Aalto University
School of Electrical
Engineering

Deployments

14.3.2019
Santeri Paavolainen

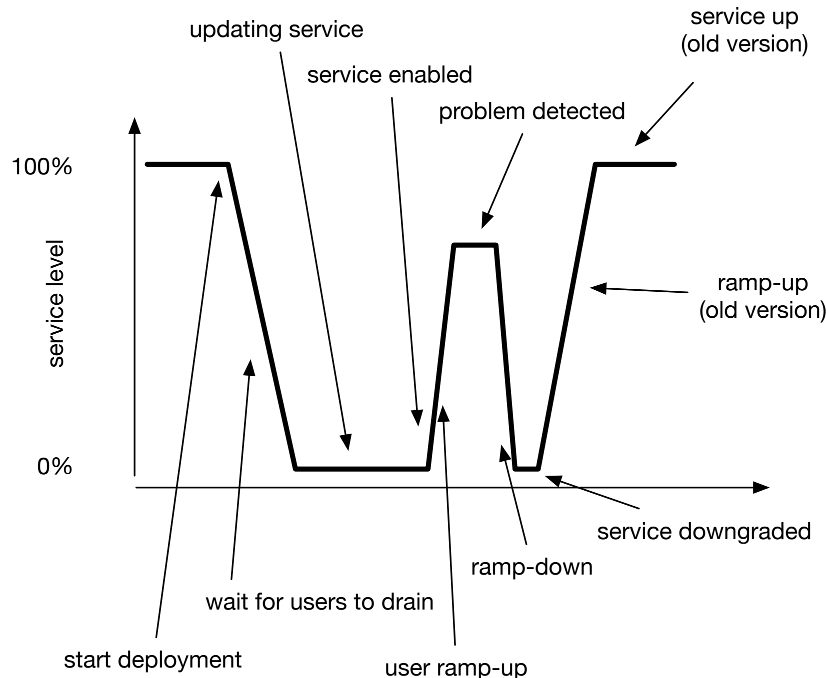
Previously …

- **Assumed that a service or a system magically**
  - Starts from code on a developer's machine, and
  - turns into artifacts suitable for deployment, and
  - gets to being run in a computing environment somewhere.
- **During development this not a problem**
  - Single user system (developer!), no conflicts
  - Manual deployment (`kubectl apply`, `./start.sh`)
- **… however, in the real world …**

CHELSEA HOTEL
STAR LOUNGE
LIFE IS GAME

自転車等
放置禁止区域
● 図の赤色部分は、渋谷区自転車等の放置防止等に関する条例に基づく放置禁止区域です。
● 禁止区域内に放置してある自転車等は直ちに撤去します。

Why deployment relevant?



- **Simple business logic**
  - Customers not served = $ not generated
  - Customers not happy = $ less in future
  - Loss of reputation = $ less in future
- **Two problems**
  - Maximize service time
  - Minimize affected users
    - *In **any way**, including corruption*

Corollary:

If you don’t care about users, neither downtime nor correctness matters

General solutions

- **Stop-and-go deployment**
  - Stop the world
  - Update
  - Start the world
- **Service degradation**
  - Fallback services
  - Read-only mode
- **Non-stop deployments**
  - Blue-green deployments
  - Canaries etc.

- **Minimizing critical intervals**
  - Database techniques

- **Minimizing affected users**
  - E.g. avoiding big bugs
  - Scientist
  - Multivariate feature flags

- **Later: Destructive changes**

Stop-and-go deployment

- **Simplest**
  - Almost all problems during upgrade are related to state!
  - State in stable (not changing) state easiest to handle
- **All-or-nothing**
  - Difficult to test with small number of users (possible, but bad $)
  - Rollback affects also everything similarly (stop for rollback)
- **Any scripting tool with or without CI/CD works**
  - Shell scripts (used this with early EC2!)
  - Nowadays Puppet, Chef, Fabric, CloudFormation, Terraform, …
  - ”`kubectl delete -f old.yml; kubectl apply -f new.yml`”

Other loosely coupled services

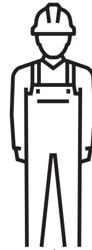
- **Workflow engines & batch processing easier**
  - Stop and go works pretty well
- **Same with any kind of queued service**
  - If the underlying application is tolerant on delays (e.g. deployment length)

