CI Pipeline with Docker 2015-02-27

Juho Mäkinen, Technical Operations, Unity Technologies Finland
http://www.juhonkoti.net
http://github.com/garo

Overview

1. Scale on how we use Docker
2. Overview on the production infrastructure
3. Continuous Integration pipeline with Jenkins
4. Production deployment

unity

History on our Docker usage

- Started writing new deployment platform with Chef, fall 2013
- Started experimenting with Docker 0.7, Jan 2014
- First production tests with mongodb and Docker, Mar 2014
- Nearly every backend service migrated into Docker by end of May 2014
- Started Orbit development on Jul 2014
- First production usage with Orbit around Nov 2014

Some statistics

- 31130 Docker layers
- 1950 commits on the Chef deployment repository
- 540 GiB of Docker containers in our private repository
- 96 distinct different container names
- Zero production downtime due to Docker itself

unity

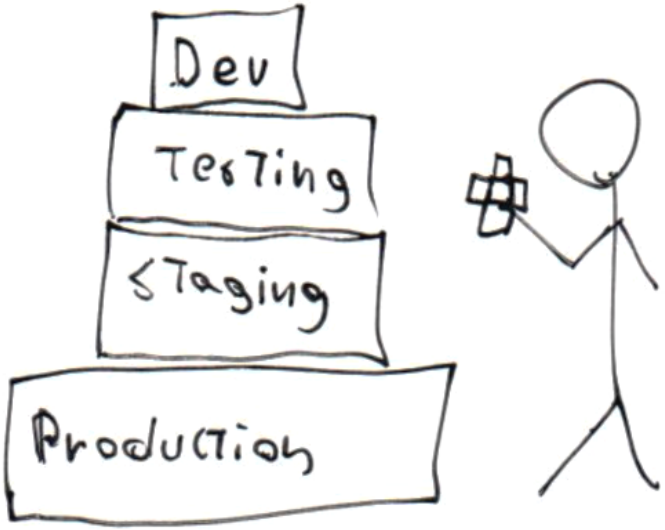
List of software deployed in Docker containers

1. Bifrost
2. Cassandra
3. nginx
4. Datastax Opscenter
5. Elasticsearch
6. Etcd
7. Graphite-beacon
8. Graphite
9. haproxy2statsd
10. ice
11. jenkins
12. kafka
13. memcached
14. mongodb
15. mongos
16. nsqlookupd
17. ocular
18. rabbitmq
19. redis
20. Docker registry
21. Rundeck
22. s3-sinopia
23. s3ninja
24. scribe2kafka
25. scribed
26. secor
27. squid
28. statsd
29. statsite
30. unifi-nvr
31. wowza
32. zabbix
33. zookeeper

Plus around 40 internally developed software components

unity

Four different environments



1. Development environment

   Minimum set of services, no clustering, everything in default port

2. Testing environment
3. Staging environment
4. Production environment

   Full set of services with clustering. Identical host names and port numbers.

   Only minor differences which are mostly limited to visible URLs for web services.

