Architecting for Continuous Delivery

Russ Barnett, Chief Architect
John Esser, Director of Engineering Productivity

Ancestry.com

Background

Growth at Ancestry.com

- Subscribers have doubled in the past 3 years
  - (~1M to > 2M)
- Page views have doubled in the past 3 years
  - (~25M/day to ~50M/day)
- Development head count has tripled
  - (100 to 300)
- Feature throughput has dramatically increased

Continuous Delivery

is consistently and reliably

releasing business value increments

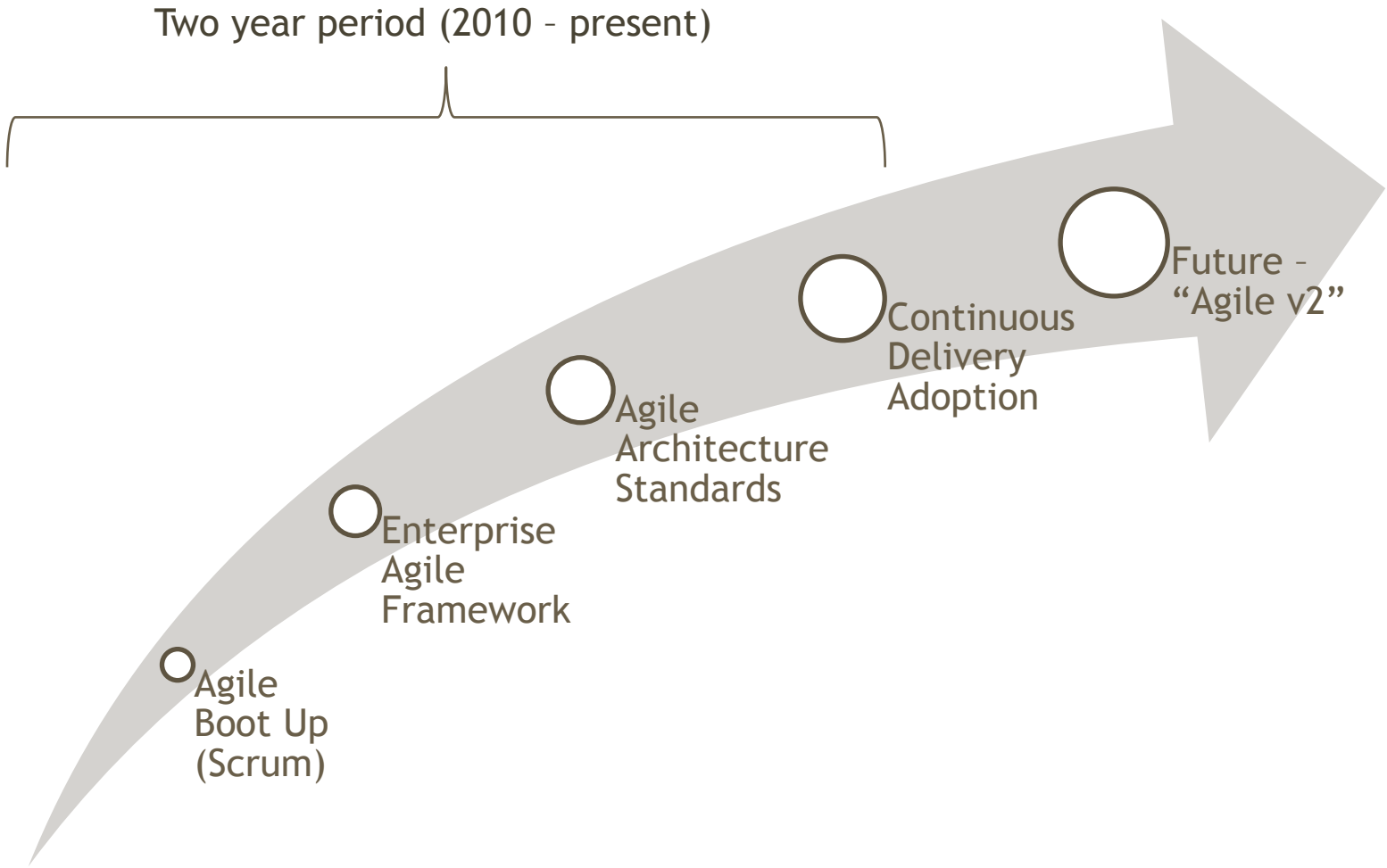
fast

through automated build, test, configuration and deployment.