Degraded service deployments

- **Stop-and-go, except stoppage is hidden from users**
- **Discussed service degradation earlier**
  - If a fallback service exists, can use it
  - Trigger circuit breaker explicitly (don't wait for failures)
  - Deploy temporary (reduced) service and switch on LB/DNS

Degraded…: Read-only mode

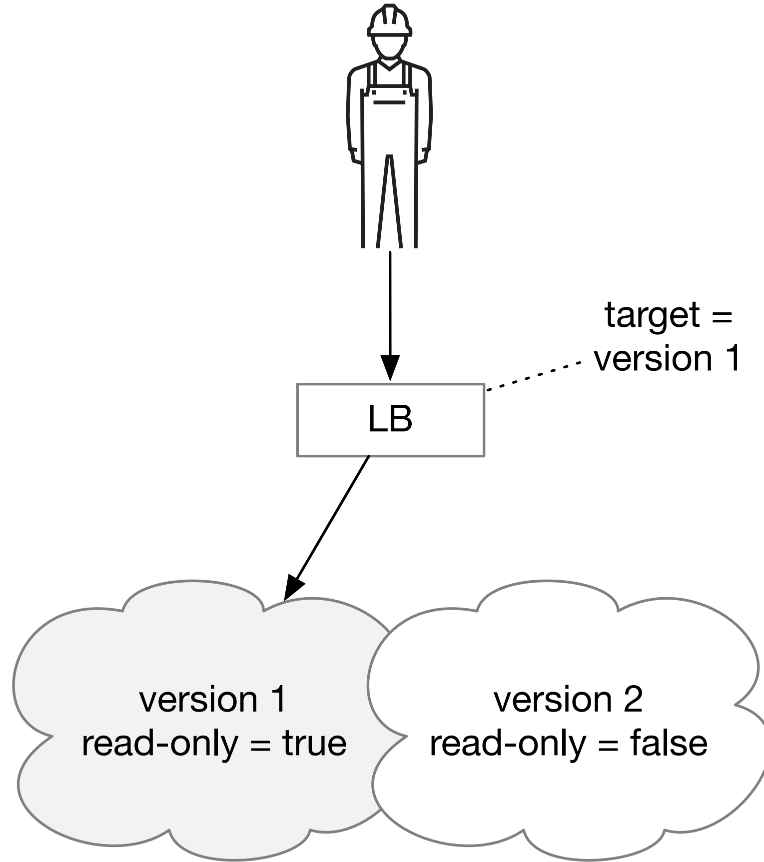
- **Read-only version of service**
- **Keeps old version running and serving (in read-only mode)**
  - Avoid data changes when new version being deployed
- **Very difficult with monolithic services and "big" frameworks**
  - Lots of hidden functionality: "user last active" schema, logging, …
- **Easier with narrowly defined microservices & dynamic configs**
  - Though propagates to user interface problems (e.g. what if read-only flag is raised at the end of a long operation?)