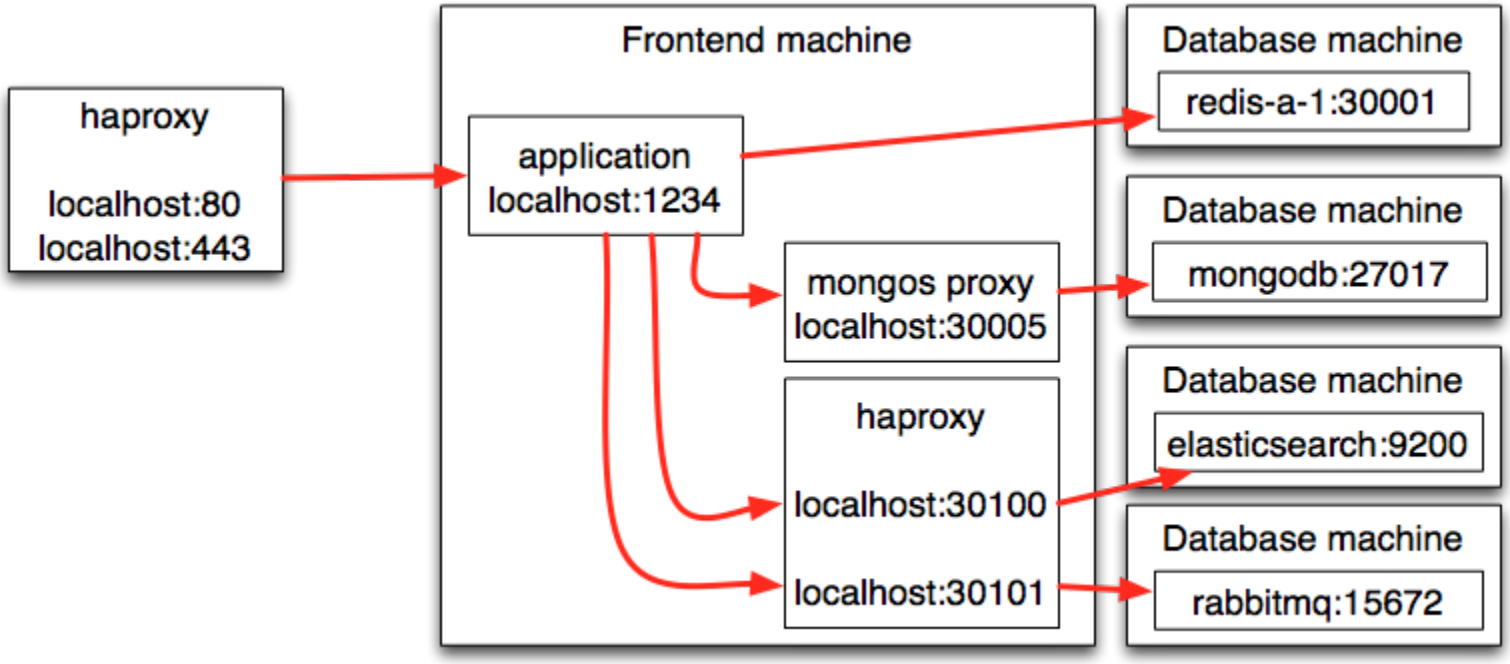
unity

Continuous Integration workflow

1. Developer runs minimum set of required backend services in Docker containers, provided in a single Vagrant virtual machine
2. Developer runs full test suite in his laptop (make test)
3. Commit and push to git server. Jenkins is triggered and build is started
4. If build success the Docker container is uploaded to internal registry
5. Developer can commit the new build into production using our in-house tool Orbit

unity

How does the production look?



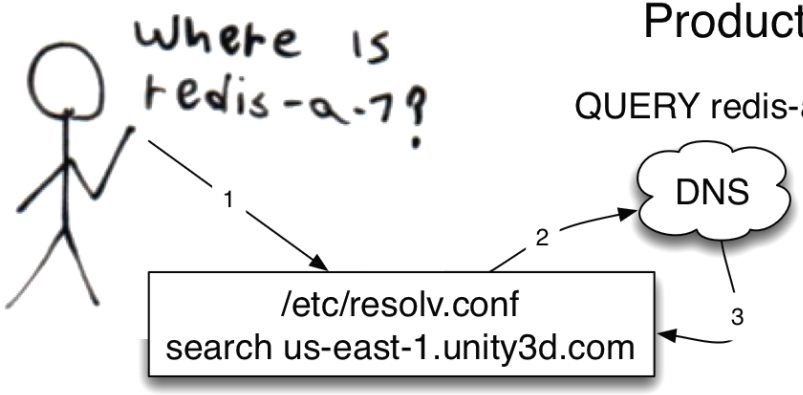
Service naming

- Each service has a name which is unique within a deployment group (AWS region). Example names: mongodb-a-1, mongodb-a-2, mongodb-a-3, redis-b-1
- Each physical machine has a separated name. Example names: db01-1, db02-3
- Each physical machine has an A record
  - “db01-1.us-east-1.unity3d.com A 10.0.0.1”
- Each service has an CNAME record:
  - “mongodb-a-1 CNAME db01-1.us-east-1.unity3d.com”

Service naming (cont)

- In testing/staging/production environments the applications use the service name **without the full domain** in their configurations:
  - client = new Redis([“redis-a-1:30001”, “redis-a-2:30001”])
- In testing/staging the redis (in this example) services are inside Docker hosts which use the “-h=redis-a-1” -option to bind a host name for the container.
  - dnsmasq script-fu is used so that the application can do DNS query for “redis-a-1” and get the container ip as a response.
- In production the frontend machine has **/etc/resolv.conf** configured with “**search us-east-1.unity3d.com”** so that the dns finds the redis-a-1.us-east-1.unity3d.com CNAME which then points out to the A record, thus finally getting the ip.

