Markus Juopperi

Deployment automation with ChatOps and Ansible

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

9.5.2017

Helsinki Metropolia University of Applied Sciences

Author Title Number of Pages Date	Markus Juopperi Deployment automation with ChatOps and Ansible 22 pages 09 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructors	Markku Nuutinen, Principal Lecturer Tommi Berg, Head of Service Operations

This Bachelor's thesis explores different solutions for automating deployments and other common production tasks. The focus is on the viability of ChatOps as a tool for deployment automation and how it can be a useful tool for performing complex tasks like blue-green deployments.

Much of this thesis examines how a deployment process using ChatOps was developed for an existing application. It starts with a description of the problems the previous deployment system had and how ChatOps was chosen as the solution to solve them.

Ansible is used as a tool for infrastructure automation. This is necessary to perform zero-downtime deployments using so called blue-green deployments, where instead of updating existing servers, completely new environments are created to replace the previous version.

The findings and examples are based on real world production use of the implemented deployment system over a period of close to six months. In the end, ChatOps proved to be a viable solution even for a small team without any dedicated DevOps personnel. The main benefit of this way of managing deployments is the ability to automate the complex tasks while retaining control of when the tasks are performed. After the initial development work, functionality is easily extended provided that code is kept well structured.

Keywords	deployment automation, ChatOps, Ansible

Tekijä Otsikko Sivumäärä Aika	Markus Juopperi Tuotantoonsiirron automatisointi ChatOpsin ja Ansiblen avulla 22 sivua 9.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	Yliopettaja Markku Nuutinen Head of Service Operations Tommi Berg

Insinöörityön tarkoituksena oli vertailla vaihtoehtoja verkkosovellusten ylläpitotehtävien automatisointiin ja toteuttaa tuotantoonsiirron automatisointi olemassa olevaa sovellusta varten. Työssä perehdyttiin ChatOps-toimintamallin soveltuvuuteen tähän tehtävään ja sen hyötyihin toteutettaessa monimutkaisia toimintoja, kuten sini-vihreäpäivityksiä.

Työ aloitettiin perehtymällä alkuperäiseen toimintamalliin ja sen kohtaamiin ongelmiin. Ongelmista merkittävimpiä olivat palvelukatkokset tuotantoonsiirtojen aikana ja olemassa olevien työkalujen huono käytettävyys. Työssä kehitettiin uusia automaattisia toimintoja käyttäen ChatOps-toimintamallia ja arvioitiin sen soveltuvuutta näiden ongelmien ratkaisemiseen. ChatOps-toimintamallissa erilaisia ylläpitotehtäviä tehdään komentamalla robotteja keskustelupalvelun kautta sen sijaan, että ohjelmoija itse ajavat komentoja komentoriviltä.

Työssä käytettiin Ansible-sovellusta palvelininfrastruktuurin hallinnan automatisointiin. Tämä oli tarpeen palvelukatkottomien sovelluspäivitysten aikaansaamiseksi niin sanottujen sini-vihreäpäivitysten avulla. Näissä päivityksissä luodaan täysin uusi ympäristö uudella sovellusversiolla korvaamaan edellisiä palvelimia sen sijaan, että sovellukset päivitettäisiin vanhoilla palvelimilla.

Insinöörityön löydökset ja esimerkit perustuvat tosielämän käyttökokemuksiin kehitetystä tuotantoonsiirtomenetelmästä vajaan puolen vuoden ajalta. ChatOpsin todettiin olevan toimiva vaihtoehto jopa pienelle kehitystiimille ilman ylläpitoon keskittynyttä henkilöstöä. Suurin etu tässä toimintamallissa on kyky automatisoida prosessin monimutkaisimmat osat pitäen silti kokonaisuuden hallinnan ja tehtävien ajoituksen kehittäjien hallinnassa. Perusominaisuuksien kehittämisen jälkeen jatkokehitys havaittiin helpoksi olettaen, että koodin rakenne pidetään hyvänä.

Avainsanat	Automaatio, ChatOps, Ansible

Contents

Abbreviations

API	Application Programming Interface
REST	Representational State Transfer
SSH	Secure Shell
CLI	Command Line Interface
AWS	Amazon Web Services
S3	Simple Storage Service
CDN	Content Delivery Network

1 Introduction

Frequent release of software, as new features are developed, is a trend that has been growing since agile methodologies became the standard for software development.

