AARHUS UNIVERSITET

Microservices and DevOps

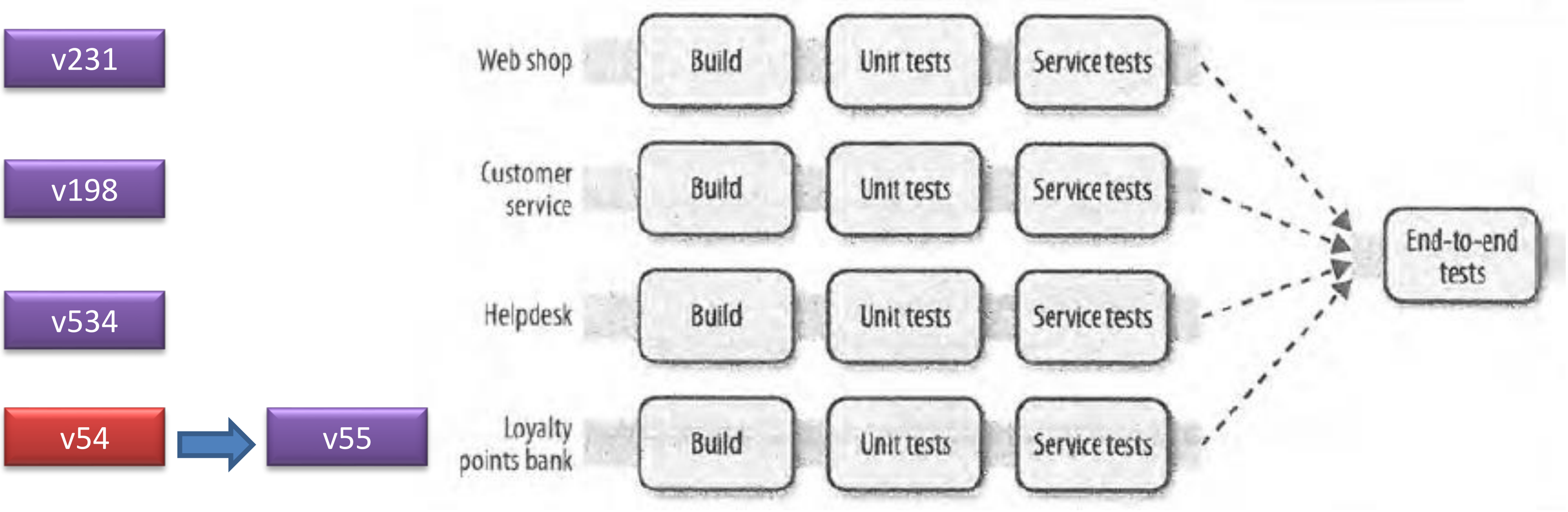
DevOps and Container Technology

Design for Deployment

Henrik Bærbak Christensen

Motivation

- Continuous Delivery of Services
  - Individual versions of services, combined into full system



