

The background image is a dark, moody photograph of a person wearing a dark, hooded cloak and a light-colored hat, standing on a stone path in a forest. The path is made of large, irregular stones and is flanked by dense foliage and trees. The lighting is low, creating a mysterious and somewhat somber atmosphere. The title text is overlaid on the left side of the image.

The Lost Tales of Platform Design

Wisdom from the Masters, learned the Hard Way

Julien Simon
julien@julien.org
@julsimon

Who am I ?

- Software Engineer
- 20+ years in R&D teams, from smartcards to web platforms
- VP Eng/CTO @ Digiplug, Pixmania, Criteo, Aldebaran Robotics, Viadeo
- Now working for Amazon Web Services
- AWS PR people: this talk is strictly personal. All views expressed are my own, which is why you will probably be offended :D

Agenda

- The Only Thing that Matters
- A Method to the Madness
- Infrastructure
- Architecture

- Code Hygiene
- Optimization
- The Bleeding Edge
- Summing things up

The only thing that matters
no, not drugs



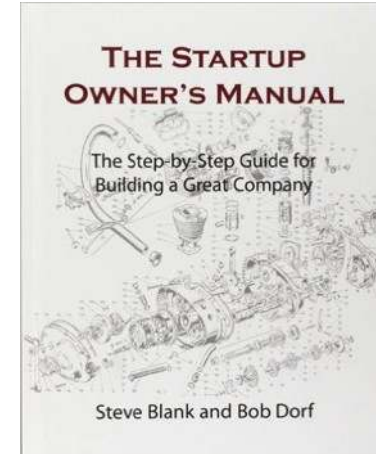
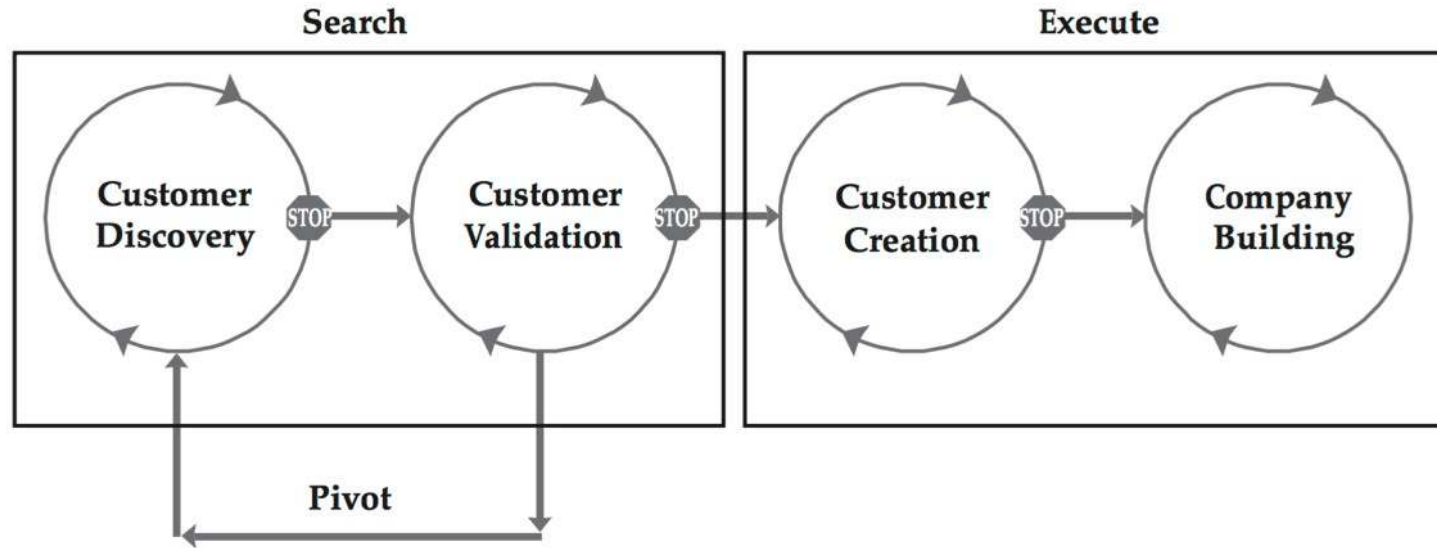
What Founders & Investors obsess about

*“Do whatever is required to get to **product/market fit**. Including changing out people, **rewriting your product**, **moving** into a different market, **telling customers no** when you don't want to, **telling customers yes** when you don't want to, raising that fourth round of highly dilutive venture capital - **whatever is required**. When you get right down to it, you can ignore almost everything else”*

Marc Andreessen, June 2007

<http://web.archive.org/web/20070701074943/http://blog.pmarca.com/2007/06/the-pmarca-gu-2.html>

The Startup Owner's Manual - Steve Blank & Bob Dorf (2012)



“I’m a dev, that’s not my problem”

EXCUSE ME?

Are you a pro, or just playing with computers?

What do you think you’re getting paid for?

What do you think happens when the company runs out of cash?