Continuous delivery has benefits for both users of the service as well as its developers. Fixes and small improvements to features can be made available as they are developed without the need for large releases. Small and frequent releases have the added benefit of reducing the risk introduced by deploying new versions. Deploying becomes a weekly or daily routine fresh in the minds of the team. It also encourages developing tooling to make the process as efficient as possible.

The goal of this thesis is to implement an automated process for deploying a complex system with minimal service interruption for its users. The service has network-connected IOT devices whose downtime should be kept to a minimum to ensure good user experience. Due to the nature of the service, complete automatization of the process is not desired. Instead, each deployment is initiated by the developers.

The thesis begins with a description of the service, the problems it faced, covers possible solutions and explains why ChatOps was the one chosen for this project. Much of this document goes over technical details on how a process like this can be implemented.

2 The Service

The service is built around IOT devices that require constant connection to backend servers. This poses a challenge in deployments since data loss is inevitable during service interruptions. Thus, the goal is to reduce normal service interruptions as well as to make the deployment process as robust as possible to prevent failed deployments.

3 Original deployment process

Deployment was originally done by hand with a script that would update the server package on a single server. In addition to being a tedious process requiring a significant amount of developer time, this way of deploying had other problems as well.

Upgrading the software on a server requires restarting the service. This caused any cameras connected to the server to timeout and reconnect to a new server. With many servers, it was common that a camera would have to reconnect multiple times during each deployment. This ended up creating a noticeable break in the history timeline.

The same problem would arise if updating packages on the servers would require a restart of the service. It also made updates risky due to a rollback being next to impossible if some updated package was causing problems.

Since the newly upgraded servers join the production environment immediately, there is no way to verify that the new version works in production beforehand. In case of problems, the server would have to be stopped and quickly rolled back to a previous working version. This was also done by hand and required developers to connect to the server with ssh.

With broken releases being rare, the way to correct problems was not always clear in the minds of everyone. There was no single tool for the whole process, and fixing problems required making manual changes on servers. Throughout documentation could fix some of the problems with this, but the cognitive load associated with deployment would persist.

4 Possible solutions

4.1 Continuous deployment

Automated tests runners like Jenkins make it possible create a workflow where new versions would be automatically deployed after all tests passed successfully. This method of deploying new versions of software is called continuous deployment [2, 266]. Because of the expectation of uninterrupted uptime, a completely automated system was deemed unfit for the service. Deployments would have to be initiated by the team to ensure all service interruptions would be controlled.

4.2 ChatOps

ChatOps is a recent addition to DevOps. It is a process where changes to a service can be made through a bot connected to a chat service the development team is using [12]. It is a compromise between full automation and manual deployment. The complex parts of deployment are automated while giving full control of when deployments are done to the development team.

A major benefit of ChatOps is visibility. Every action done is seen by everyone in the team. A history of actions is also preserved in chat. This gives a clear view to what has been deployed to production or is in testing to the whole team. [13]

A centralized way of making changes to the service encourages further automation. If deployments are done by telling a bot to do it, it makes sense to automate rollbacks and other actions as well.

5 Separating frontend and backend

Keeping interruptions to connections of the remote devices to a minimum is one of the main goals of this project. Devices are disconnected when the server applications re-start during deployment. Since most small changes are to the clients, including the web application, separating the frontend from the less regularly altered backend greatly reduces the downside of frequent deployments.

Services where the frontend is mostly static and consumes some API for dynamic content with JavaScript can be moved to a content delivery network. AWS offers CloudFront for this purpose.

5.1 CloudFront

CloudFront is the content delivery network offered by Amazon Web Services. It can serve both static and dynamic web content. Content is spread to Amazon's data centres around the world and can then be delivered from locations close to the requesting users. Static content is hosted in AWS S3. [3]