What is the value of going fast?

A LOT!

Evolution to Continuous Delivery

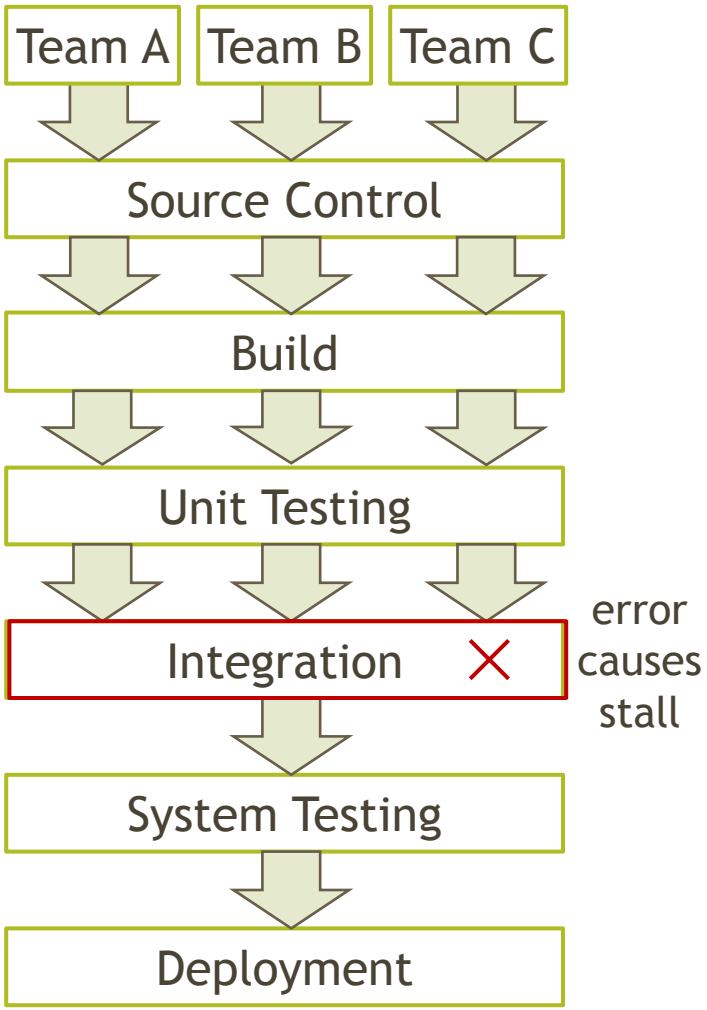


Typical Impediments to Continuous Delivery

- Cultural
- Technical practices
- Quality ownership
- Infrastructure
- **Architectural**

Limiting Factors

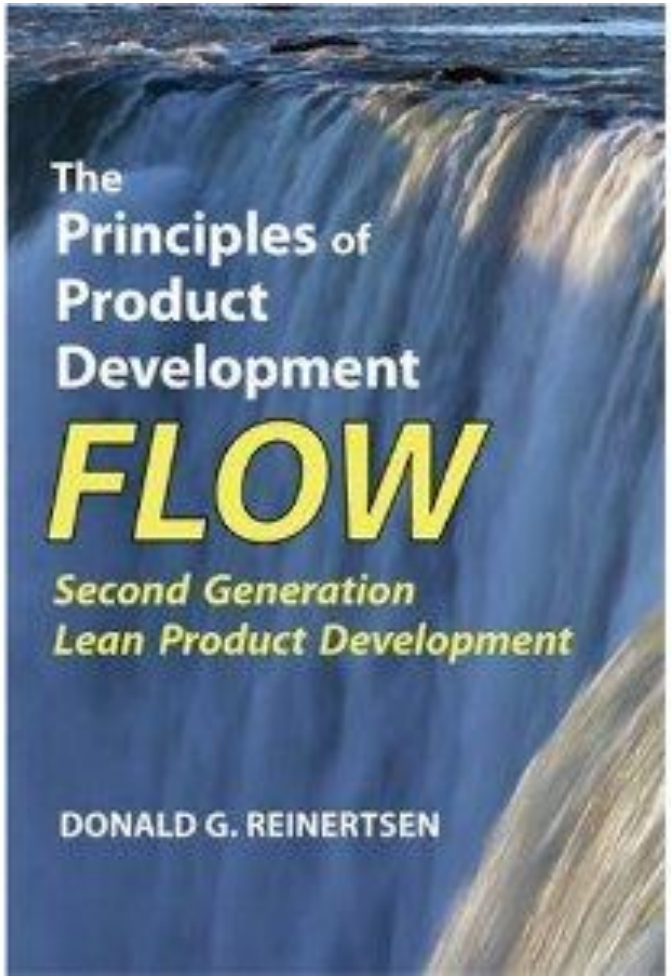
- Pipeline serialized at integration
  - Errors that occurred in this stage stalled the pipeline
- Stalls in integration induced additional problems
- Increasing frequency of stalls
  - As number of development teams grew, frequency of stalls increased