v231

v198

v534

v54

v55

Web shop

Customer service

Helpdesk

Loyalty points bank

Build

Unit tests

Service tests

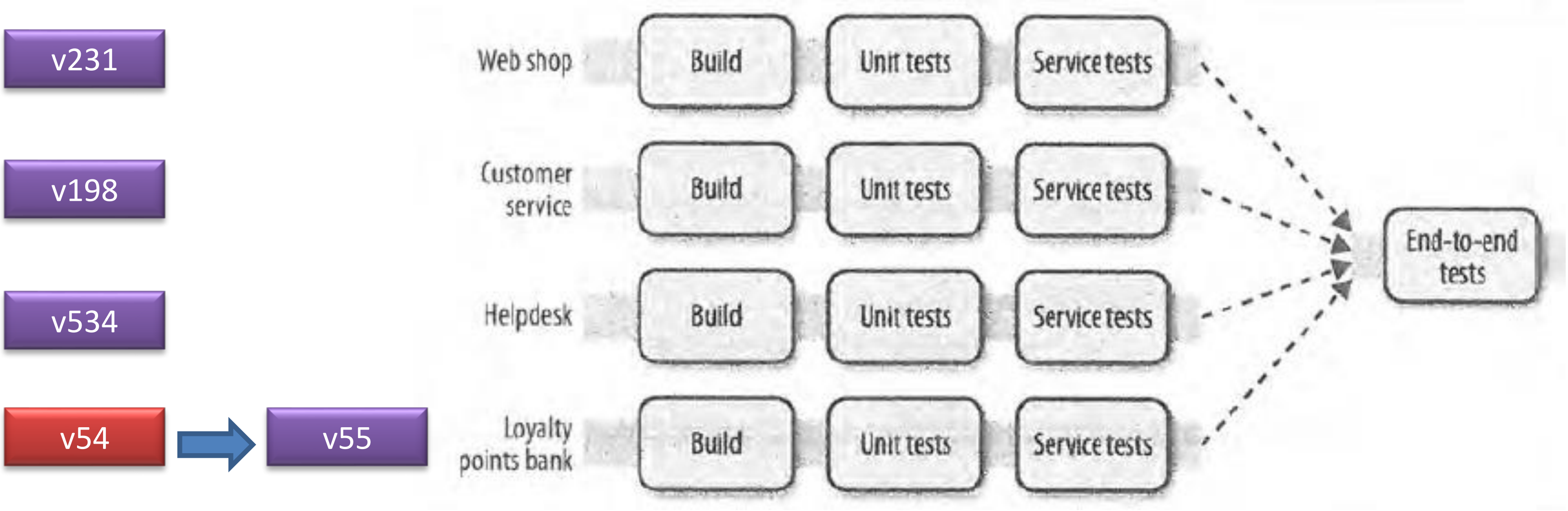
End-to-end tests

Easy? Edit composefile, and ‘docker stack deploy’. Right?

AARHUS UNIVERSITET

Motivation

- **Co-existence of versions of services**
  - During deployment (horizontally scaled)
  - Assumptions on version of *interfaces between services*



Rollout takes time

- Full Deployment phases
  - Preparation
  - Rollout N applications
  - Cleanup
- Each Application Rollout, again contains phases
  - Preparation
  - Drain Stop new request, await pending processed
  - Update Deploy new application
  - Startup Loading, Warm up caches, state resynch
- That is, **multiple versions co-exist in production ☹**

Simple Example

- EcoSense MongoDB replica set disk size expansion
  - (ended at 3 x 6 TeraBytes)

  - For three instances do:
    - **Drain:** Shutdown db server
      - If Primary, await new primary promoted
    - LVM magic to extend size (manual process, takes 1-3 minutes)
    - **Update:** Start db server
    - **Startup:** Await replica status shows server is ‘uptodate’
      - (Bass ‘State resynch’ finished)

Issues

- Versions differ on several aspects
  - (Rest/Web) APIs
    - (Nygard §14 / We will return in second course ‘versioning’)
    - *Robustness Principle*: Be conservative in what you do, be liberal in what you accept from others. [Jon Postel]
  - Database schemas
  - Web assets

Database Migration: SQL

- *Shim:* Bit of code that helps join the old and new version of application.