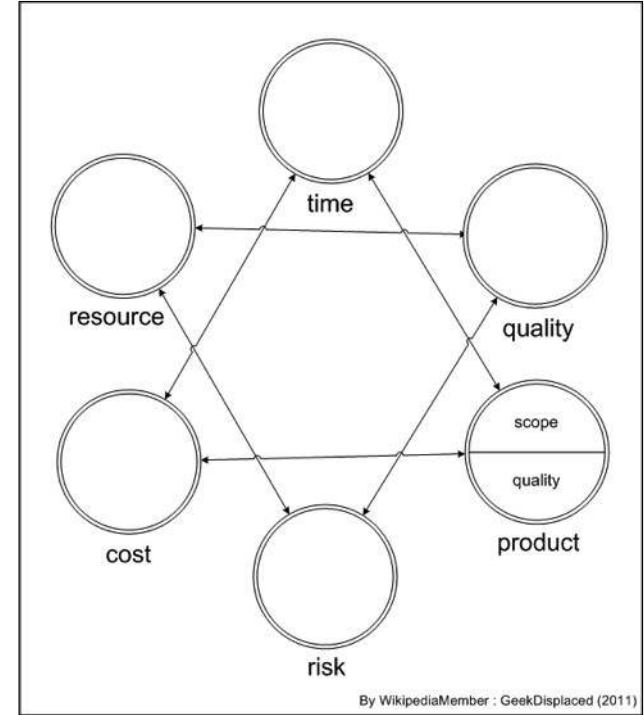
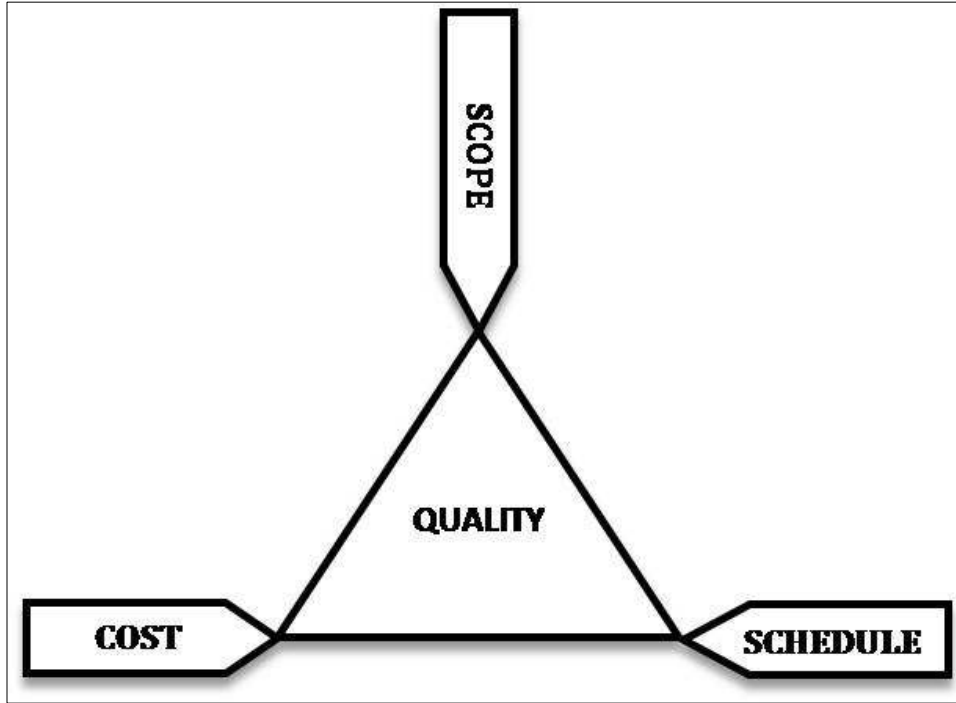
Do you want to work for shit companies or great ones?

A Method to the Madness



The Trilemma of Project Management

aka The Iron Triangle (Martin Barnes, 1969)



Less than 1/3 of projects are successful

and it's not getting better

MODERN RESOLUTION FOR ALL PROJECTS					
	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

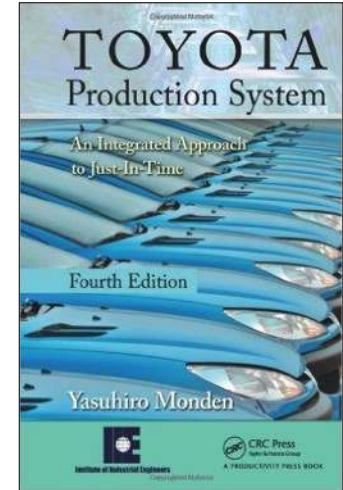
The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011-2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

Kaizen (1950)

改善

Reduce waste

Spend no money
Add no people
Add no space



Boyd's Law (1950)



- Boyd was a US Air Force Colonel, trying to understand why the highly superior MIG-15 was **consistently** losing dogfights to the F-86
- Answer: the **hydraulic controls** of the F-86 allowed its pilot to repeatedly **observe, orient, plan and act** with less effort than the MIG-15 pilot
- Hence, Boyd's Law: in analyzing complexity, **fast iteration** almost always produces better results than in-depth analysis

No Silver Bullet — Essence and Accidents of Software Engineering (Fred Brooks, 1975)



*“The essence of a software entity is a **construct of interlocking** of data sets, relationships among data items, algorithms, and invocations of functions [...] I believe **the hard part of building software** to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation [...] If this is true, building software will **always** be hard. There is inherently **no silver bullet**.”*

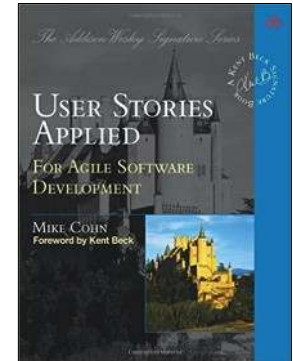
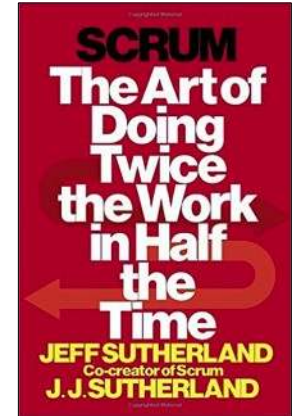
- Exploiting the mass market to **avoid constructing** what can be bought.
- Using rapid prototyping as part of a **planned iteration** in establishing software requirements.
- Growing software organically, **adding more and more function to systems** as they are run, used, and tested.
- Identifying and developing the **great conceptual designers** of the rising generation

The Agile Manifesto & Scrum (2001)

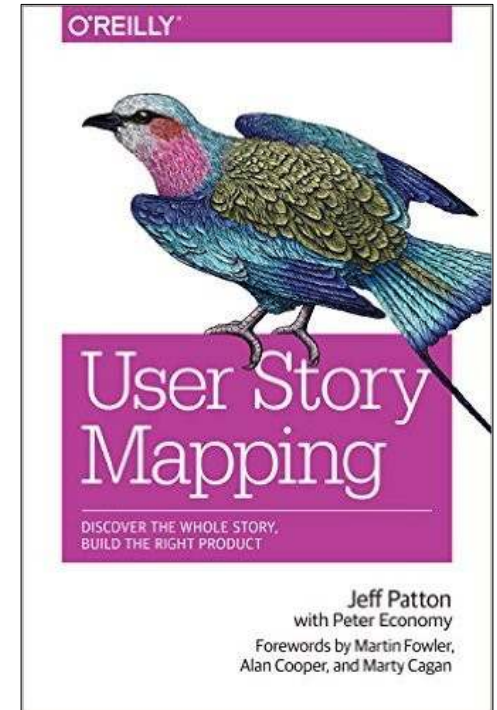
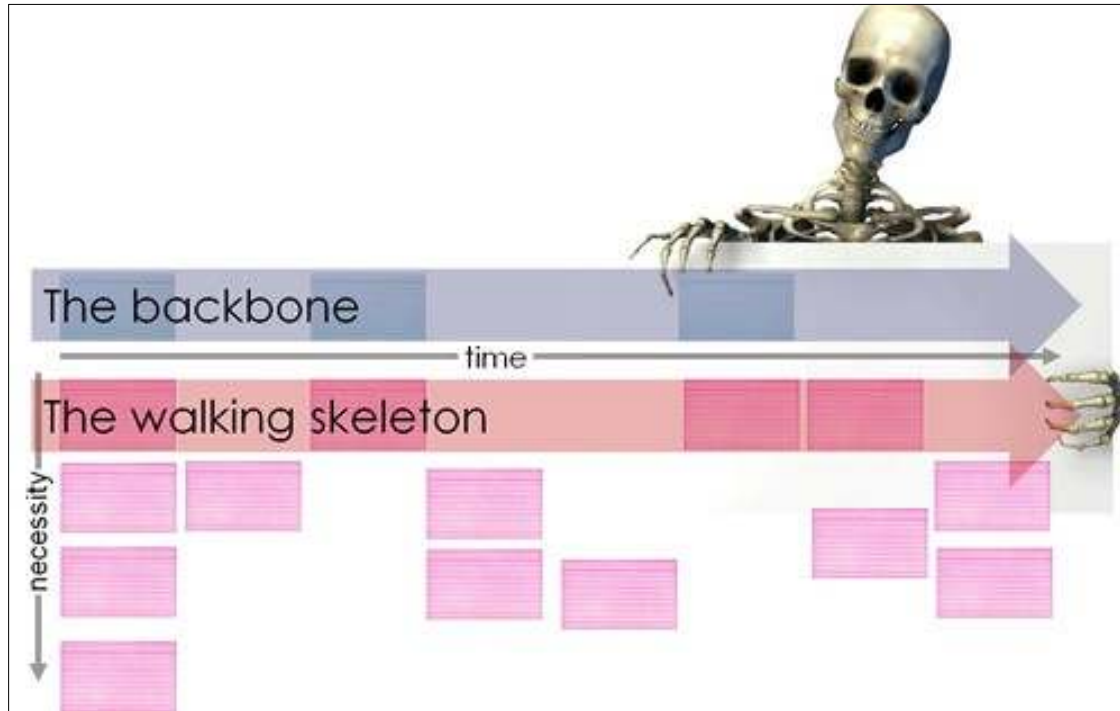
We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

