Scalable Applications

How to survive a tsunami of users

What is an application?

Anything that contains non trivial logic and faces directly or indirectly your customers.

Scaling requires

Paradigm change

GILT

Top 50 Internet — Retail
~150 Engineers

Started on 2007, Run on a Monolith till 2009

Big growth spikes, service failures, titanic imminent
Monolith be gone

Cloud & Agility

Commodity ==

Something anyone can buy ==

Not a competitive advantage

Vertical scaling

Old beast vs new beast

Old beast is slow to innovate

Old Mantra

Build perfection

New Mantra

Disregard perfection, publish fast, iterate fast

Problem

Our software is still following the old paradigm

Our programmers are largely following the old paradigm

Monoliths

Huge chunks of rock - try moving them forward

Adding new features becomes increasingly difficult with time

So does maintenance

..and testing

Do not think in boulders

or you’ll end up with a Monolith

Do not think of rocks

you'll still end up with a monolith.

Rocks are not different that boulders because they too don't have the key property you need

CHANGE

as in “to change”, not pennies

Think wet clay.

Evolving Systems

Changes must be able to become incorporated fast, robustly and cheaply

Simplicity

Small

Single purpose

How small?

Enough to allow change and enable evolution

Polyglot

If it catches mice, I don’t care what its color is

Pull talent from more pools

No learning curve

Natural Selection

Integral part of innovation

Competition between implementations - A/B

You want to be able to do that fast and cheaply

Structure

Not building hierarchies of abstractions

Structure

We are building pipelines of data-flow

Matrix of independent mini-services

Scale a Monolith

Hard

Very hard

Painful

Very painful

Scale LCS

Scaling Lightly Coupled Services can be a system condition

Don’t even have to be handled by you

Scaling Efficiency

Linear scale

Linear cost

You need both

Failover

LCS is build with one key principle idea - it should be ok to fail at any time.

Failover baked in by design - major win

Fragile or Robust?

Fragile -> Blue Screen of Death

Fragile or Robust?

Robustness stems from many layers of wrapping and abstracting

Hard to change or worse … resistant to change

Anti-fragile

T-1000 -> Super easy to change, hard to bring down

Handles run-time conditions better, adapts to stress, gets better

Anti-fragile services

Release early - release fast

No fear of breaking things

Autonomy of parts

Clash of teams gone

Dependencies of code leads to dependencies of people

Core Services

Storage

Messaging

Search

Multi-tenancy

Build once

Use again and again (and again)

Service versioning

Allows for transparent upgrade of functionality

Applications

Thin terminal

Build upon layers of services

Data Ecosystem

Producers

Info

Consumers

Pub/Sub

Girl walks into a bar…

Inactive services wake up to serve in response to stimulus

Focus is on Events - not Entities

Message Bus

Information must be able to flow from one place to another

All your services take the same bus - major win

Discoverability

I want to buy a pokemon, who’s selling?

Usually hardcoded - bad

Scalability & Discoverability

Surprise - this is yet another service

DNS
Web API
Multi-datacenter

Topology

Pipelines with redundancy

Distribution

Everything virtual

Docker orchestration

Quality Assurance

Logging

Tests

Monitoring via Aggregation