Version v54

Version v55

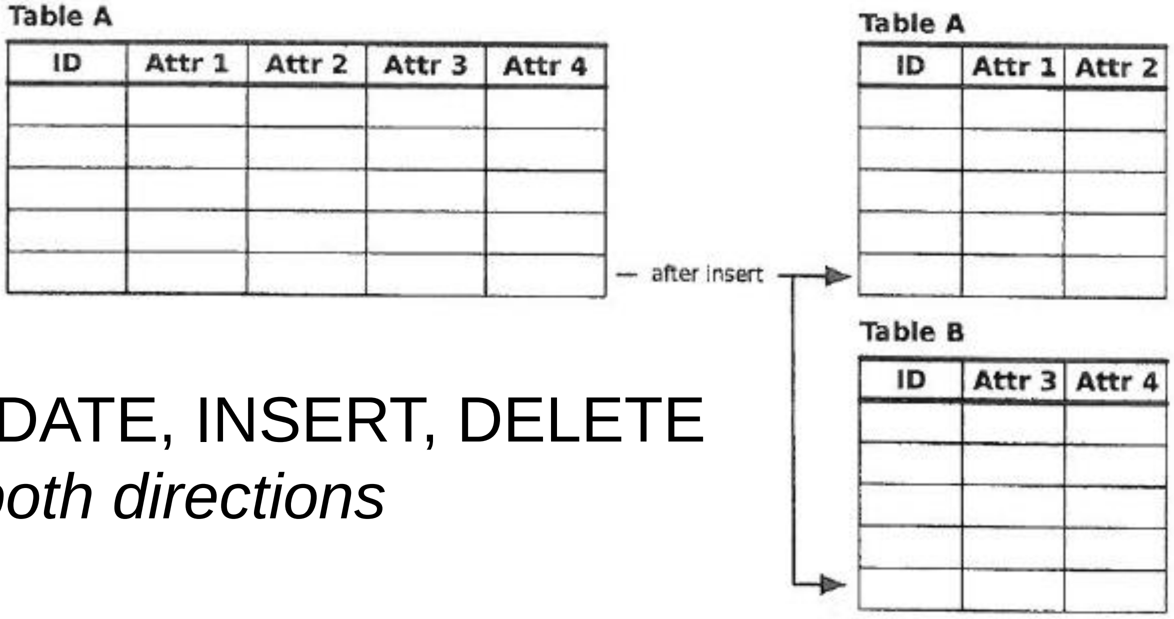


Table A

ID	Attr 1	Attr 2	Attr 3	Attr 4

Table A

ID	Attr 1	Attr 2

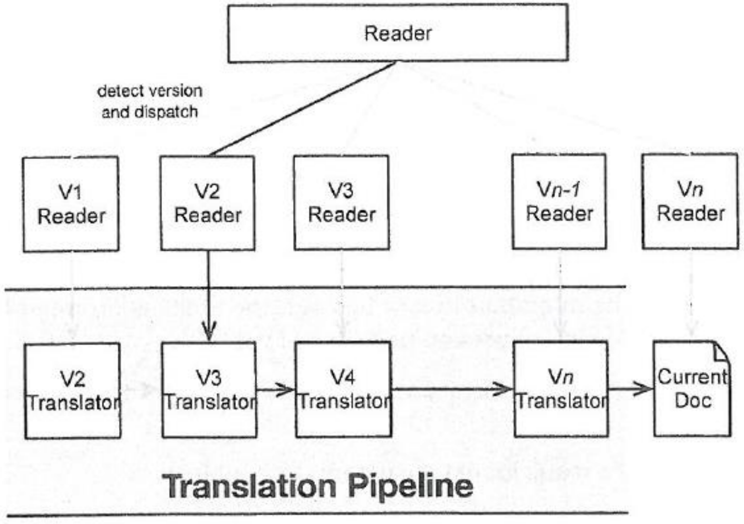
Table B

ID	Attr 3	Attr 4

- Triggers on UPDATE, INSERT, DELETE that update *in both directions*

Migration: NoSQL

- Schema-less? Hah! Applications expect schemas!
- ***Translation pipeline:*** Code the reader so it can read all versions ever made.
  - *Corollary: Always include version identity in documents!*
  - Keep old data around to test the translations
    - v2 -> v3; v3 –> v4; etc.
- Liability:
  - Deep pipeline (slow)
  - Cumbersome code
  - Db contents highly mixed

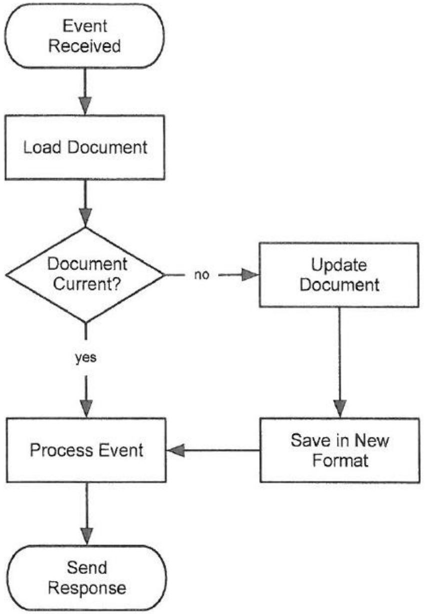


Migration

- Approach two
- ***Migration routine:*** Run a special schema lifting process during deployment, *after* all instances have been updated
- Liability
  - Big data means *hours* spent on migration => **Planned downtime**
- Why instance update *before* migration routine?
  - Consider *old version instance* reads from *new format* db
    - Cascading failure…

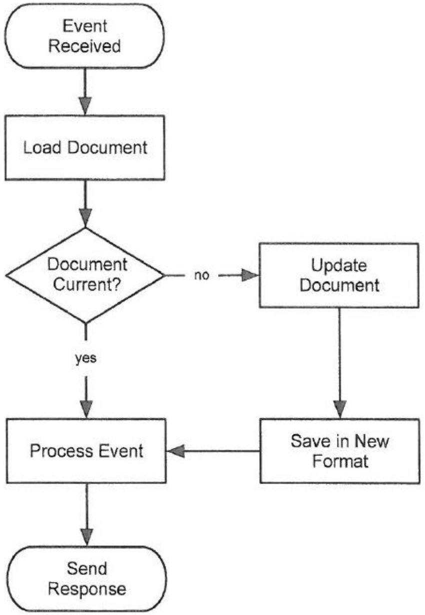
Migration

- Approach three (Nygard’s favorite)
- ***Trickle, then batch:*** Initiate migration as they are touched (conditional code in the application code). *After some time* (at least all instances are updated), perform a *batch migration routine* on all documents not converted. *Finally,* conditional code can be removed (final update).