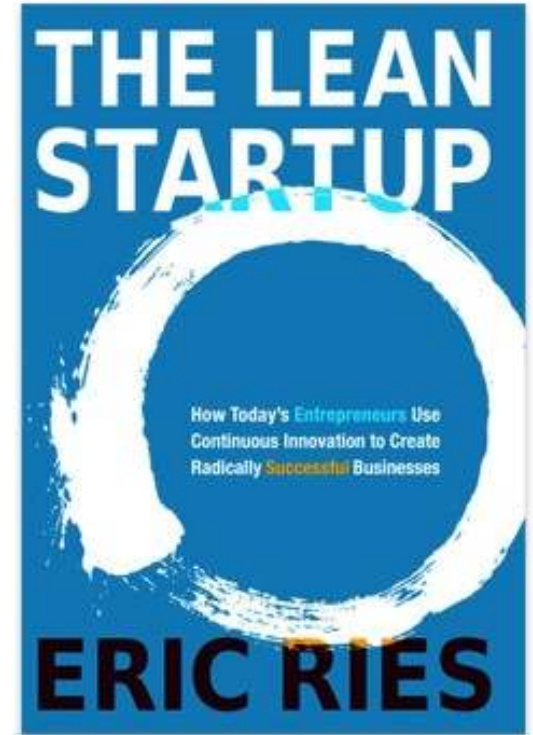
That is, while there is value in the items on the right, we value the items on the left more.



Story Mapping (Jeff Patton, 2005)

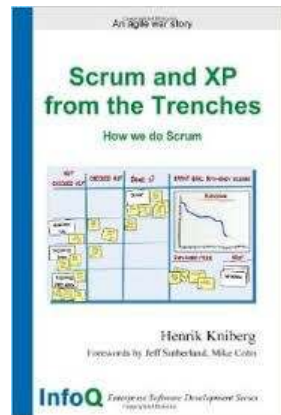


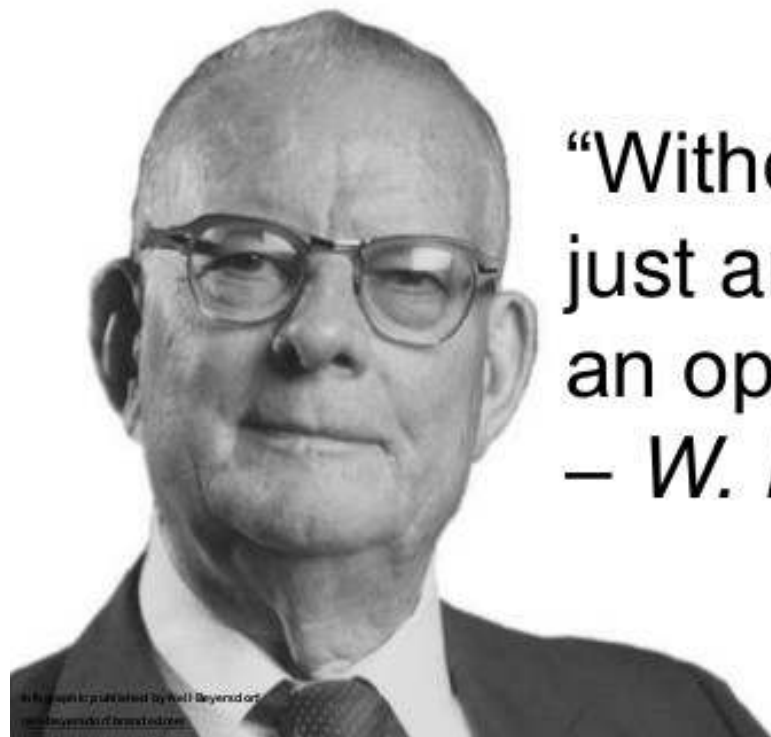
The Lean Startup (Eric Ries, 2011)



Caveats

- Agile has become a cult, like ISO9001 30 years ago.
- Be a practitioner, not a priest.
- Yes, it's OK to adapt Scrum/Kanban to your own Whatever works.
- The Agile Manifesto is the light in the Stay on the path, you'll be fine.
- Lean, MVP, walking skeleton: fine, but make sure you eventually deliver something consistent. Sum of demos and PoCs != Product
- « Fail fast », « move fast, break stuff », « trial and error »: fine too, but make sure you have data and KPIs, or how will you know you failed?
- Which brings me to...





“Without data you’re
just another person with
an opinion.”

– *W. Edwards Deming*



**Opinions are like assholes.
Everybody has one.**

In summary, you are expected to...

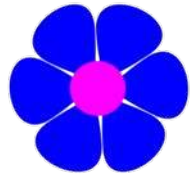
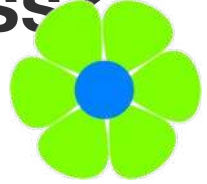
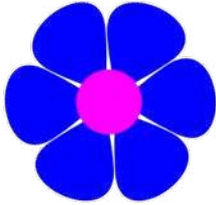
- Deliver new, working code at the drop of a hat
- Build it AND run it (DevOps, remember?)
- Iterate, learn & adapt as fast as possible
- Manage all necessary data for company-wide decisions
- Preserve cash while doing it: survival starts with not spending foolishly
- Be prepared to scale like Hell when you're told to

Infrastructure

Get it out of the way



Ready for 30 seconds of infrastructure bliss?