Everything was coupled!

(Aka, large batch size)

(It became known as the “big blob!”)



Little's Law

$$Wait\ Time = \frac{Queue\ size}{Processing\ Rate}$$

$$Queue\ size \propto Batch\ size$$

We can reduce wait time (cycle time)
by reducing batch size
without changing demand or capacity.

Problems with Large Batches

- Increases cycle time
- Increases variability non-linearly as 2^n
- Increases risk
- Reduces efficiency
- Limited by its worst element.

from *Principles of Product Development Flow*, Don Reinertsen

Answer?

Utilize small batches.

Fluidity Principle

Loose coupling
between product systems
enables small batches

"Once a product developer realizes that small batches are desirable, they start adopting product architectures that permit work to flow in small, decoupled batches."

from *Principles of Product Development Flow*, Don Reinertsen

Fluidity Principle

Loose coupling
between product systems
enables small batches

"Once a product developer realizes that small batches are desirable, they start adopting product architectures that permit work to flow in small, decoupled batches."

from *Principles of Product Development Flow*, Don Reinertsen

Creating an Architecture for Agility

Architectural Impediments

- Cross-Component Coupling
  - Creates groups of systems that must be deployed together
- Insufficient Rollback Capability
  - Causes teams to resort to cascading rollback
- Poor Testing and Monitoring
  - Requires a long testing period
  - Lengthens feedback cycle
  - Allows quality problems to escape to and affect customers

Architectural Methods for Removing Impediments

- Partition into small single-responsibility components

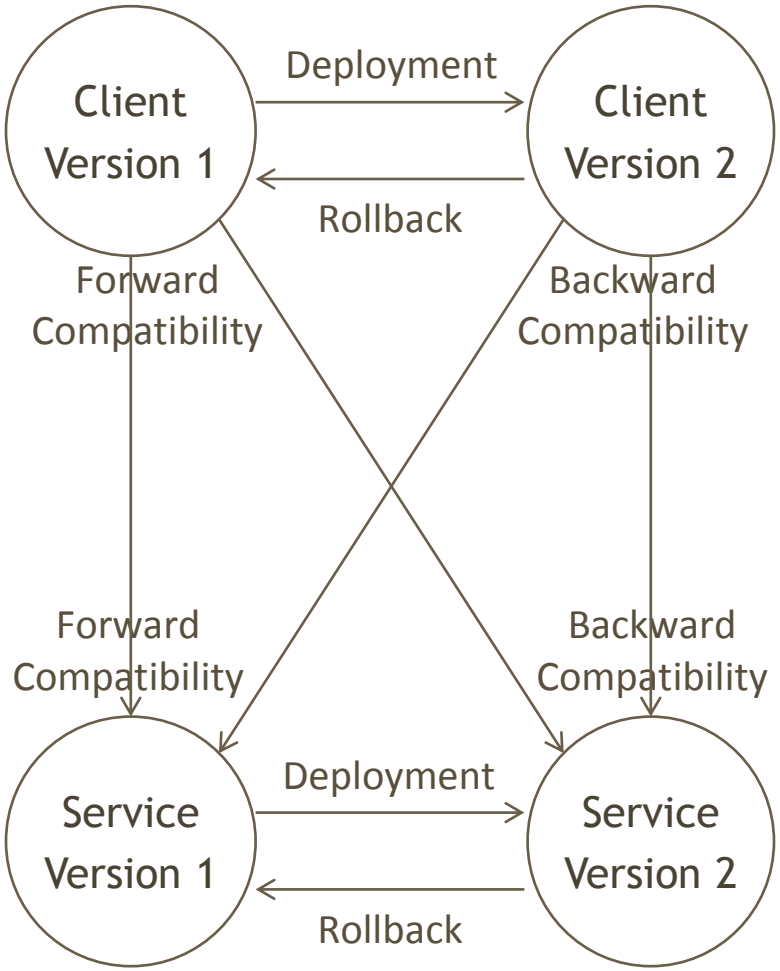
  "There should only be one reason for a [component] to change"