Migration

- Benefits
  - Migration time is amortized (no downtime)
  - Batch can run concurrently in production
  - Only one version in DB (eventually ☺)
  - Clean code (no translation pipeline; no conditional code; eventually ☺)



- Liabilities
  - Complex deployment setup
    - Two application updates for all apps that read that kind of document
      - v_n: with trickle code; v_n+1: trickle code removed

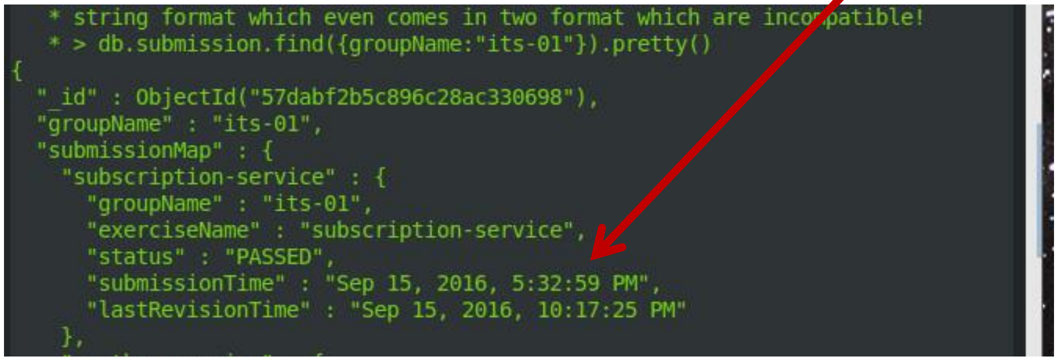
Example

- From my own, small-scale, backyard

- Crunch3: Predecessor for ‘your Crunch’
  - – Date’s were generated by GSON library
    - And they s…., as
      - – Plain text in **two different incompatible formats**
      - – **UTC or EST or ?**