How about the real story?

Meet the Horsemen of Infrapocalypse ©

Buy

Build

Run

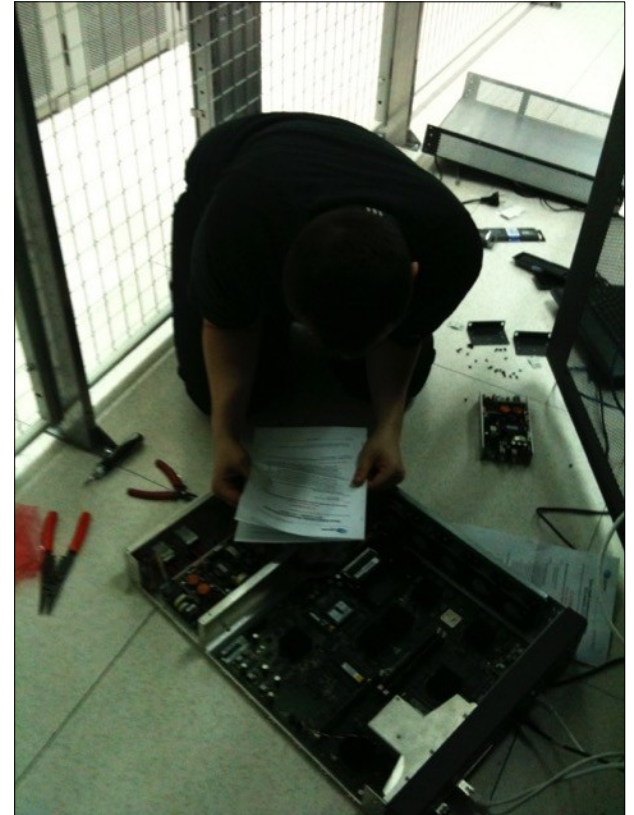
Disaster recovery

MATCOK
2011

Buying physical infrastructure



Building physical infrastructure



Running, scaling, fixing



Disaster Recovery (or not)



Verizon facilities (Nov'12, Sandy in NYC)



Personal collection



IBM datacenter (Dec'11, Tokyo)



Do you see the light?

Reminder:

- Deliver new, working code at the drop of a hat
- Build it AND run it (DevOps, remember?)
- Iterate, learn & adapt as fast as possible
- Manage all necessary data for company-wide decisions
- Preserve cash while doing it: survival starts with not spending foolishly
- Be prepared to scale like Hell when you're told to



Ask yourself:

does physical infrastructure **help** with these goals?

Or does it **stand in the way**?

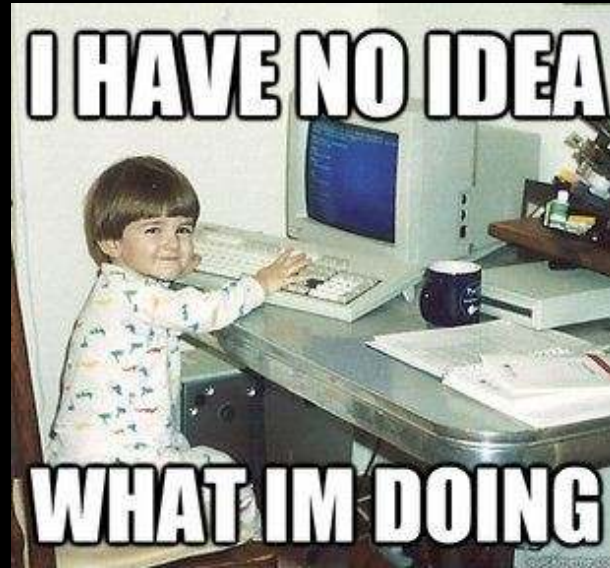
Yeah, thought so. The **Cloud**, then.

Minimum Viable Infrastructure

- Infrastructure should be **iterative** as well
- **Early days**
 - Take the **fastest route** to customer discovery
 - **PaaS platforms** rock: Heroku, Elastic Beanstalk, Convoy, etc.
- **As you grow**
 - **Immutable** infrastructure (and Docker isn't the only way)
 - No plumbing! Use **Managed Services** as much as you can
 - **Automate** everything
 - Infrastructure Deployment: CloudFormation, Terraform, etc.
 - Application Deployment: Chef, Puppet, etc.
 - **Scale-out**

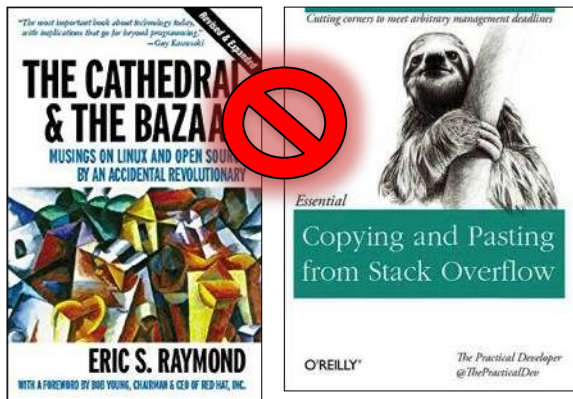


Architecture

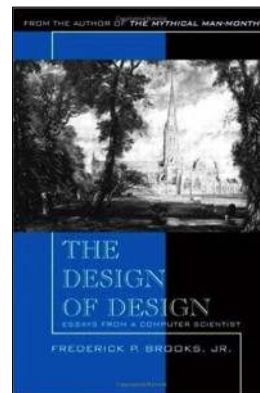
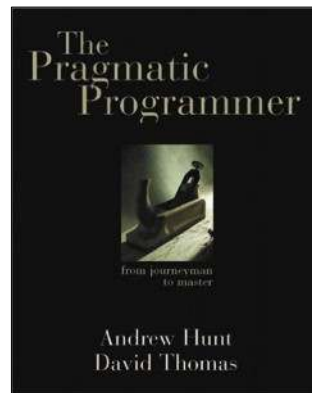


A generation lost in the Bazaar (PHK, 2011)

*“That is the sorry reality of the bazaar Raymond praised in his book: a pile of **old festering hacks**, **endlessly copied and pasted** by a clueless generation of IT “professionals” who wouldn’t recognize sound IT architecture if you hit them over the head with it”*



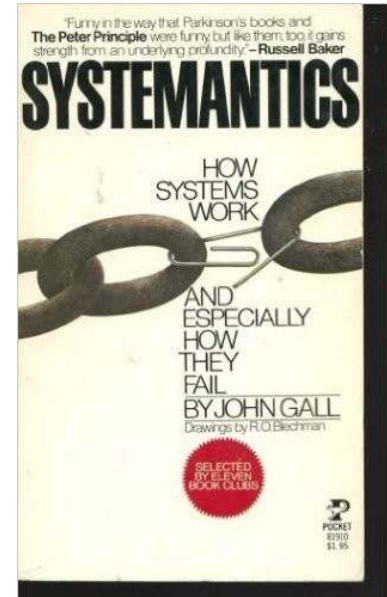
*“Quality happens only
when someone is
responsible for it”*