  - Robert Martin

- Decouple deployment of components
  - Separately deploy components
  - Remove order dependent deployment
- Support Independent Rollback
  - Enforce strict backward and forward compatibility

Backward and Forward Compatibility

- Server Backward Compatibility
  - Newer servers work with clients written to old interface
- Server Forward Compatibility
  - Existing servers work with clients written to newer interface
  - Supports early client deployment
- Client Backward Compatibility
  - Newer clients work with servers that implement old interface
  - Supports server rollback
- Client Forward Compatibility
  - Old clients work with servers that implement new interface

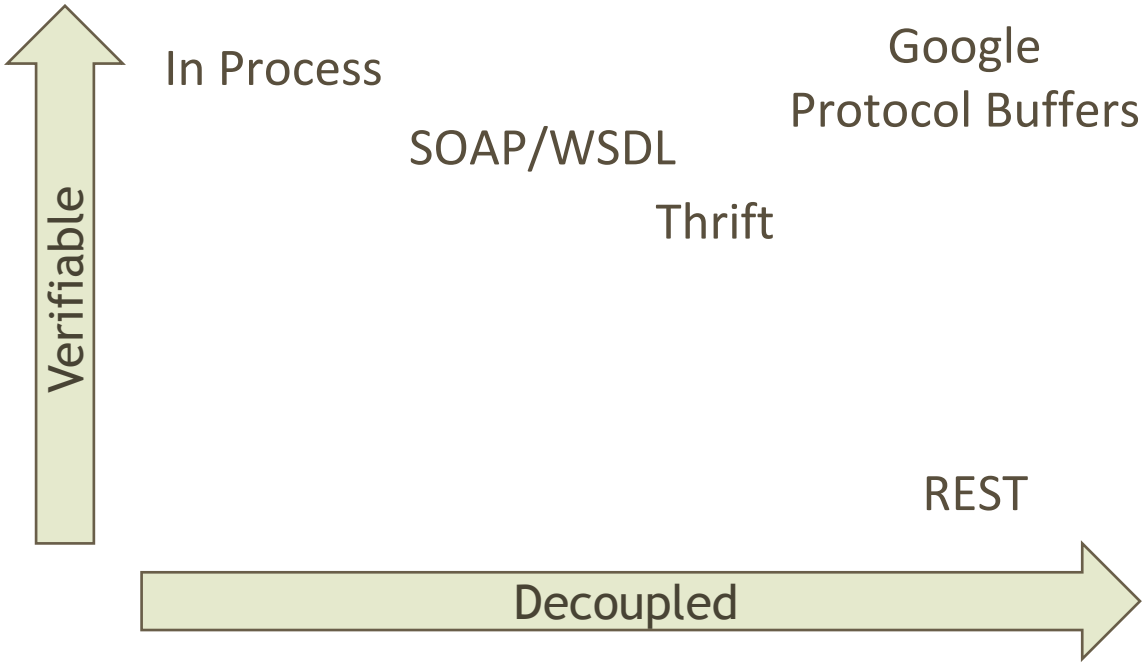


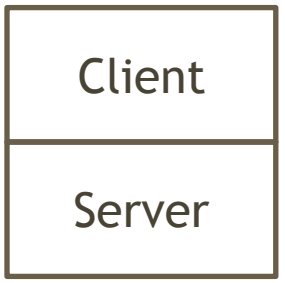
Enforcing Decoupled Components

- Implementing standards is insufficient
  - Independent deployment forces some decoupling
  - High rate of deployment issues indicate remaining coupling
- Improve integration testing
  - Verify backward and forward compatibility
  - Identify breaking changes quickly
  - Make writing integration tests easier

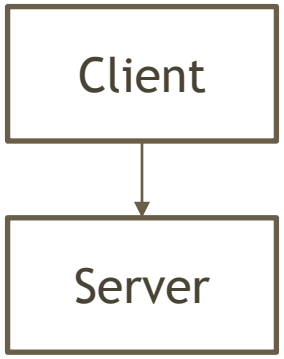
Improving Interface Verification

- Remember when you could run your entire application in one process?