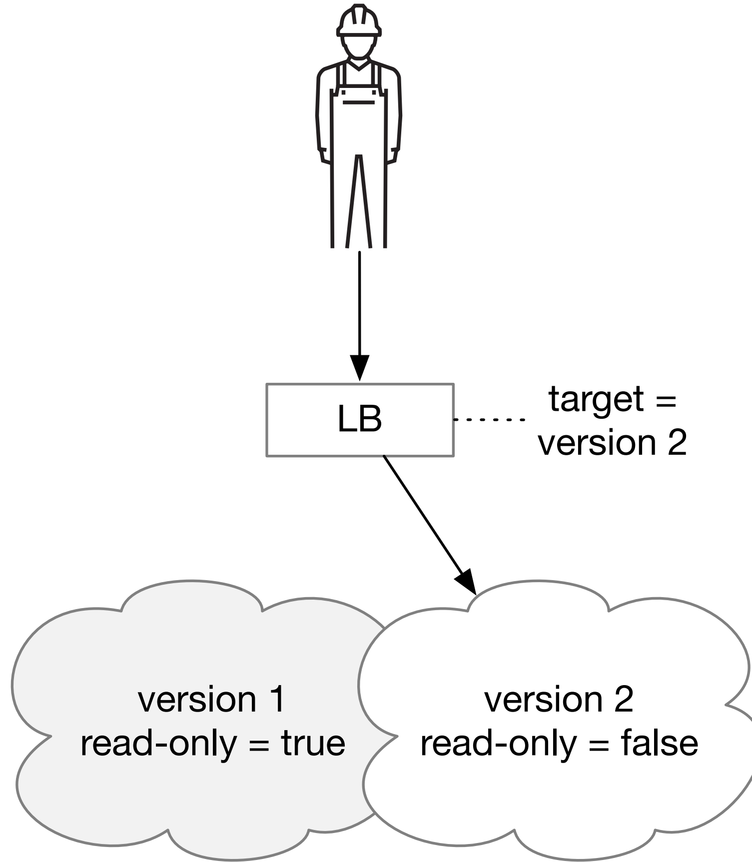


LB

target =
version 1

version 1
read-only = false


target =
version 1
LB
version 1
read-only = true
version 2
read-only = false



(Viability of RO mode rests on cut-over between versions being faster than underlying service upgrade. Which is practically always true.)

Read-only mode

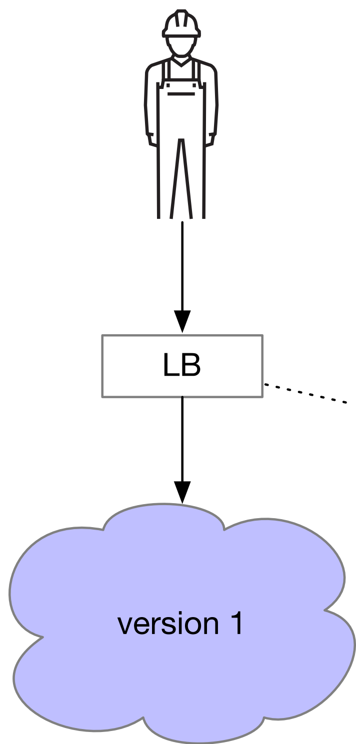
- **Service logic vs. blanket proxy**
  - Read-only logic built into service itself (if complex logic)
  - Blanket proxy blocks POST / PUT / DELETE operations when read-only (sidecar or ELB or own reverse proxy)
  - Depends on application logic (does GET change state?)
- **Redeployment vs. reconfiguration**
  - Redeploy old version with read-only flag enabled
  - With dynamic configuration flag, much, much easier