Gall's Law (1975)

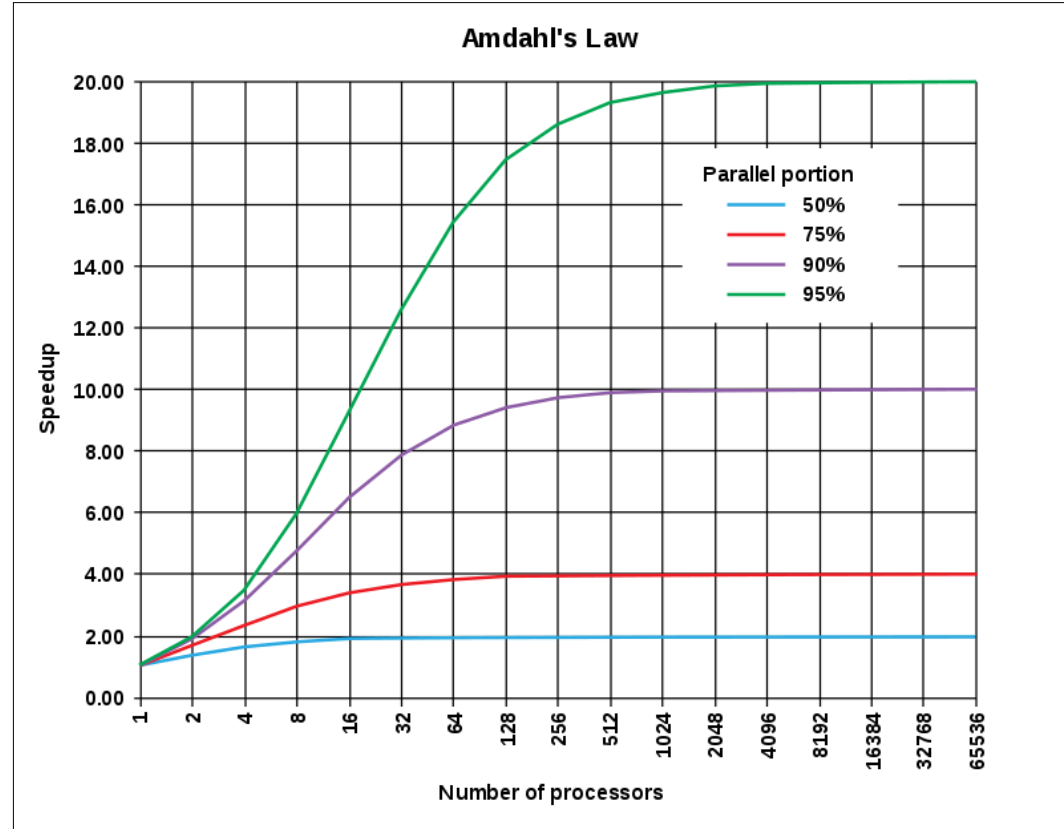
“A complex system that works is invariably found to have evolved from a simple system that worked.”

“A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.”



Amdahl's Law (1967)

- Theoretical **speedup** of the execution of a task - at fixed workload - that can be expected of a system whose **resources** are improved.
- In other words: **hardware** parallelism only helps as far as **software** can be parallelized. **Moore's Law** doesn't come for free!



“You are not paid to write code” (2016)

“Many people grumble or reject the notion of using proven tools or techniques. It’s boring. It requires investing time to learn at the expense of shipping code. It doesn’t do this one thing that we need it to do. It won’t work for us. For some reason—and I continue to be completely baffled by this—everyone sees their situation as a unique snowflake despite the fact that a million other people have probably done the same thing [...]

I was able to come to terms with this once I internalized something a colleague once said: you are not paid to write code. You have never been paid to write code. In fact, code is a nasty byproduct of being a software engineer.”

I'll say it again: don't write code unless you must

- Every single line of code is **technical debt**
- Design Patterns: only if they make **sense**. Don't play with yourself.
- What's your self-delusional story for not using:
 - **Off-the-shelf libs & tools**
 - Standard language libs, Boost, Guava, etc.
 - Logging, parsing, math, crypto, etc.
 - **SaaS platforms**
 - Authentication, SSO
 - Tags, trackers, A/B testing, dashboards
 - Complex but non-core business processes: media transcoding, document processing, etc.

Monitoring is a first-class citizen

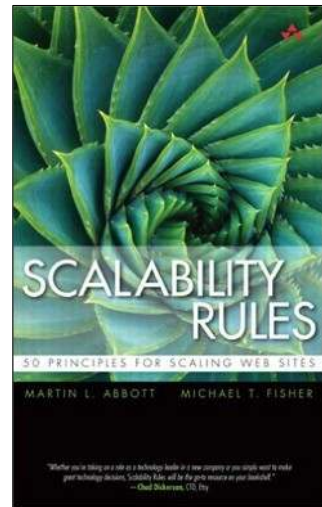
- Design for monitoring **from day 1**.
- Don't send monitoring information by **email**. No one reads it.
- Logging to **local files** is not enough and may create more problems
 - **Centralize** log storage and processing (ELK, Splunk, etc.)
 - If something is **really** urgent, it shouldn't be written to a file!
- Monitor all services carefully
 - **Inside view** (Nagios) and **outside view** (Catchpoint, PagerDuty)
 - Don't forget **3rd party** services
- **Real-time** technical & business metrics are priceless (Graphite, Kibana)

Do it to them before they do it to you

- Make sure you understand **how things fail** (periodic testing)
- Make sure you understand **why and how they failed** (post mortems)
- Implement **circuit breakers** and **feature toggles**
- Find out how your service behaves in **degraded mode**
- **Break stuff**, see what happens. Most of your assumptions will be **wrong**.

Minimum Viable Architecture

- Early days: take the **fastest route** to customer discovery. No points for style, but you should be aware of the shortcuts.
- As you grow: **APIs** and **microservices**, but there is no silver bullet ☺
 - State management
 - Dependency Hell
 - Impedance mismatch between services and backends
 - Deployment & monitoring
 - Authentication across services
 - Etc.
- Good news: scalability is **almost free** when your design is clean



Code Hygiene



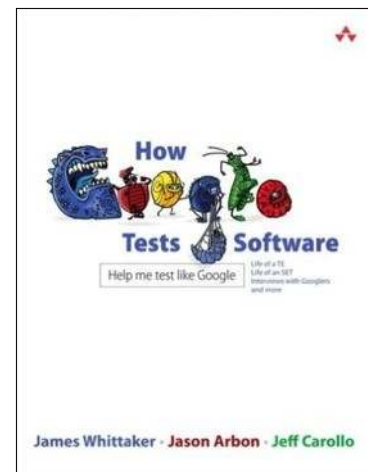
Coding vs Debugging vs WTF

- How much of your time is spent
 - working on product features?
 - investigating or fixing bugs?
 - doing something else?
- From my experience: 25% / 50% / 25%
- You only get a day's worth of coding per week. Make it count.