Production

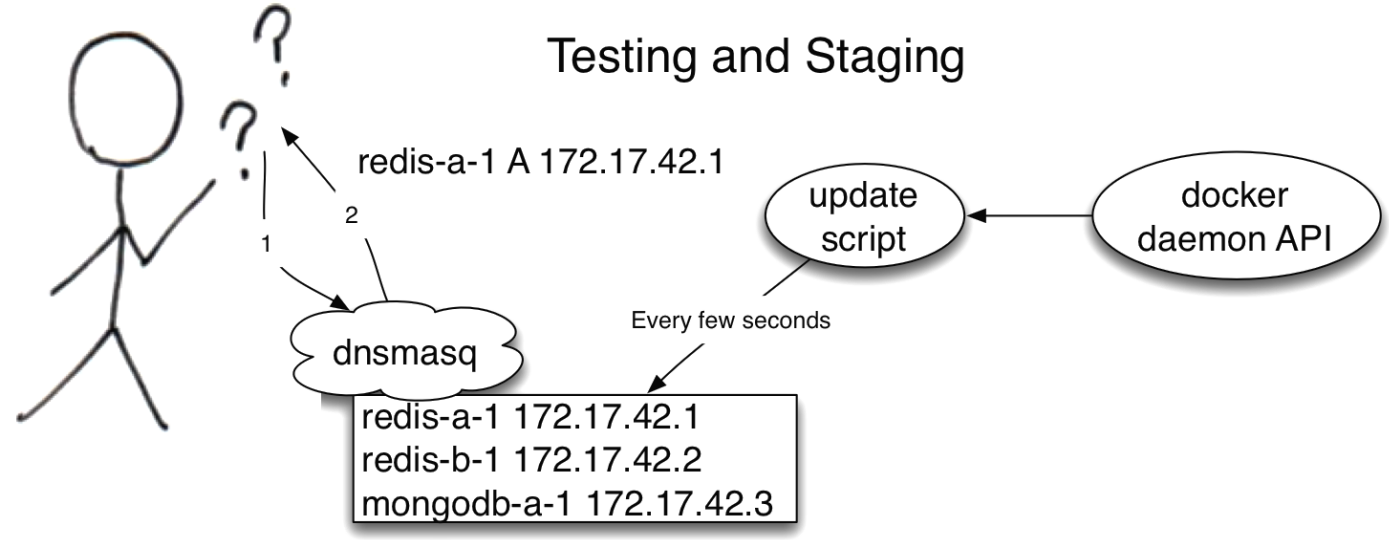


QUERY redis-a-1.us-east-1.unity3d.com

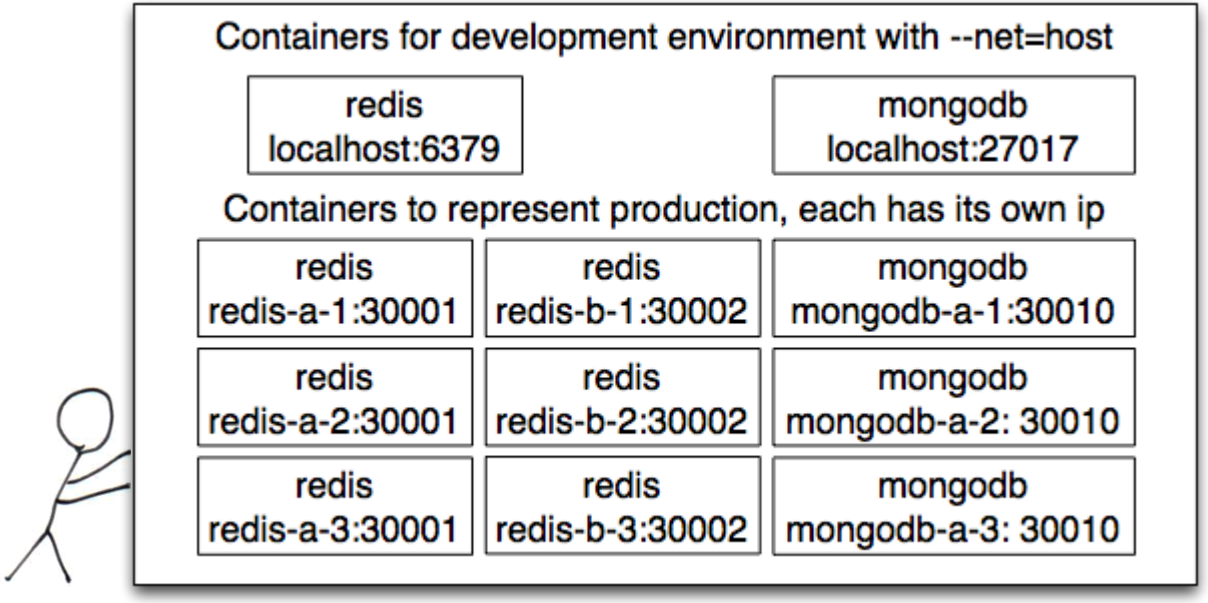
redis-a-1.us-east-1.unity3d.com CNAME db01.us-east-1.unity3d.com