- Migrate to ISO8601

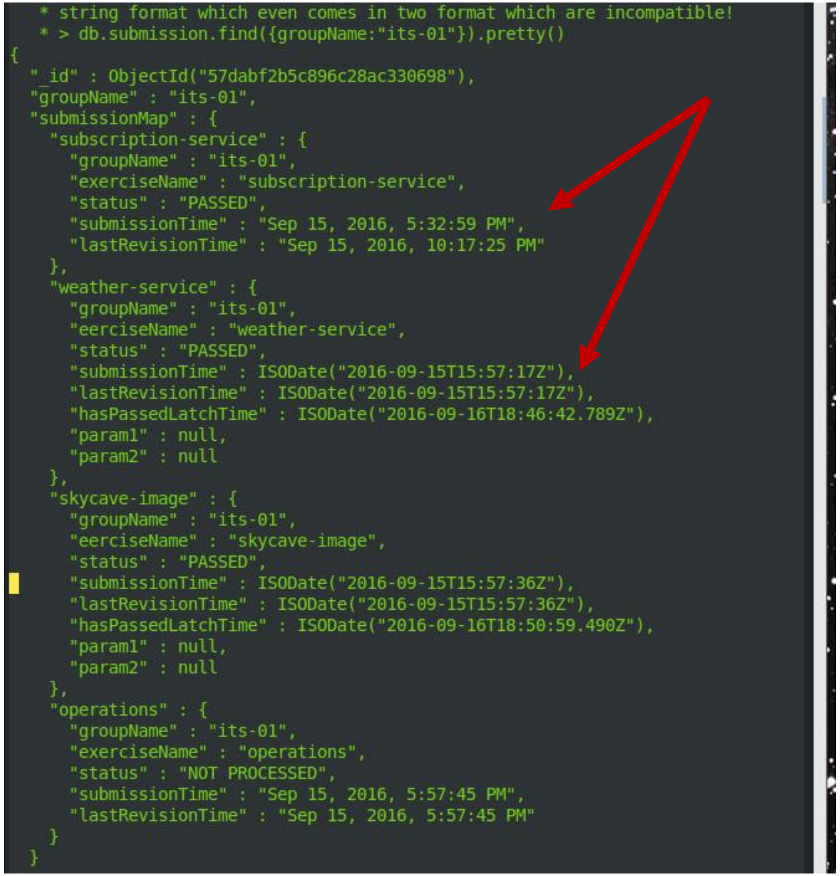
- *Always use ISO8601 strings !!!*



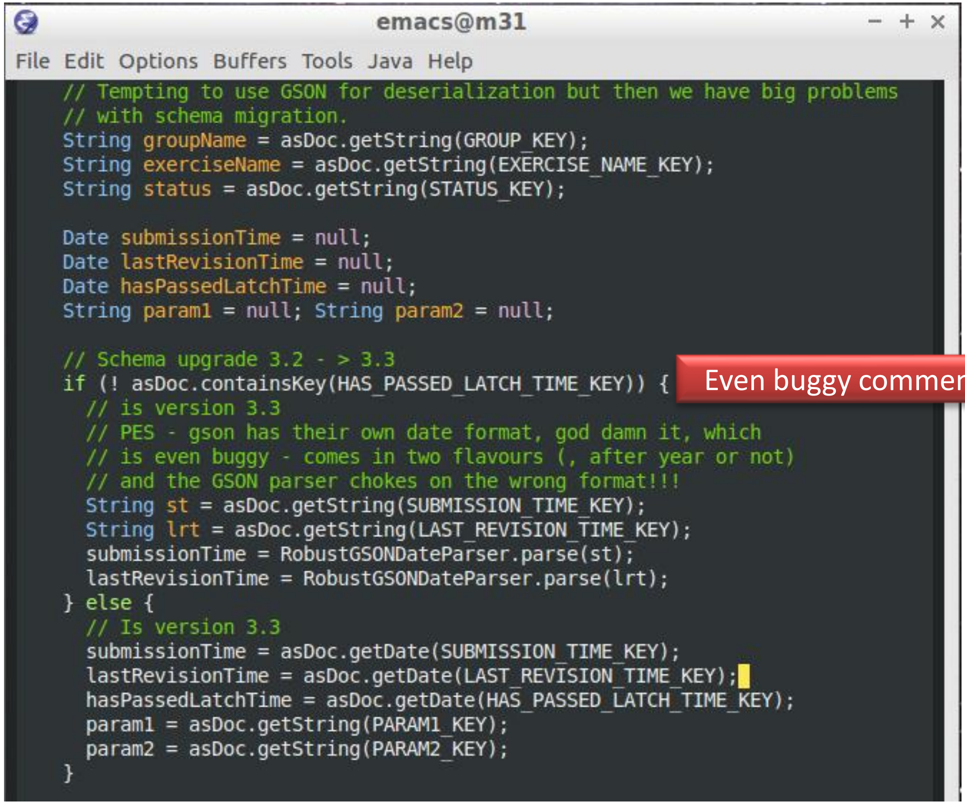
```
   * string format which even comes in two format which are incompatible!
   * > db.submission.find({groupName:"its-01"}).pretty()
{
  "_id" : ObjectId("57dabf2b5c896c28ac330698"),
  "groupName" : "its-01",
  "submissionMap" : {
    "subscription-service" : {
      "groupName" : "its-01",
      "exerciseName" : "subscription-service",
      "status" : "PASSED",
      "submissionTime" : "Sep 15, 2016, 5:32:59 PM",
      "lastRevisionTime" : "Sep 15, 2016, 10:17:25 PM"
    },
```

AARHUS UNIVERSITET

Fear of ‘Oh-No’ seconds…



```
   * string format which even comes in two format which are incompatible!
   * > db.submission.find({groupName:"its-01"}).pretty()
{
  "_id" : ObjectId("57dabf2b5c896c28ac330698"),
  "groupName" : "its-01",
  "submissionMap" : {
    "subscription-service" : {
      "groupName" : "its-01",
      "exerciseName" : "subscription-service",
      "status" : "PASSED",
      "submissionTime" : "Sep 15, 2016, 5:32:59 PM",
      "lastRevisionTime" : "Sep 15, 2016, 10:17:25 PM"
    },
    "weather-service" : {
      "groupName" : "its-01",
      "eerciseName" : "weather-service",
      "status" : "PASSED",
      "submissionTime" : ISODate("2016-09-15T15:57:17Z"),
      "lastRevisionTime" : ISODate("2016-09-15T15:57:17Z"),
      "hasPassedLatchTime" : ISODate("2016-09-16T18:46:42.789Z"),
      "param1" : null,
      "param2" : null
    },
    "skycave-image" : {
      "groupName" : "its-01",
      "eerciseName" : "skycave-image",
      "status" : "PASSED",
      "submissionTime" : ISODate("2016-09-15T15:57:36Z"),
