Smaller Loosely Coupled Components;
- Smaller Independent Efficient Teams;
- Easier to Scale work;
- Teams more Agile;
- Teams use only Practices they need;
- No overall processes.

Example Teams Structure



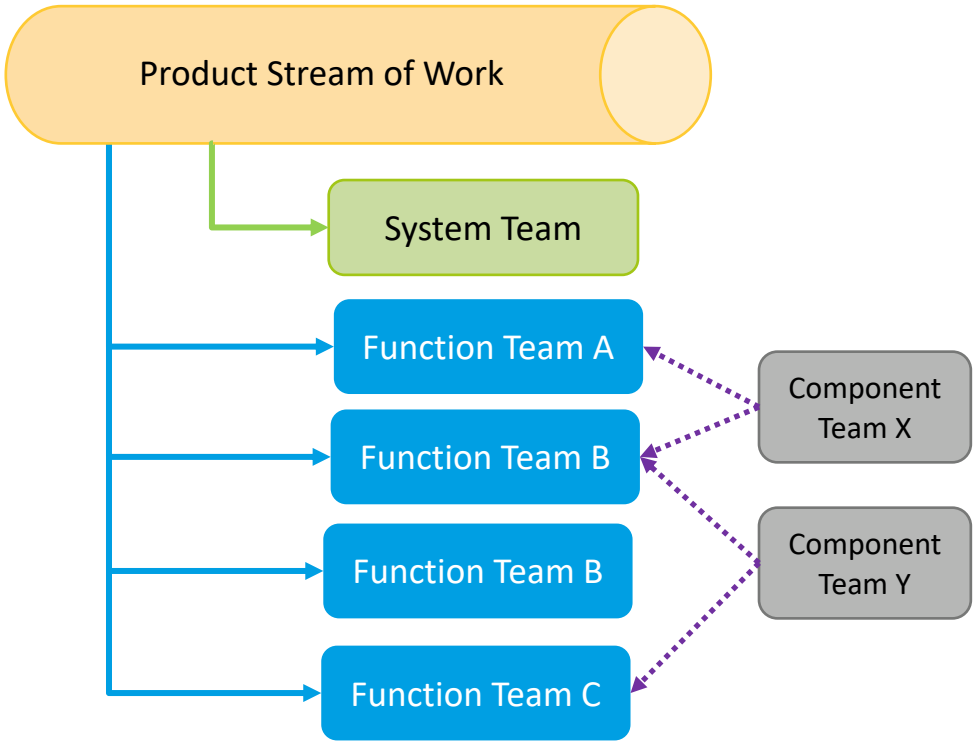
Spotify has demonstrated that teams can be effectively organised around:

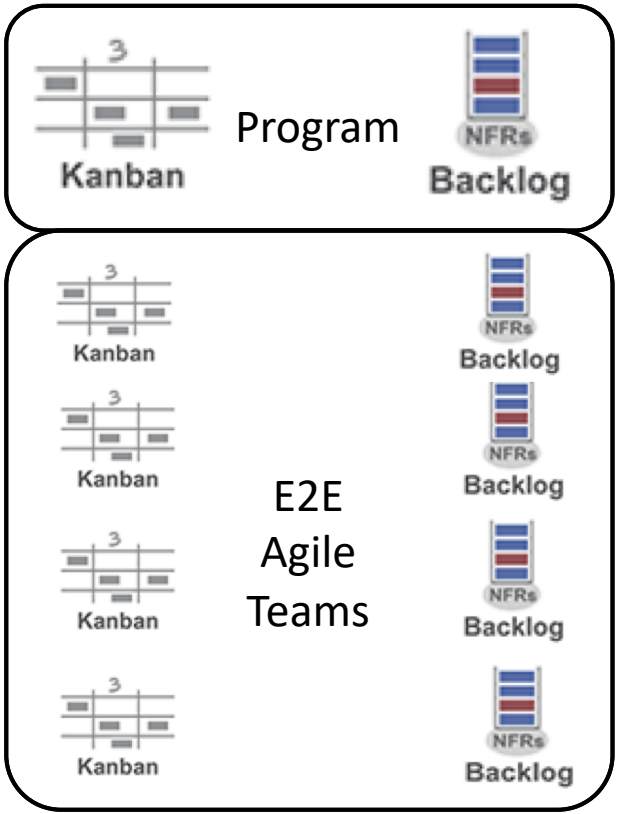
- **Features**
- **Client Applications**
  - enable and support Feature teams
- **Infrastructure**
  - enable and support feature and app teams

Amazon has also organised around:

- **components**
- **Recommended to organise around a full bounded context; own full functional slice, including infrastructure;**
- **Open source components, do not centralise;**

3 Kinds of Team – SAFe Model – E2E Agile Teams





Need to be aware of possible coupling when same team does full slice/stack

Architects

“One organization structure that we know does not work is to isolate all the architects in their own team”

Lean Architecture” – Coplien

Architects must be part of the delivery teams they wok with (in space and time*), and not separate. Everybody in the team must work together, not re-assembled for each project.

* Most communication happens informally, and decisions made on the fly, quick feedback channels.

Iterate – you can't predict the future, you won't get it right up front.

Up front Analysis will fail, as no one ever knows what they want, or they just can't articulate it well, or are just bad at predicting the future. Analysts will work with users but their biases get in the way and 'interpret' what's wanted, adding noise, and errors.

You need to test the market - running measured experiments.

Often customers don't know what they want until they use the software, the rubber hits the road, touch it, feel it, and in most cases its not what they wanted. They can 'reliably' describe what they don't like. It's an iterative knowledge discovery process.

Delivering Software requires the Scientific Method.

A released deliverable is a hypothesis about what's wanted, a best bet, test it experimentally. Iterate over:

1. **Build** and release something in a controlled experiment. E.g. A/B.
2. **Measure** how it was used, liked by the customer, measure process improvements.
3. **Change** the software, based on the evidence, go back to 1 with a new hypothesis to test.

You just can't get it right up front – humans don't work this way. Its impossible, so don't worry about it. BUT you must base change and improvements on scientific process/method.

Fail Fast - Everywhere

As we want software components to fail fast, we also want:

- Development to fail fast – automated tests fail early - to catch issues at build.
- Move Failure to the left.

However, don't stop there, we want human processes with teams that:

- Are small that can practically manage their destiny;
- Fully own and run small experiments;
- Experiments are tested in the real world using real customers;
- Experiments are well designed using valid statistical analysis;
- Are based on a culture of innovation, exploration and failure (teams not not fear trying things that may not work)
- Innovate as part of teams normal work cycle. Innovation is not centralized but distributed.

Finale

Steps to CI/CD Success

1. Architecture comes first - must have a way to deliver small independent components.
   - Monolithic applications won't get value from CI/CD;
   - Implement 'Share Nothing' Architecture;
2. Understand the team structure and social / dependency relationships between teams
   - Small teams, scale down not up, that are autonomous, masters and purposeful.
   - Organise around function and bounded context, not components
3. Master Iterative Ways Of Working:
   - Evolutionary Architecture – Principle and Quality driven;
   - Manage complexity with Bounded Contexts;
   - Consumer driven;
   - Short lived to no branching – add feature toggles;

NOW (and only now)

5. Introduce CI tools and process – automate tests and integrate when tests pass;
6. Move to Continuous Delivery - allow business to deliver to production;
7. Continuous Deployment – when tests pass deploy to production;
8. Measure and Adapt. – It's a Scientific Evidence Based Process.

"What Darwinism says is either evolve or get marginalized. Darwin doesn't care. At the end of the day you perform or you don't."

Moore, 2006

The End