Half your time is spent fixing bugs

1. Source Lines of Code (KSLOC) Generated Per Year	200
2. Average Bugs Per 1000 SLOC ⁱⁱ	x 8
3. Number of Bugs in Code	= 1,600
4. Average Cost to Fix a Bug ⁱⁱⁱ	x \$1,500
5. Total Yearly Cost of Bug Fixing	= \$2,400,000
6. Yearly Cost of an Engineer ^{iv}	/ \$150,000
7. Number of Engineers Consumed with Bug Fixing	= 16
8. Engineering Team Size	/ 40
9. Percentage of Staff Used for Bug Fixing	= 40%

Software Testing Phase Where Bug Found	Estimated Cost per Bug
System Test	\$5000
Integration Test	\$500
Full Build	\$50
Unit Test / Test Driven Development	\$5



Spending less time & money on bugfixing

- Write **less code**
- Find bugs **faster** and **sooner**
 - Deploy the **best CI/CD** you can get
 - And don't go building it! Lots of **SaaS options**: Travis, CircleCI, etc.
- Fix bugs **faster**
 - Deploy the best **real-time monitoring** you can get
 - If you have budget for **APM tools** like New Relic or Dynatrace, spend it!
- Don't fix all bugs: *"Fixing bugs is only important when the **value** of having the bug fixed exceeds the cost of the fixing it"* - Joel Spolsky

The Joel Test – 12 Steps to Better Code (2001)

The Joel Test

1. *Do you use source control?*
2. *Can you make a build in one step?*
3. *Do you make daily builds?*
4. *Do you have a bug database?*
5. *Do you fix bugs before writing new code?*
6. *Do you have an up-to-date schedule?*
7. *Do you have a spec?*
8. *Do programmers have quiet working conditions?*
9. *Do you use the best tools money can buy?*
10. *Do you have testers?*
11. *Do new candidates write code during their interview?*
12. *Do you do hallway usability testing?*

The Simple Programmer Test

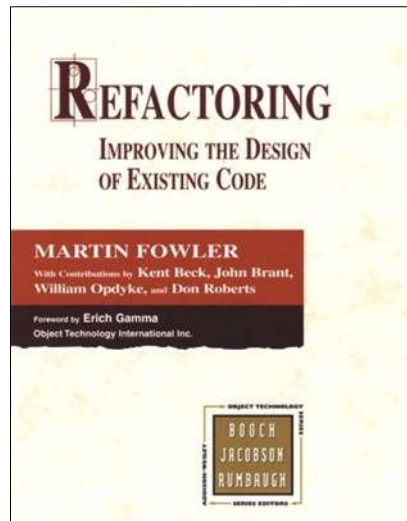
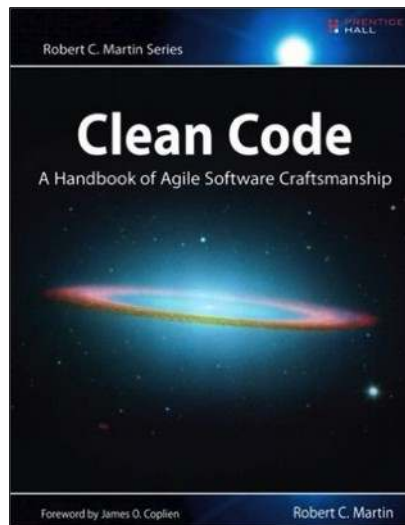
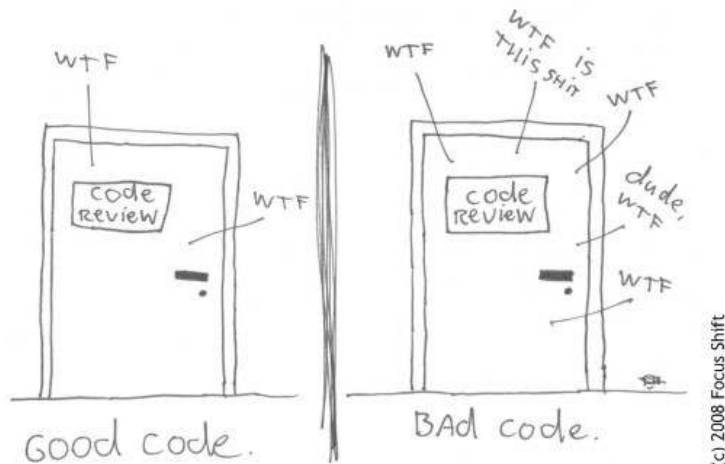
1. *Can you use source control effectively?*
2. *Can you solve algorithm-type problems?*
3. *Can you program in more than one language or technology?*
4. *Do you do something to increase your education or skills every day?*
5. *Do you name things appropriately?*
6. *Can you communicate your ideas effectively?*
7. *Do you understand basic design patterns?*
8. *Do you know how to debug effectively?*
9. *Do you test your own code?*
10. *Do you share your knowledge?*
11. *Do you use the best tools for your job?*
12. *Can you build an actual application?*

BONUS

Your real score is the one I'd find if I audited you ;)

Code smell

The only valid measurement
of code quality: WTFs/minute



Optimization

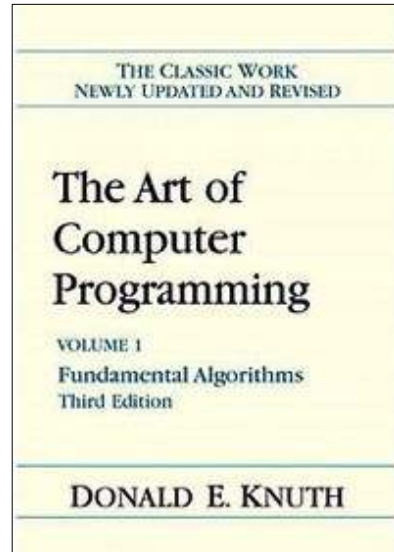
Playing with yourself again?