      "lastRevisionTime" : ISODate("2016-09-15T15:57:36Z"),
      "hasPassedLatchTime" : ISODate("2016-09-16T18:50:59.490Z"),
      "param1" : null,
      "param2" : null
    },
    "operations" : {
      "groupName" : "its-01",
      "exerciseName" : "operations",
      "status" : "NOT PROCESSED",
      "submissionTime" : "Sep 15, 2016, 5:57:45 PM",
      "lastRevisionTime" : "Sep 15, 2016, 5:57:45 PM"
    }
  }
```



```java
  // Tempting to use GSON for deserialization but then we have big problems
  // with schema migration.
  String groupName = asDoc.getString(GROUP_KEY);
  String exerciseName = asDoc.getString(EXERCISE_NAME_KEY);
  String status = asDoc.getString(STATUS_KEY);

  Date submissionTime = null;
  Date lastRevisionTime = null;
  Date hasPassedLatchTime = null;
  String param1 = null; String param2 = null;

  // Schema upgrade 3.2 - > 3.3
  if (! asDoc.containsKey(HAS_PASSED_LATCH_TIME_KEY)) {
    // is version 3.3
    // PES - gson has their own date format, god damn it, which
    // is even buggy - comes in two flavours (, after year or not)
    // and the GSON parser chokes on the wrong format!!!
    String st = asDoc.getString(SUBMISSION_TIME_KEY);
    String lrt = asDoc.getString(LAST_REVISION_TIME_KEY);
    submissionTime = RobustGSONDateParser.parse(st);
    lastRevisionTime = RobustGSONDateParser.parse(lrt);
  } else {
    // Is version 3.3
    submissionTime = asDoc.getDate(SUBMISSION_TIME_KEY);
    lastRevisionTime = asDoc.getDate(LAST_REVISION_TIME_KEY);
    hasPassedLatchTime = asDoc.getDate(HAS_PASSED_LATCH_TIME_KEY);
    param1 = asDoc.getString(PARAM1_KEY);
    param2 = asDoc.getString(PARAM2_KEY);
  }
```