In Process

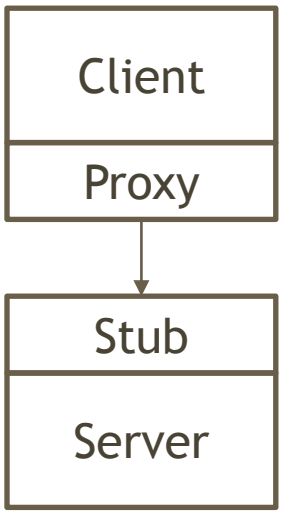


Client/Server

- How do we get better interface verification with services?

Interface Verification using Proxies and Stubs

- Verifies interface at compile time
- Isolates code from versioning issues
- Easier to provide mock implementations
- Can test backward and forward compatibility

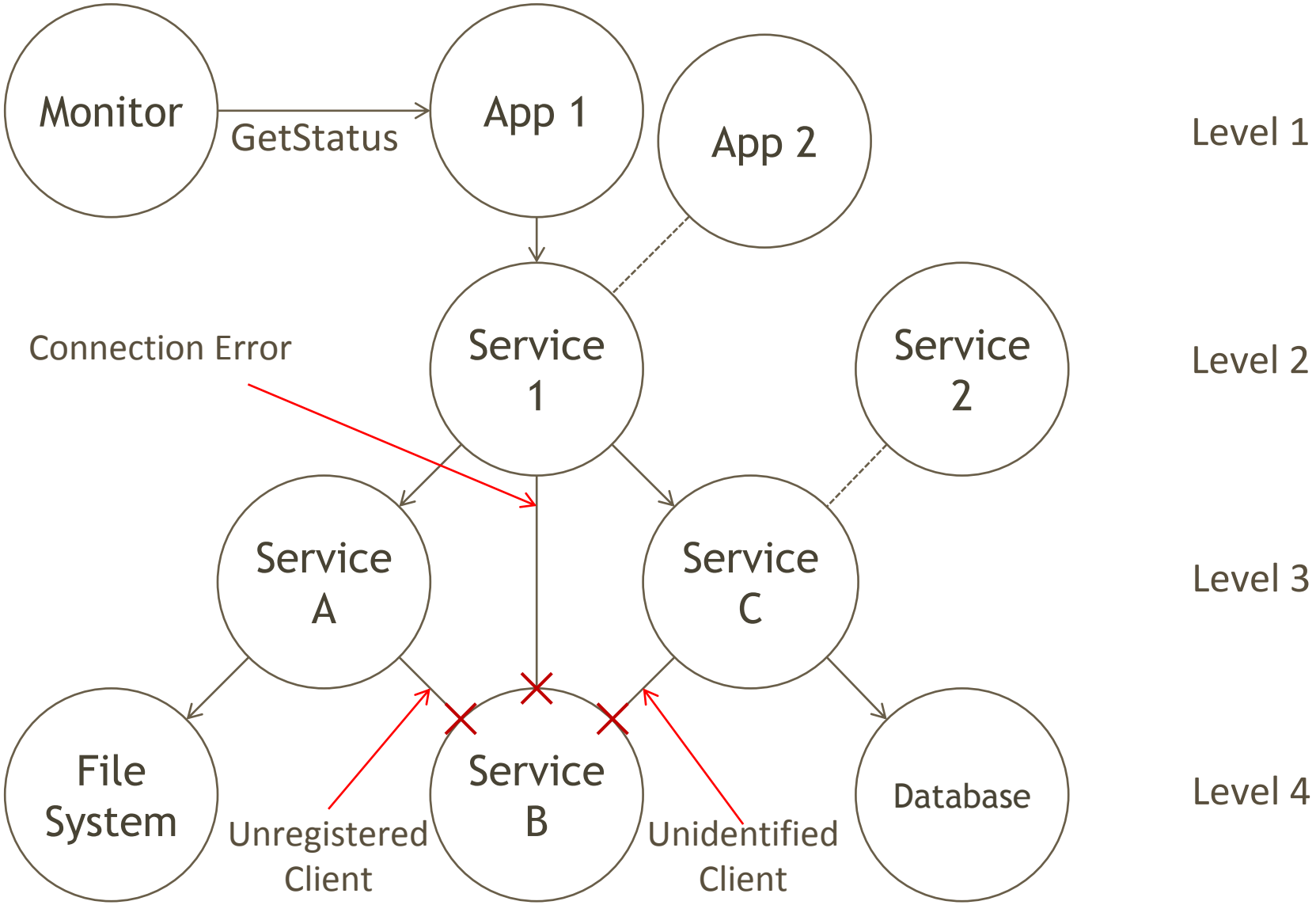


Client/Server with Proxy/Stub

Managing a Complex Network of Services