(thank me for not adding a picture here)

Premature optimization is the root of all evil (Knuth, 1974)

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

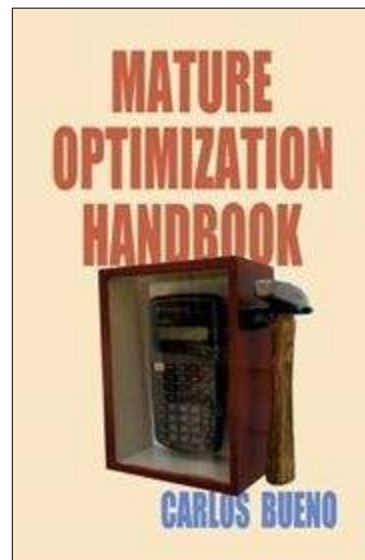
We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%”



Mature Optimization Handbook (Carlos Bueno @ Facebook, 2013)

*“Performance optimization is, or should be, a **cost/benefit decision**”*

*“In my experience, it makes most sense on **mature systems** whose architectures have settled down. New code is almost by definition slow code, but it’s also likely to be ripped out and replaced as a young program slouches towards beta-test. Unless your optimizations are going to stick around **long enough** to pay for the time you spend making them, plus the **opportunity cost** of not doing something else, it’s a **net loss**”*



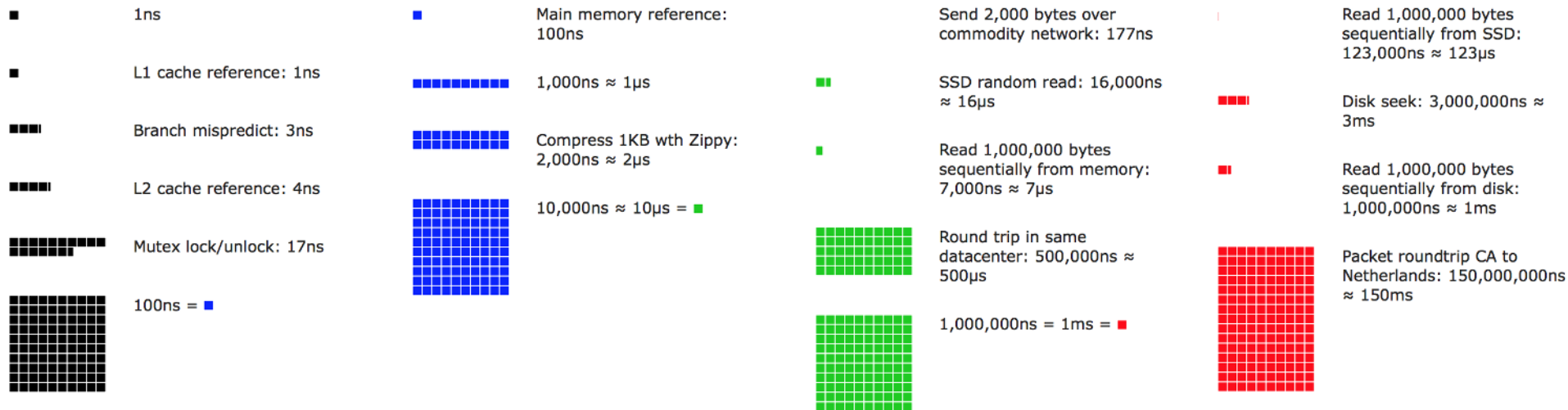
What's the ROI of saving x ms on a feature?

You don't know? My point exactly!

Find out before launching into a useless, risky optimization crusade

Latency Numbers Every Programmer Should Know

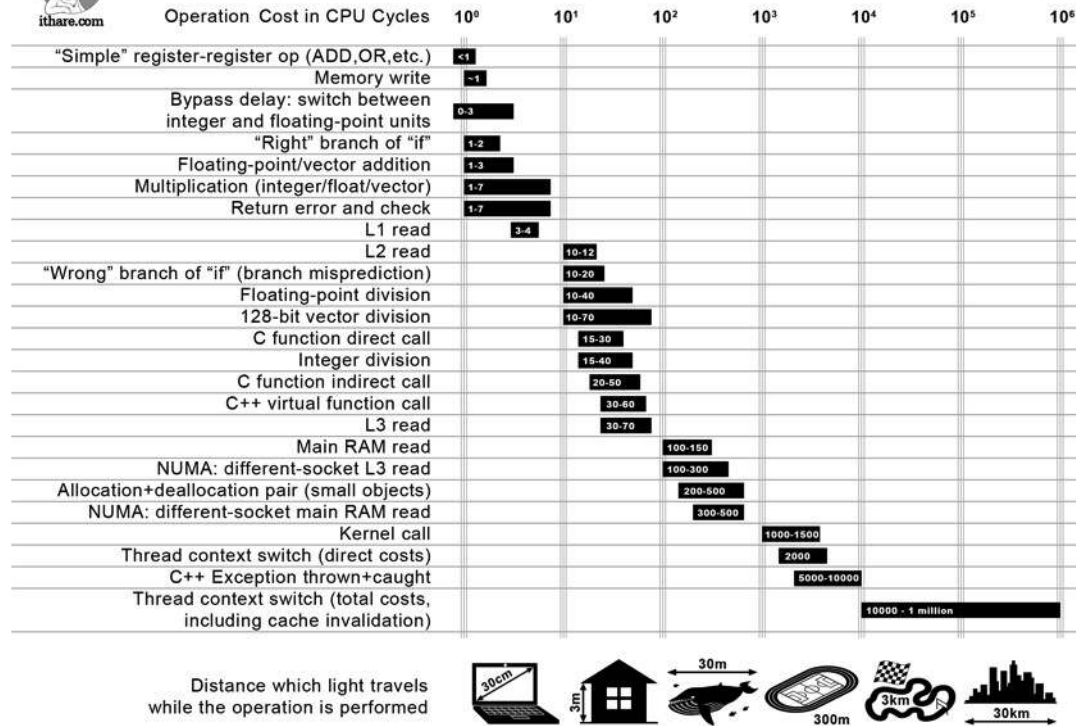
Actually, the numbers don't matter, the ratios do!



Numbers only 0.1% of Programmers Should Know



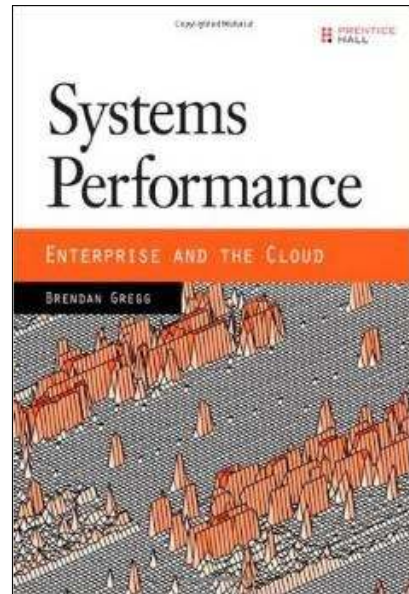
Not all CPU operations are created equal



- Are you writing critical embedded code?
- Or compilers?
- Or HFT code?
- No? Keep out, then