Even buggy comment ☹

Trickle, without the batch ☺

Web assets

- The issue:
  - Web UI’s contains static assets (stylesheets, java script, images, …)
  - Web browsers cache these assets
    - Thank god, so server load is minimized
  - *Usually there is a hard coupling* between UI elements and server side
- So, when we update the server, we must force the browser to ‘bust the cache’ and fetch all assets afresh!

- *Tips and trick – see Nygard ☺*

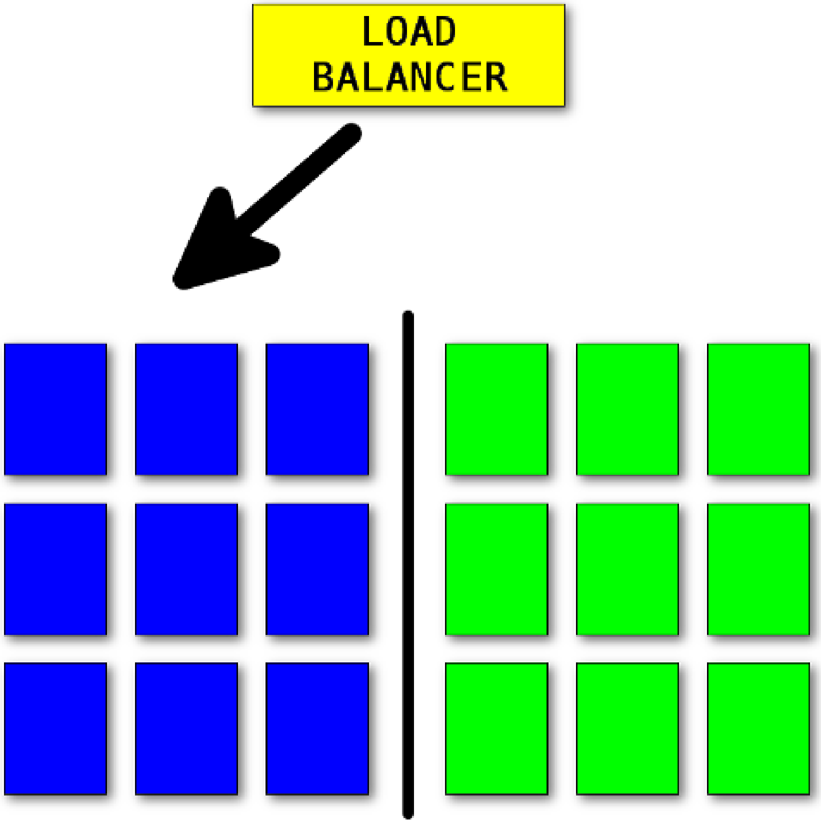
AARHUS UNIVERSITET

Rollout Techniques

Birds, colors, and more…

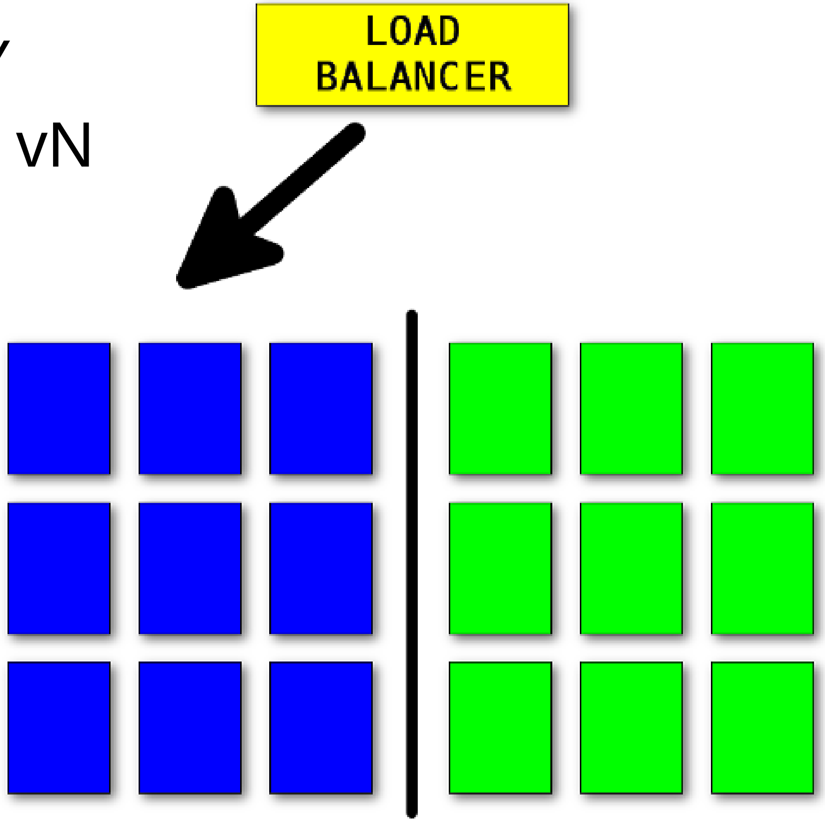
Blue/Green Deployments

- Algorithm
  - Deploy v N+1
  - Smoke test
  - Swich load
  - Monitor!
- Benefit
  - Rollback is easy
- Liability
  - 2x HW cost!
  - Common DB ???



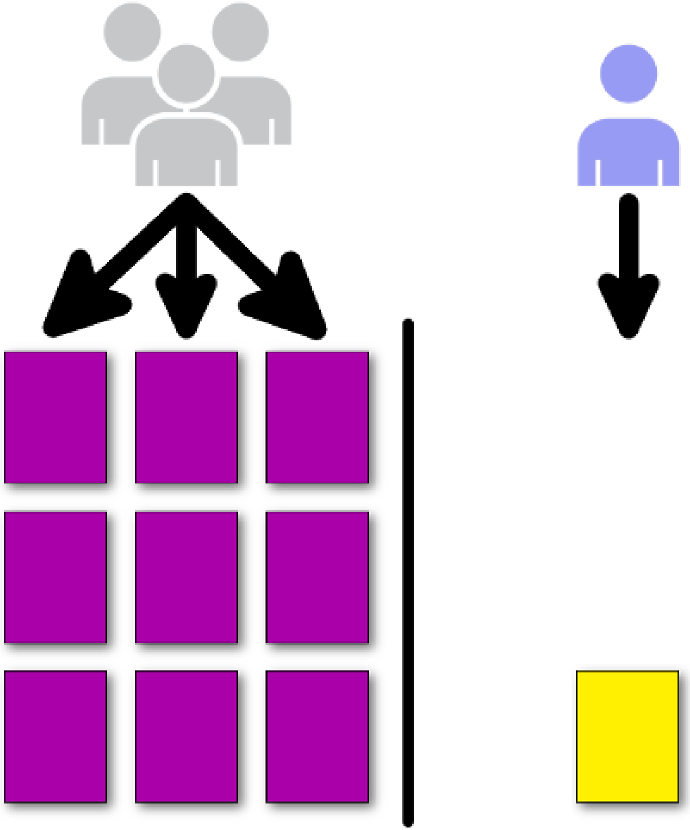
Release Vrs Deploy

- Blue/Green embody the distinction between *release* and *deploy*
- Deploy vN+1 but Release is vN



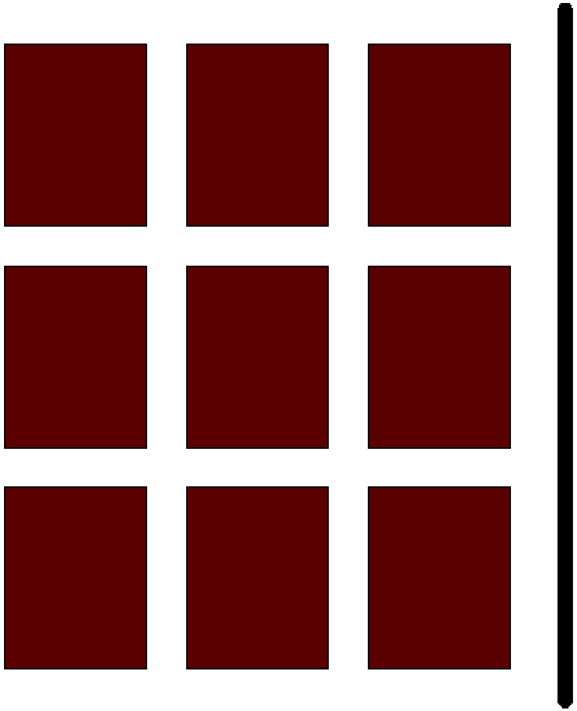
Canary Deployment

- Algorithm
  - Direct d% traffic to version n+1
  - Monitor
    - If OK, direct (d+20)% traffic to it
- Benefits
  - Scientific experimentation!
  - Lower HW costs
  - Easy rollback
- Liabilities
  - Complex setup