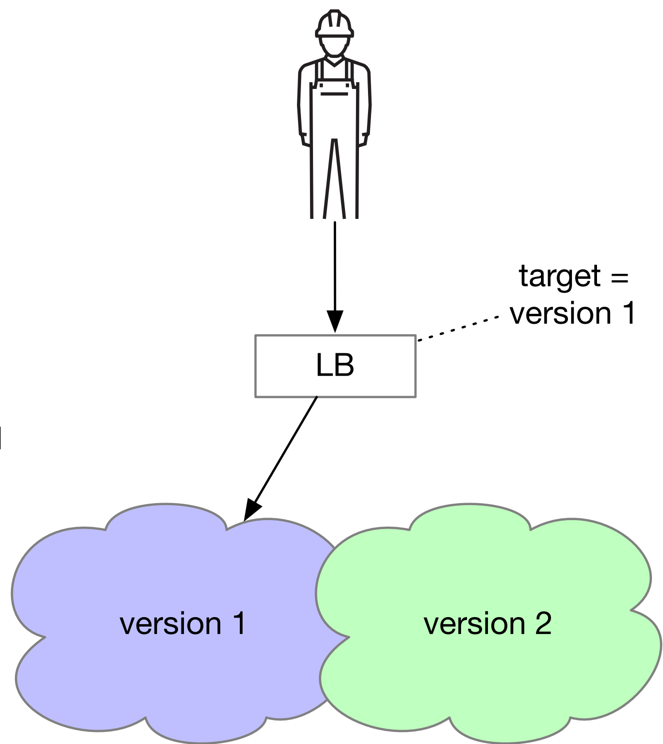
Gradual deployments

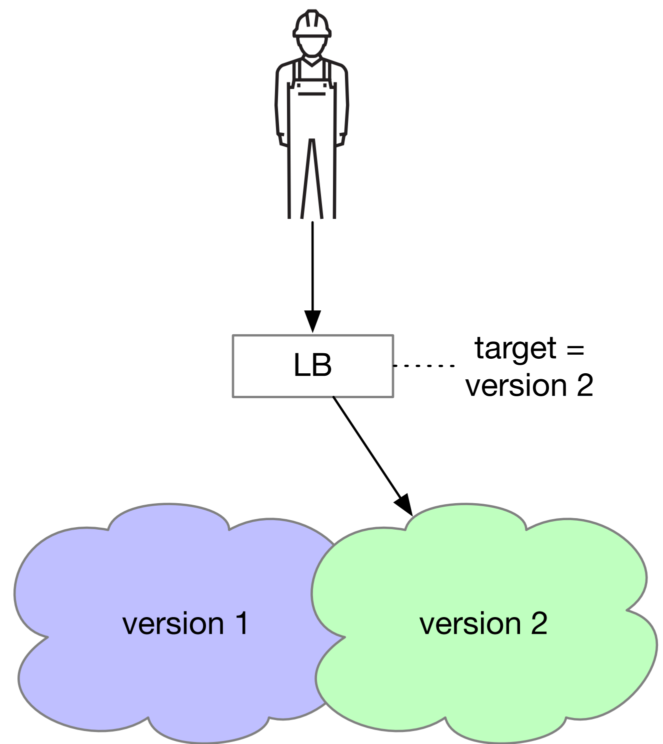
- **Idea: Have old and new version running side-by-side**
  - Instaneous cut-over (blue-green)
  - Gradual migration (canary release)

Blue-green deployments

- **Blue-green deployment model (aka red-black)**
  - Current version is blue
  - Deploy new version (green)
    - *Make sure it is “warm” and has same amount of resources as blue*
  - Perform cut-over and wait
  - Reverse cut-over if problems (fast rollback)
- **Pros: no downtime**
- **Cons**
  - Requires double the infrastructure during deployment
  - Difficult with stateful services
  - Dreadful with schema changes (potentially requiring multi-phase upgrades)
- **See https://www.ianlewis.org/en/bluegreen-deployments-kubernetes**

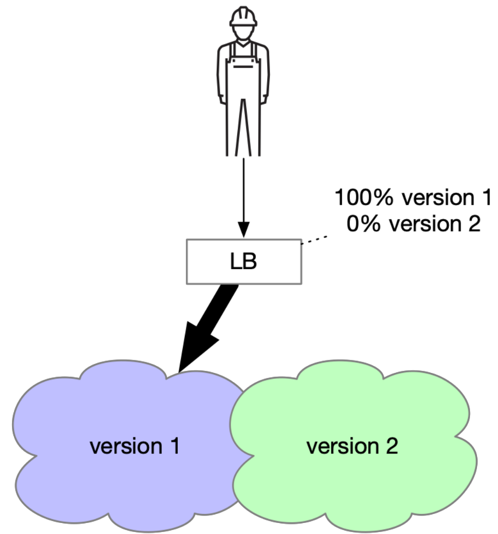

LB
target =
version 1
version 1

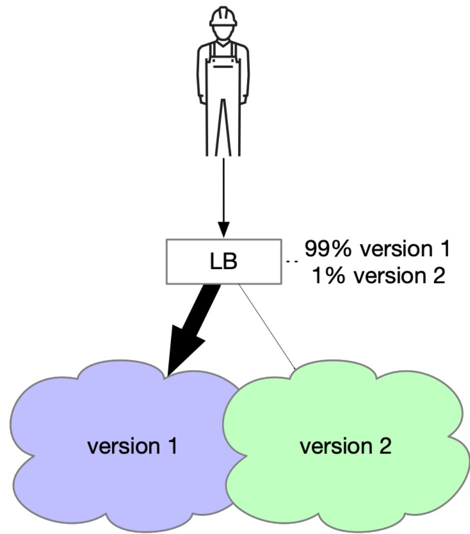

target =
version 1
LB
version 1
version 2


LB
target =
version 2
version 1
version 2

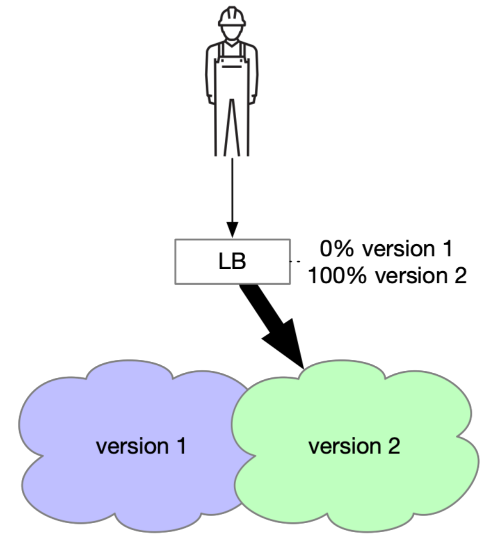
Canary release

- **Idea: Since there is >0 chance of catastrophic failure, why subject <u>all</u> users to that?**
  - E.g. minimize the amount of affected users