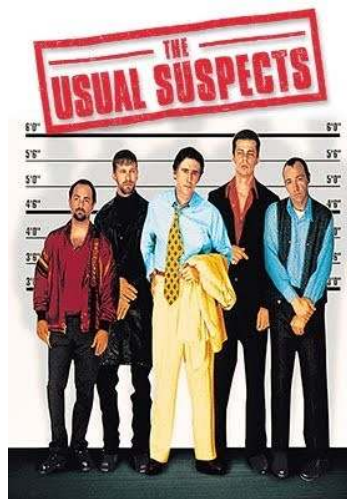
What I think really matters

- Make sure that **critical** product features are “**fast enough**”
- Don't wait **after** product launch to find out that they they're not
- **Performance monitoring** is part on the dev process
 - Unit testing for **raw performance**
 - Load testing for **throughput**
 - Profiling: gprof, Callgrind, Dtrace, APM tools, etc.
- Most of the issues will be **design** or **configuration**, so start there instead of chasing mythical deep-sea monsters



Silent but Deadly (in no particular order)

1. Stupid code (StringBuilder, not String)
2. Naive Memory / Thread Allocation (pooling, dammit)
3. 1+2 inside a loop
4. Database contention, misbehaving connection pools
5. In-memory cache contention (memcached, etc.)
6. Blocking calls and locks in general
7. Garbage Collection in “Workstation Mode”
8. Stupid SQL, too few or too many indexes, ORMs
9. Additional A/B tests or logs “that someone needed”
10. Parsing data (XML, JSON, etc) without a parser “because it’s simpler”
11. ...





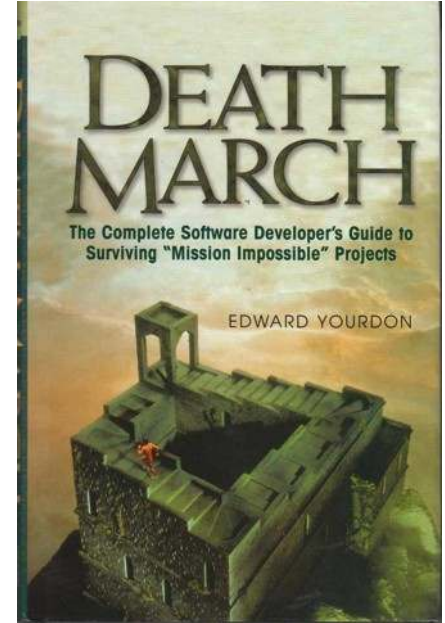
The Bleeding Edge

you're gonna bleed, oh yes you are

Death march projects (Yourdon, 1996)

“What really contributes to the death march nature of such projects [...] is the attempt to use bleeding-edge technology for an industrial-strength application. Even if the technology is basically usable, it often does not scale up well for large-scale usage; and nobody knows how to exploit its strengths and avoid its weaknesses; and the vendors don't know how to support it properly; and on and on...”

“Is it any wonder that things degenerate into a death march project, with everyone working late nights and long weekends in order to coax the experimental new technology into some semblance of working order?”



Choose boring technology (McKinley, 2015)

What counts as boring? That's a little tricky. *"Boring" should not be conflated with "bad" [...]* There are many choices of technology that are boring and good, or at least good enough. MySQL is boring. Postgres is boring. PHP is boring. Python is boring. Memcached is boring. Squid is boring. Cron is boring.

"The nice thing about boringness (so constrained) is that the capabilities of these things are well understood. But more importantly, their failure modes are well understood."

"New technology choices might be purely additive (for example: "we don't have caching yet, so let's add memcached"). But they might also overlap or replace things you are already using. If that's the case, you should set clear expectations about migrating old functionality to the new system."

Horror Stories Are Just Stories

- Every single piece of tech has horror stories
- They're fun to read and may teach you a thing or two
- However, most of the time they're not relevant to your platform
 - Different version
 - Different workload
 - Different scale
 - etc.
- So do not base any technology choice on horror stories
 - One exception: ESBs and other shit Enterprise middleware. Just say no.
- Remember: it's easier to blame tech than to expose your own mistakes

And please don't be one of these

- **Trolls**: « *Java is for pussies. Real men use C++* »
- **Lunatics**: « *Erlang is the future* ». Also works with Scala, Haskell, Clojure, etc.
- **Old fart**: « *SQL Server has always worked for us* » (and they buy us lunch)
- **PhD egghead**: « *I think we can reduce latency by tweaking the Ethernet driver* »

- And the worst of all, **fanboys**
« *guys, HackyLib v0.1 has just been put in production. It's totally awesome. Spotify, Netflix and Amazon are all using it in production. Let's use it too!* »

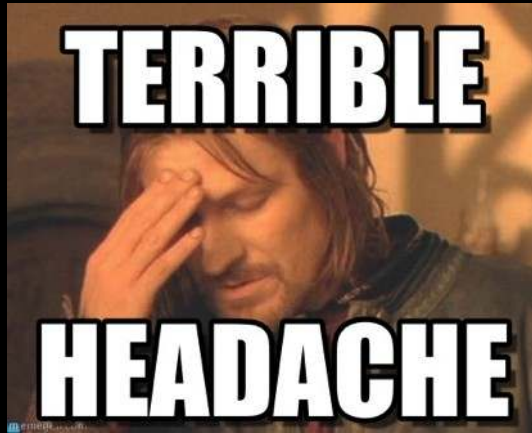
Make **your own choices**, build **your own platform**.



Use common sense

1. Focus on **business needs**
2. Identify **top challenges**: time to market? UI? Perf? Security? Don't know?
3. List **candidate** technologies, expecting them to last **at least a year** (think 10x)
4. KPIs, benchmarks, PoC: **educated** guess is OK, **random** decision isn't!
5. Implement, deploy and **monitor**
6. Anything on fire?
 - Can it be fixed by **refactoring** or **optimization**?
 - If not (are you **really** sure?), can it be fixed with **new technology**?
 - Yes: you need a new building block in your stack, GOTO 2.
 - No: WTF? Are you scared? Man up! **Not moving = death**
 - If there is absolutely no other way, add servers...
but **it won't work forever!**

Summing things up



- Your project will die unless it finds **product-market fit** fast enough
- Unless you're GAFA / FANG / NATU, you have the **same ideas, tech and algos** as everyone else: this isn't how you'll win the fight
- Turn ideas into MVPs as **fast** and **frugally** as possible
 - **Lean & Agile methods**
 - **Flexible** design, **well-understood** technology choices
 - **Best** development tools possible
 - **Solid** data stack to collect, store, process and visualize
- Keep a **critical eye** out for **new tech**, but think twice (or more) before adding it to your stack
- When the Big Boys push the button, be ready to **scale like a motherfucker**

Thank you!

julien@julien.org
@julsimon

*"All that is gold does not glitter,
Not all those who wander are lost;
The old that is strong does not wither,
Deep roots are not reached by the frost"*