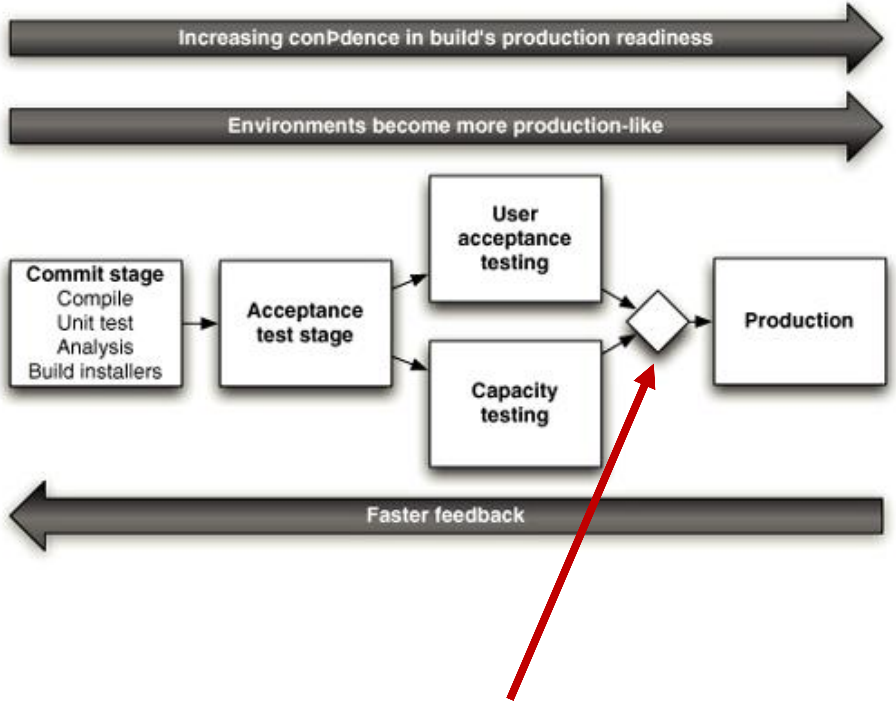
Rolling Deployment

- Algorithm
  - One-by-one
- Benefits
  - Constant HW
- Liability
  - Complexity



Humble:

- Commit stage
  - Asserts technical level
    - Build + Unit tests
- Accept test stage
  - System works at functional and non-functional level
    - *Meets demand of users*
- Manual test stages
  - UAT
- Release stage
  - Deliver system to users



A manual go/no-go ?

Monitoring

- Rollout depends deeply on monitoring…
- Overview the *machines* and overview the *services*
- We will return to that in course two…

Nygard

- Nygard provides more detail in the *rollout* phase than Newman ☺
  - *”Canary release, yes, but hey – what if the canaries migrates the shared database schema ???”*
  - *”What if requests gets load-balanced to (new, old, new, old, new) version instances?*

- The rollout phases
  - Drain, update, startup
  - Important: *monitor after each phase*

Sessions

- If you do blue/green deployments
  - *What happens to open connections when the load balancer switches from the blue to the green cluster?*
- If you do canary or rolling deployments
  - *What happens if you hit an old version instance that tries to access new database schema documents?*

Session Stickiness

- You must avoid hitting new/old instances as they co-exists
  - *Sticky sessions:* All requests from a given session is routed to the same server
    - *Or perhaps 'same version of server'*

- Alternatives
  - Session database: Session data is stored in session database, shared by all servers
  - Client sessions (browser cookies): Session data is stored in client, sent with each request

Co-existing Versions

- If you have (old/new) version applications co-existing
  - Which is a consequence of no-down-time continuous delivery ☺
- Then you *have to ensure session stickiness*
  - To avoid (new) version request put an Order object (new format) into the DB, and next a (old) version request read it expecting the old format
- **But**
  - **What is the big issue with session stickiness?**

Cleanup

- Monitor hard ☺

- If everything looks OK, then ***cleanup***
  - Remove shims, drop tables, initiate *then-batch* of trickle-then-batch, remove trickle code (new version, then new ‘delivery’)

Summary

- Nygard digs that one step deeper into *real issues !*
- *Lots of code near techniques must be employed in order to make CD work in practice…*
  - *Characterization of the process*
    - ***Preparation, Rollout N, Cleanup***
      - ***Preparation, Drain, Update, Startup***
  - *Tactics for state migration*
    - *Translation pipeline, Migration process, Trickle-then-batch*
  - *Issues in Deployment Rollout*
    - *(Temporary?) Session stickiness* *a bit vague there…*