db01-1.us-east-1.unity3d.com A 10.0.0.1

Testing and Staging



Testing machine configuration



- Contains two set of services:
  - Default environment, default ports running in --net=host container mode
  - Testing environment, same port numbers as in production, running in normal --net=bridge mode (the default for Docker)

unity

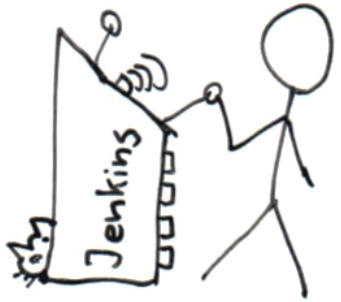
What do we want to test?

- Statistical code analysis.
- Unit tests. Usually without database access as they are mocked, except on database access code which uses real databases. Needs to be really fast test suite, usually under 10 seconds.
  - Also check code coverage.
- Acceptance tests which in our case usually tests big parts of the system together.
- Integration and black box testing. Start the software and do (HTTP) queries to its visible interfaces.
- Settings: Verify that software can connect to all required databases with full clustering and high availability support.

Local development machine, ENV=development

- Developer uses a Vagrant virtual machine which starts a minimum amount of required services in Docker container.
- Minimum amount means that there’s no clustering and no sharding, so just one redis, one mongodb, one cassandra etc.
- All services are on their default ports and they are NAT’ed from the virtual machine into the host computer.
- Depending on project developer can run the actual application either directly in the host machine or in the VM

Jenkins build flow



1. docker build -t $docker_image .
2. docker run -i --net=host -e ENV=development $docker_image make test-unit
3. docker run -i --net=host -e ENV=development $docker_image make test-unit-coverage
4. docker run -i --net=host -e ENV=development $docker_image make test-acceptance

5. docker run --name $JOB_NAME-net /redir.sh $ETH0_IP 30001:30001 30002:30002
6. docker run --net=”container:$JOB_NAME-net” -e ENV=testing make load-testing-data
7. docker run --net=”container:$JOB_NAME-net” -e ENV=testing make node app.js
8. docker run --net=”container:$JOB_NAME-net” -e ENV=testing make test-integration

9. docker push $docker_image

unity

Closer look on unit tests

docker run -i --net=host -e ENV=development $docker_image make test-unit

- Uses the --net=host and because ENV=development is used, it tries to find the databases from localhost
    - Redis at localhost:6379
    - Mongodb at localhost:27017
- This is the same what the developer has in his own laptop
- the Makefile contains steps which loads test data into the databases in the beginning of each test suite run

Closer look on integration tests

docker run --name $JOB_NAME-net /redir.sh $ETH0_IP 30005:30005 30100:30100

1. This creates a container which is used as a base network for the following steps.

2. The redis.sh uses “redir” command to redirects few required ports from the “localhost” of this container into outside of the container, which in this case is the eth0 of the host machine. This emulates for example mongos proxies which exists in every frontend where the application is executed.

Closer look on integration tests (cont)

1. docker run --name $JOB_NAME-net /redir.sh $ETH0_IP 30005:30005 30100:30100
2. docker run --net=”container:$JOB_NAME-net” -e ENV=testing make load-testing-data
3. docker run --net=”container:$JOB_NAME-net” -e ENV=testing make node app.js
    a. The 2nd and 3rd command creates container which reuses the networking stack from the first container.
    b. The “node app.js” starts the to-be-tested application to listen in localhost:1234 for incoming HTTP requests, just like it would do in production
4. docker run --net=”container:$JOB_NAME-net” -e ENV=testing make test-integration
    a. The integration test can now do HTTP queries to localhost:1234 and test the application and then verify that the application updated databases like it should

unity

Purging and squashing docker images

Because most components needs full build environment it creates an unnecessary security risk in production.