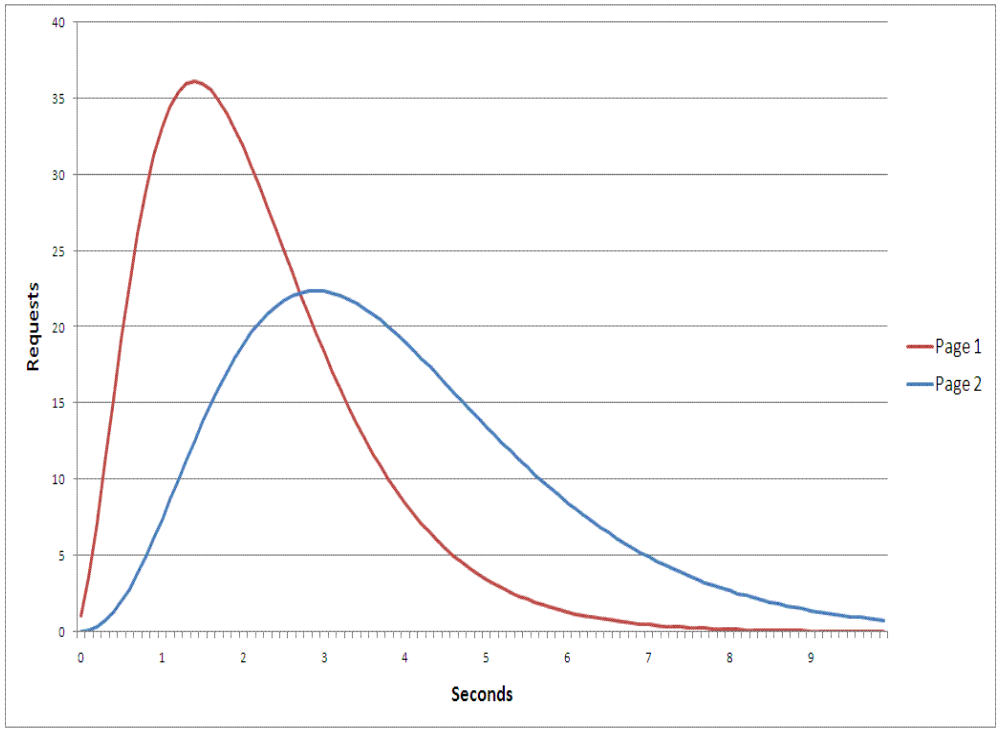
- Ancestry Scale
  - About 40 different teams
  - Over 300 separate application or service systems
  - Stack is 5+ levels deep
- Historical Diagnostics
  - Presented a client centric or top down view
  - Insufficient for identifying problems in a *network* of services
- Solution: Deep Status Check
  - Components provide dynamic status information for each client and dependency
  - Report traverses dependencies up to a given depth

