Computer systems are broken from top to bottom, yet ...we speak of craftsmanship.
The many faces of

Theo Schlossnagle

@postwait
CEO Circonus
The nature of the problem

Software Sucks

Once you’ve run software at scale, you have a deep understanding of how it is all tied together with loose string and hope.
To write good software is exceptionally hard

- Specifications are hard to write
- Projects can be long
  - requirements change
  - stakeholders change
  - resources change
- Environments change
- People think laziness is a virtue
Technical debt is non-linear.

Large projects have more code, higher learning curve, develop dangerous inertia.
Large monoliths are more likely to fail.

Rule \( \lambda_1 \)

Big projects have more risk.
An existing tool should be used instead of a new tool.

Strawman .h1

or you’ll end up with a sprawling, unmaintainable architecture

in languages no one knows
Diversity is both pre-existing and emergent.

Diversity

- 30 git repos
- 30 fixes to open-source (external) projects per month
- 6 programming languages
- 4 database technologies
- 2 networking vendors
- 3 server vendors
- 3 operating systems

http://www.bonkersworld.net/building-software/
It happens in the cloud just the same.

Diversity

- 30 git repos
- 30 fixes to open-source (external) projects per month
- 6 programming languages
- 4 database technologies
- 2 networking vendors
- 3 server vendors
- 3 operating systems
- 4 cloud “services” (RDS, ElasticCache, etc.)
Diversity is an emergent property of scale.

Rule $\lambda_2$

Different languages & different architectures exist because of different problems.
Engineers are a weird folk.

Engineers

Different temporal universe.
World is flat mentality.
Interesting rituals.
Learn

❖ Learn to hit a fucking deadline
  Deadlines are arbitrary things...
  so is your product, your selected solution, & your job.

❖ Learn to do a fucking paper search
  Masturbation might be fun...
  not at work, not with your colleagues.

❖ Learn to fucking balance
  Interruptions that derail your work may just add more
  value to the team than they cost you.
Engineers function better when autonomous.

Rule $\lambda 3$

Not all perhaps, just the ones you want to hire.

Autonomy of approach, not purpose.
Society in toto

Social Contract

open source projects
companies
teams
The social contracts of software

APIs

Application Programming Interfaces set expectations between any two components in an architecture.

Life is simple when expectations are simple.

Strive to minimize the surface area.
You shall be judged by your word API; be terse.

**Rule .\text{\textlambda}4**

Society is held together by a social contract;
software architectures by an interface contract.
Components shall be right sized.

Corollary $\gamma_1$

- Too large violates $\lambda_1$
- Just right leverages $\lambda_3$
- Too small violates $\lambda_4$
Data storage backend replacement

Anecdote \(a1\)

From PostgreSQL
To a custom time series DB
**APIs are about minimizing surface area.**

**SQL is not an API**

SQL’s surface area is so large it is daunting to replace.
Storage growth on our PostgreSQL clusters

Out of control

Three events of removing retained data
Snowth design

- Need: zero-downtime
- Know: Agreement is hard.
- Know: Consensus is expensive.
- CAP theorem tradeoffs suck.
- CRDT (Commutative Replicated Data Type)
Availability Zone 1

Availability Zone 2

Availability Zone 2

Availability Zone 1

Zone 1

Zone 2

n1-1
n1-2
n1-4
n3-1
n3-2
n3-4
n5-1
n5-2
n5-3
n5-4
n1-3
n3-3
n5-3
n6-2
n6-1
n6-3
n2-1
n2-2
n2-3
n4-1
n4-2
n4-3
n4-4

o1
6 nodes, 85 vnodes per node.

A real ring

Keep it simple, stupid.
We actually don’t do split AZ
Storage growth on our Snowth cluster

Rethinking it all
Production deployment overlap of ~12 months

**Time & Safety**

Small, well-defined API allowed for low-maintenance, concurrent operations and ongoing development.
Snowth has its own issues.

Craftsmanship

Where it was needed most.

And even there...

Snowth is composed of several subsystems that allow for different levels of scrutiny
Message queue replacements

Anecdote \(a2\)

From RabbitMQ
To Fq
A love affair with the rabbit... let’s just be friends

RabbitMQ

RabbitMQ is awesome for certain things
not for everything.
Learning the hard way

Oops... we broke it

Once we passed 5-digit volumes, the world came crashing down.

Rabbit simply turned into an operational nightmare for us at > 60,000 messages/second

We thought it would do better, but we had tested with the wrong message sizes.
Replacing RabbitMQ is actually quite daunting.

AMQP is complex

Large surface area. Many features.

Finding a better “thing” is almost impossible.
Step back, rethink.

Separate

Separate requirements to isolate used surface area.

In this case, we separate the:

- control plane
- data plane
Leaving the control plane on RabbitMQ

RabbitMQ lived

Reduced message volumes stabilized our use.
Fucking Queues

Fq was born

F@#$*&%Q...

Fast, brokered, in-memory and on-disk

Doesn’t have XA transactions
Doesn’t have acknowledgments
Doesn’t have delivery confirmations

It’s really fast... for us.
Deeper Realizations

- Autonomy:
  - API deliberation
  - API openness
  - data serialization stability

- Right sizing components:
  - reduces maintenance
  - increases agility
  - legitimates rewrites
Thank you.