time passes



Canary release

- **Idea: Since there is >0 chance of catastrophic failure, why subject <u>all</u> users to that?**
  - E.g. minimize the amount of affected users
- **Canary users**
  - Few users → few % → more % → all users
  - See if any problems occur ➔ move users back to old version
- **Rests on the likelihood of catastrophic failures being**
  - *Detectable quickly (monitoring!)*
  - *Triggered with high likelihood "soon" (lurking bugs difficult)*
- **Requires system to be able to operate <u>under two different versions</u> simultaneously**
  - Stateful services especially with schema changes again very difficult
- **Applicable also to multivariate feature flags ("dark launch")**

"I'm going to have to science the shit out of this."

- **How about A-B testing your code?**
- **Idea: Run original code, and a new version, and compare results**
  - "Scientist" originally for Ruby (https://github.com/github/scientist)
  - Publishes results (time, discrepancies, exceptions, …) for analysis
- **Useful for functional code (does not modify state)**

```ruby
require "scientist"

class MyWidget
  def allows?(user)
    experiment = Scientist::Default.new "widget-permissions"
    experiment.use { model.check_user?(user).valid? } # old way
    experiment.try { user.can?(:read, model) } # new way

    experiment.run
  end
end
```

Destructive changes

Destructive what?

Oops! Too small!

- **Most often database schema changes**
  - `alter table users alter column fullname varchar(16);`
  - Many schema changes are reversible
  - Should always design changes to be reversible
  - Though applies to any persistent data (files, ...)
- **If you lose information on operational data storage**
  - Backups and snapshots are your friend
  - Start polishing up your resume

Protecting your job

- **Exercise your recovery methods (so they don't fail silently)**
  - Example: deploy new versions from backups of old version
- **Code review (e.g. peer review)**
  - Share the responsibility! (No really, more eyes = better chance of finding problems)
- **Don't throw data away until (much) later**
  - ```
    alter table users rename column fullname to fullname_old;
    alter table users add column fullname varchar(16);
    update users set fullname = fullname_old;
    ```
- **Test upgrades aggressively**
  - Use copy of real data, have some sanity testing after upgrading test data
- **Implement changes gradually**
  - Would work on sharded services, not as well now …
    ```

Wait!

- **What if service X depends on new version of service Y?**
  - Should you deploy both at the same time?
  - Close coupling on service deployments (what if one fails?)
- **Canonical answer would be:**
  - either deploy version of X that works with old Y, or
  - deploy new Y so that with backward-compatible interface for X
- **This kind of versioning may incur substantial costs**
  - Lots of effort for very short-lived compatibility issue
- **Answer: it depends …**

Some tools

- **Infrastructure management tools somewhat applicable**
  - Chef, Puppet, Fabric, Terraform, CloudFormation
- **API gateways increasingly support blue/green and canary deployments**
  - AWS API Gateway
  - Ambassador (using Envoy)
  - Nginx roll-your-own works too
  - Plus other OSS and commercial
- **Continuous Delivery – oriented systems**
  - Spinnaker
  - AWS CodeDeploy, Google CD, Azure Pipelines
  - + other hosted CI/CD
  - More comprehensive (but less wiggle room)

Living on the edge

(warning: only tangentially related to deployments)

Aalto University
School of Electrical
Engineering

Image: Netflix, APL 2.0

Chaos engineering

- **If your goal is to have fault-tolerant environments**
  - ... then your environments should be fault-tolerant
  - ... even in production
  - ... so they should not fail even when you intentionally introduce faults
  - ... in production systems!
- **Original chaos monkey: randomly terminate instances in production environment**
- **Originally developed at Netflix**
  - Chaos Monkey, later the whole Simian Army
  - Now maturing as an engineering field
- **Breaking Containers: Chaos Engineering for Modern Applications**
- **This can be scary**
  - Fault-injection testing does increase system reliability

Summary

- **User impact an important consideration for production environments**
- **Deployment mechanism differ on easy ⬄ impact axis**
  - Stop-and-go
  - Service degradation
  - Blue-green and canary releases
- **Be careful with upgrades changing schema and state!**
- **Testing your system**
  - Scientist for testing functional code
  - Chaos engineering for testing fault-tolerance