Ancestry Deep Status Check

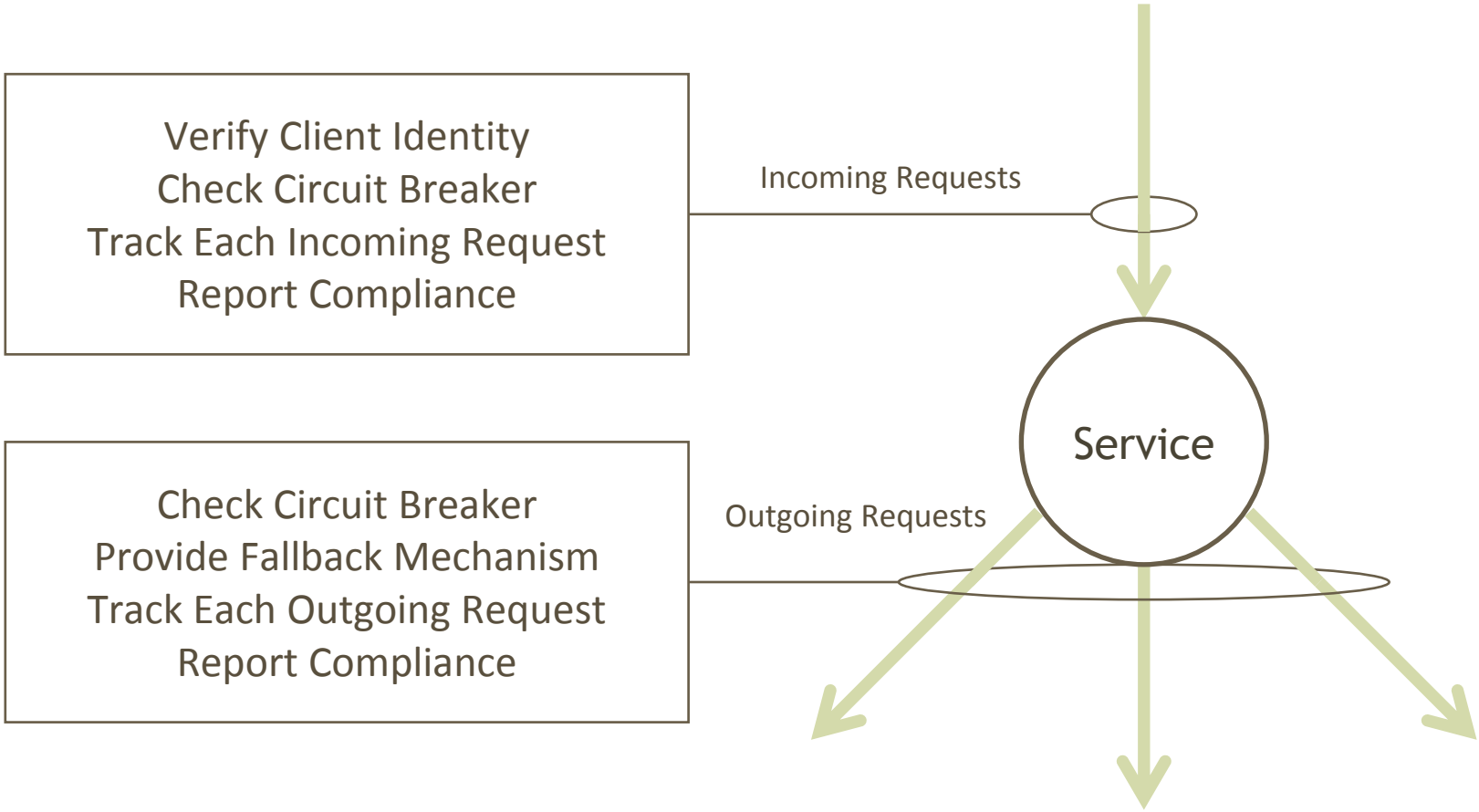


Service Level Agreements

- Understand Business Expectations
  - Each application and service establishes a contract with each client specifying the expected performance characteristics
- Methodology
  - Use percentile buckets rather than an average
  - Performance is a component of availability



Service Level Agreement System



- Compare incoming and outgoing SLA compliance

Conclusion

- Architecture affects agility and continuous delivery capability as much or more than other factors.
- Process and tool improvements alone are insufficient.
- Good architecture techniques enable effective continuous delivery at large scale.
  - Partition to single-responsibility components.
  - Decouple deployment
  - Support independent rollback
  - Improve testing and monitoring infrastructure

- Questions?

Contact info:

jesser@ancestry.com.

rbarnett@ancestry.com.

Ancestry is hiring in San Francisco and Utah