

# Lecture 35 – DevOps for P4P

Patrick Lam & Jeff Zarnett

[patrick.lam@uwaterloo.ca](mailto:patrick.lam@uwaterloo.ca) [jzarnett@uwaterloo.ca](mailto:jzarnett@uwaterloo.ca)

Department of Electrical and Computer Engineering  
University of Waterloo

October 22, 2019

So far, one-off computations:  
you need to answer a question,  
so you write code to do that.

But many systems are long-running.  
⇒ Operations.

MPI: still mostly one-off computations  
(even though on multiple computers.)

Cloud computing: often long-lived systems,  
but we didn't talk about how.

Today: many companies fuse  
development (writes the software)  
and operations (tends the software).



Startups:

No money to pay for separate developer and operations teams.

Not that many servers,  
just a few demo systems, test systems, etc...  
but it spirals out from there.

You're not really going to ask Sales to manage these servers, are you?  
So, there's DevOps.

Is DevOps a good idea?

Can be used for both good and evil.

Good:

- developers involved across the software lifecycle.  
(can learn a lot doing ops...)
- developers motivated to use correct tools & document processes.

Ugly:

- also something to be said for never letting the developers out of their cubes and keeping them far, far away from customers.

They will be scared.

Both parties.

Systems have long come with complicated (“flexible”) configuration options.

Sendmail is particularly notorious, but apache and nginx aren’t super easy to configure either.

First principle: treat *configuration as code*.

- use version control on your configuration.
- test your configurations.
- aim for a suite of modular services that integrate together smoothly.
- refactor configuration files.
- use continuous builds (more on that later).

Excellent idea: tools for configuration.

Not enough to write text

“How to Install AwesomeApp”

e.g. use Red Hat Package Manager (RPM)—  
build, installation, and update automatic & simple.

Complicated means mistakes... people forget steps.  
They are human.

Servers means servers, or virtual machines, or containers.

At scale (smaller than you think):  
use mass tools for dealing with servers,  
rather than doing tasks manually.

At least: cloud-like server initialization without  
manual intervention;  
must be able to spin up a server programmatically.

Use APIs to access your infrastructure. Examples:

- storage: access layer to MongoDB/Amazon S3/etc;
- naming and discovery infrastructure;
- monitoring infrastructure.

Avoid one-offs—use open-source tools when applicable.  
But build your own tools if needed.

eBay:

- 1995: perl scripts;
- 1997: C++/Windows;
- 2002: Java.

Each of these architectures was appropriate at the time, but not as requirements changed.

More sophisticated successor architectures would have been overkill earlier.

Hard to predict what's needed in the future.

*“Perf is a feature”.*

— Jeff Atwood

That is: apply developer time to perf,  
and make engineering tradeoffs to get it.

Some thoughts:

- design with the eventual replacement in mind;
- don't abandon internal quality (e.g. modularity);
- sacrifice individual modules at a time,  
not the whole system;
- implement new features with a rough draft and  
deploy to a test audience.

Naming is one of the hard problems in computing.

There are only two hard things in computers:

- cache invalidation,
- naming things, and
- off by one errors.

- use canonical one-word names for servers;
- but, use aliases to specify functions, e.g. 1) geography (nyc); 2) environment (dev/tst/stg/prod); 3) purpose (app/sql/etc); and 4) serial number.

There's also the Java package approach of infinite dots:  
live.application.customer.webdomain.com or however  
you want to call it.

Pick something and be consistent.

- pull code from version control;
- build;
- run tests;
- report results.

# Continuous Integration Social Convention

Don't break the build (or donuts).



Run the CI cycle on every commit;  
results sent by e-mail or instant messenger.



Deploy new code incrementally in production,  
also known as “test in prod”:

- stage for deployment;
- remove canary servers from service;
- upgrade canary servers;
- run automatic tests on upgraded canaries;
- reintroduce canary servers into service;
- see how it goes!

Of course: implement your system with rollback.

Things to think about:

- CPU Load
- Memory Utilization
- Disk Space
- Disk I/O
- Network Traffic
- Clock Skew
- Application Response Times

Multiple systems: need an overview of all the systems.

Summary needs to show whether anything is wrong,  
but not an overwhelming wall of data.



**ALERT**  
CONDITION: RED

Don't pay someone to stare at the dashboard and  
press the "Red Alert!" button  
if anything goes out of some preset range.

No, for that we need some automatic monitoring.

- **Alerts:** a human must take action now;
- **Tickets:** a human must take action soon (hours or days);
- **Logging:** no need to look at this except for forensic/diagnostic purposes.

Common bad situation: logs-as-tickets.



# Feuerwehr

Scheibe einschlagen  
Knopf tief drücken



## Action Stations! Set Condition One Throughout the Ship

What do you do when you hear the fire alarm?

If there is an actual fire, you will not only be wrong, you might also be dead.

Alerts and tickets are a great way to make user pain into developer pain.

Some SUPER CRITICAL ticket OMG KITTENS ARE ENDANGERED is an excellent way to learn the lesson...

Devs will take steps that keep these things from happening in the future.

Key idea: scaling to big data systems introduces substantial overhead.

Up next: Laptop vs. 128-core big data systems.

Are big data systems obviously good?  
Have we measured (the right thing)?

The important metric is not just scalability;  
absolute performance matters a lot.

Don't want: scaling up to  $n$  systems  
to deal with complexity of scaling up to  $n$ .

Or, as Oscar Wilde put it:  
“The bureaucracy is expanding to meet the  
needs of the expanding bureaucracy.”

Compare: competent single-threaded implementation vs. top big data systems.

Domain: graph processing algorithms—  
PageRank and graph connectivity  
(bottleneck is label propagation).

Subjects: graphs with billions of edges  
(a few GB of data.)

Twenty pagerank iterations

<b>System</b>	<b>cores</b>	<b>twitter_rv</b>	<b>uk_2007_05</b>
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Single thread	1	300s	651s

Label propagation to fixed-point (graph connectivity)

<b>System</b>	<b>cores</b>	<b>twitter_rv</b>	<b>uk_2007_05</b>
Spark	128	1784s	8000s+
Giraph	128	200s	8000s+
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread	1	153s	417s

- “If you are going to use a big data system for yourself, see if it is faster than your laptop.”
- “If you are going to build a big data system for others, see that it is faster than my laptop.”