We’re experimenting with a purge script which is executed on the built container after unit tests are done and before integration tests start.

This script removes everything which is not a library, thus the container is left with no unnecessary tools which an attacker could use for example to download a rootkit.

The resulting container is squashed with docker-squash which squashes all layers into one.

Deploying with Orbit

Orbit is an in-house developed open source tool to deploy containers into a set of machines.

Features:

- Declare machine configurations with revision controlled json files
- Allow developers to use a cli tool to upgrade a container version without revision control. (separated audit and history logging)
- Agent in all machines which uses the local Docker API to manage containers.
- Configuration stored in etcd ensemble plus RabbitMQ for real time communication.
- Support configuring haproxies and populate server lists automatically.
- Support sending metrics to Zabbix for monitoring
- Support HTTP and TCP health checks
- Used in production and still in active development.
- Written in Go, so only single static binary needs to be installed on each machine.

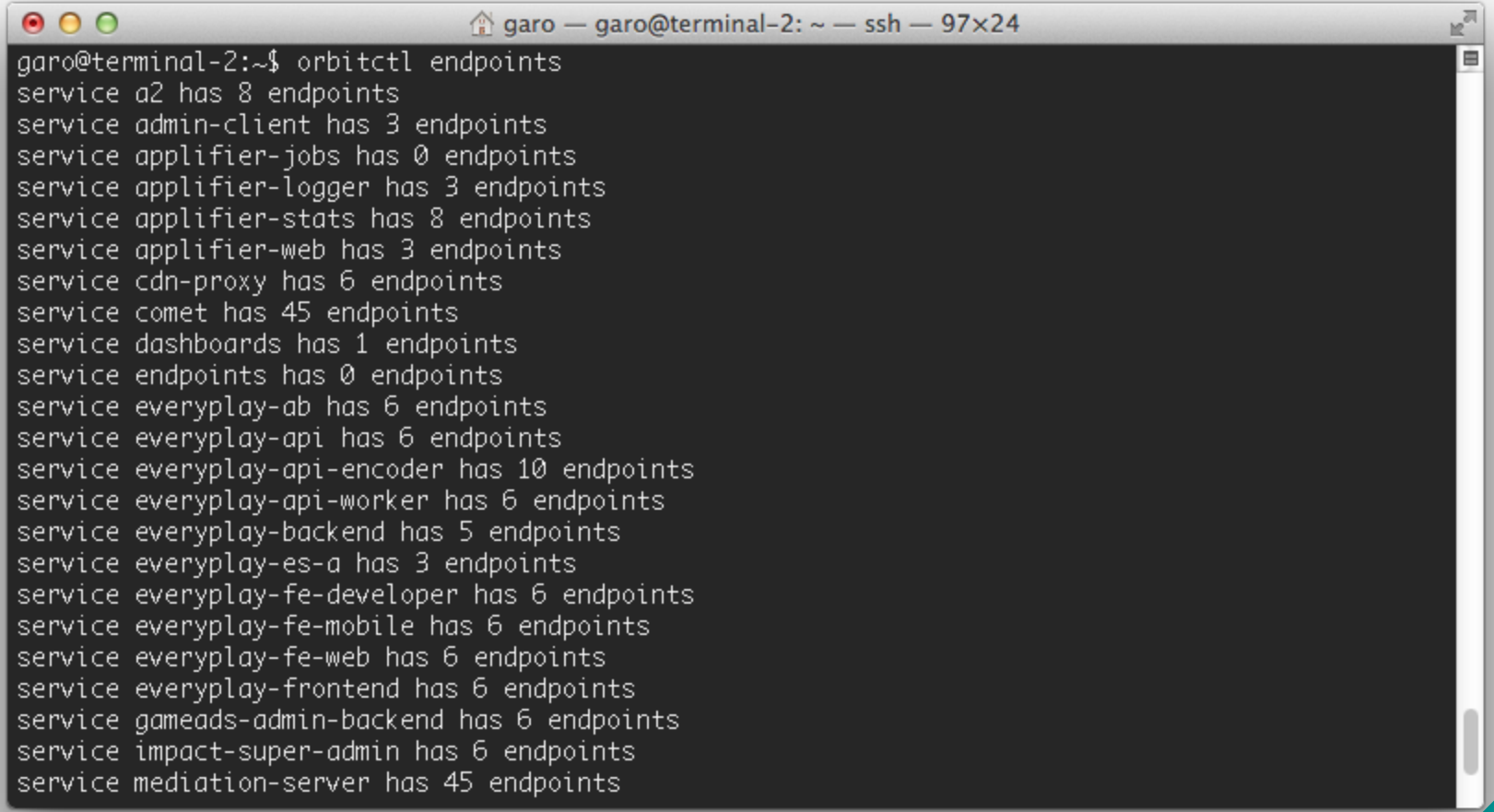
Deploying with Orbit

Really fast deployments: committing new version into production takes 60-90 seconds on average.

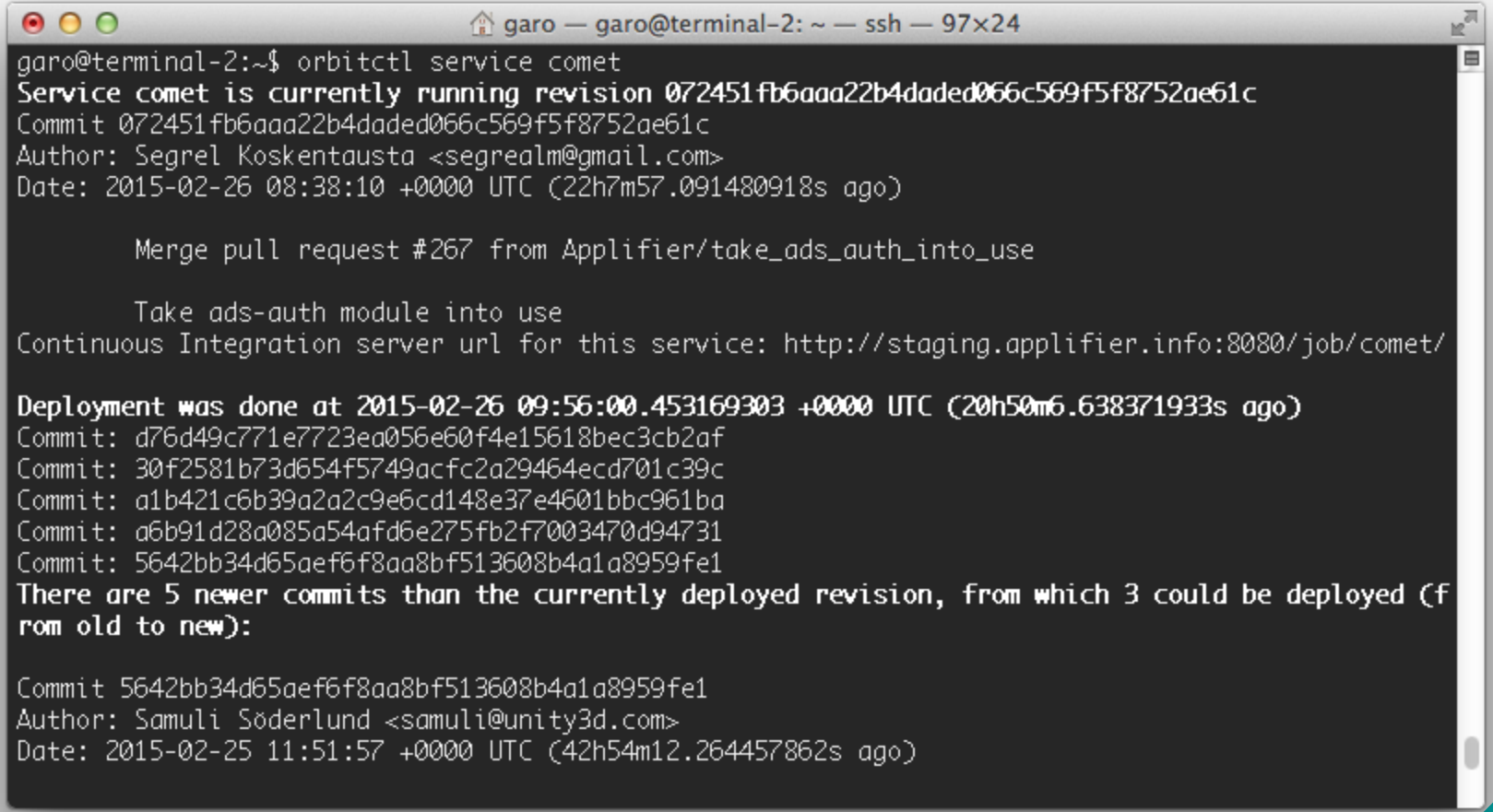
Rollback even faster: usually less than 30 seconds as the previous container is still cached in frontend machines.

Interleaves individual container restarts so that entire cluster isn’t restarted at exact same second (unless especially requested)

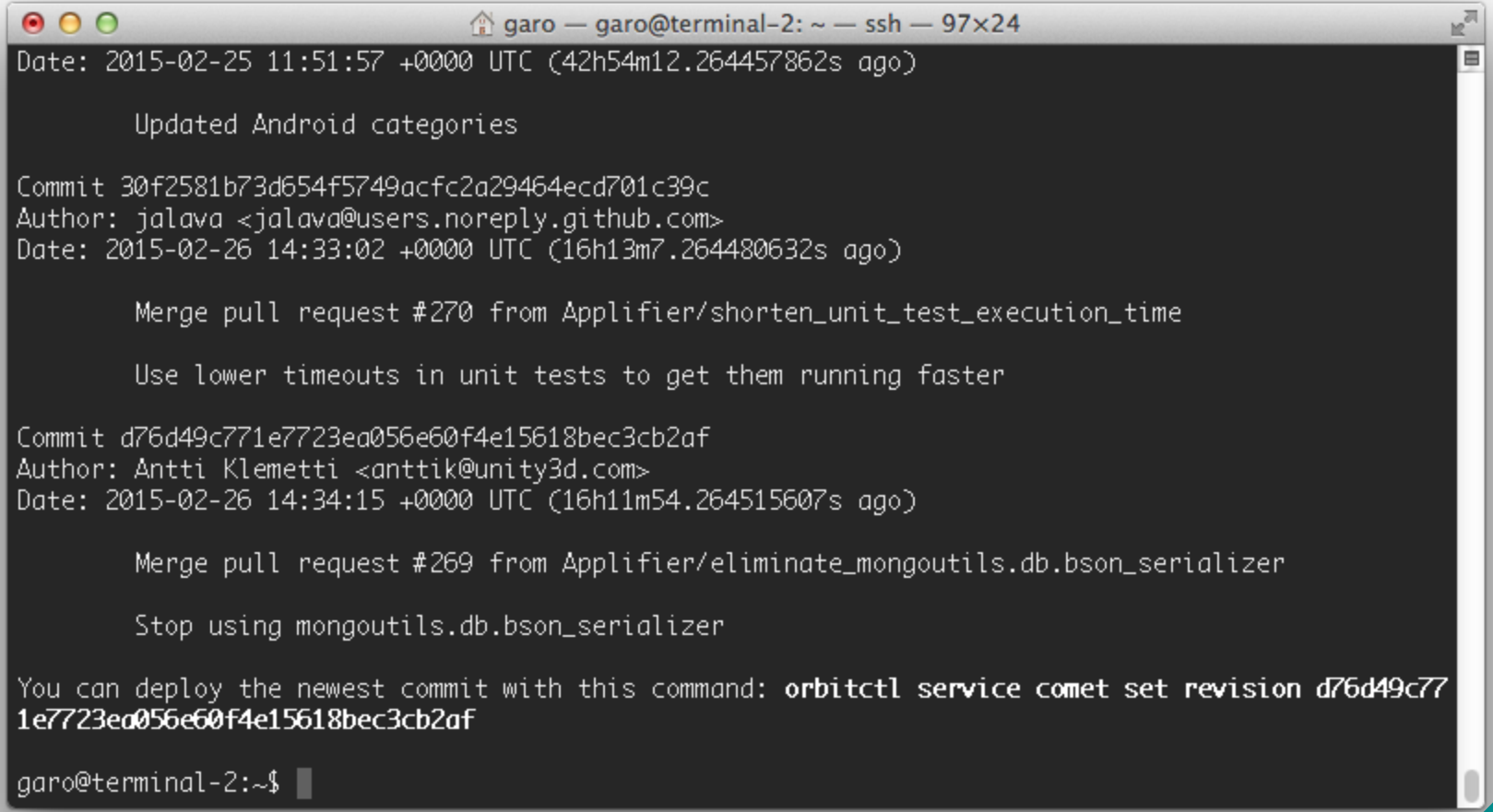
unity



unity



unity



unity

Questions?