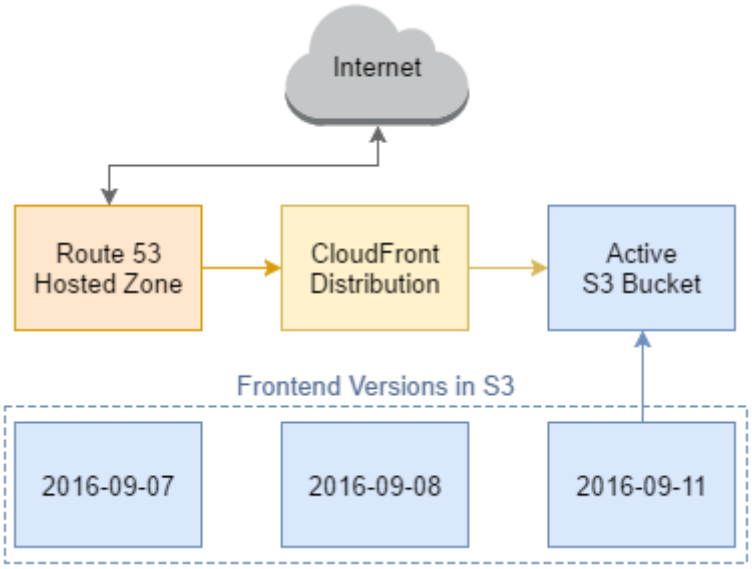

Figure 1. Frontend Architecture with AWS Route 53, CloudFront and S3

Figure 1 shows a simple hosting architecture for a static website. Traffic is directed through Route 53 DNS to CloudFront which serves files from a S3 bucket. Old versions can be kept in separate buckets and enabled by syncing them with the active bucket if the need arises.

5.2 S3

AWS S3 or Simple Storage System is Amazon's service for storing and serving files. It has a directory-like structure where the root directories, called buckets, contain files and directories. S3 buckets can be hosted as static web sites with an URL structure identical to the bucket paths. For this reason, buckets must have a globally unique

name across all AWS accounts, not just your own [5]. A good practise to avoid collisions is to prefix the buckets with the name of your product or service. [4]

5.3 Creating a CloudFront distribution

CloudFront can be configured with the web management service of AWS, AWS Console, or with programmable tools provided by Amazon and third parties. For creating a static distribution, the Console is a good tool since new distributions are not created regularly.

AWS Console has a straight forward configuration tool where the S3 bucket and domain for your distribution are chosen. HTTPS can be enabled by either assigning an existing certificate to the distribution or creating a free certificate using Amazons Certificate Manager.

6 Frontend deployment

6.1 AWS Command Line Interface

AWS CLI is a command line application for managing AWS services. It is written in Python and is supported on Mac, Linux and Windows. The CLI can be used for creating more complex scripts that combine services, in this case CloudFront and S3. [14]

6.2 Deploying using the CLI

6.2.1 Deploying new versions

With an existing CloudFront distribution, all it takes to deploy a new frontend version is to push new files to the S3 bucket attached to the distribution:

```
aws s3 sync <path> s3://<bucket name>
```

The files will be available after the CloudFront has distributed the files to the distributions edge locations. This usually happens in less than a minute.

If the cache duration for files is set to a long time, returning users will not be able to see the changes. Since static files in this project are versioned with a hash in the filenames, caching in CloudFront was set to infinity. To work around this CloudFront caches can

be invalidated. With revision tagged files, only the non-revisioned HTML files that map to URLs need to be invalidated. With the CLI, the invalidation can be done with the following command:

```
aws cloudfront create-invalidation --distribution-id <id>
--paths "/*html"
```

Invalidations take between 10 and 15 minutes to complete, during which the old files are served with no visible interruption to users.

6.2.2 Rolling back releases

Rollbacks can be made very quick by storing previous frontend versions in their own buckets. CloudFront is tied to one bucket to which new versions are synced from the buckets they were deployed to:

```
aws s3 sync s3://<old release bucket> s3://<active bucket>
```

Like with new deployment, CloudFront cache needs to be invalidated for changes to become visible to users.

6.3 Automating frontend deployment

The deployment process can be split into two phases: deploying a new version and enabling a deployed version. These were implemented as shell scripts that use the AWS CLI commands. Additionally, a third script is used to clean up files left over from previous versions sometime after enabling a new version. The enabling script does not delete any files from the hosted bucket so that resources required in html files are still found if a user requests a page before invalidation has completed.

6.3.1 Deploying a new version

Since we want to keep older versions for rollbacks, we need to create a tag which we can use to create a subdirectory inside the deployment bucket. This tag will also be pushed to git when building production versions.

```
DATE=$(date -u +"%Y-%m-%d_%H-%M")
TAG="frontend_${DATE}"

aws s3 sync ${BUILD_DIR} ${BUCKET}/${TAG}
```

Snippet 1. Simplified version of the deployment script

Snippet 1 shows the deployment process with input validation and variable definition omitted. The bucket variable is set based on the environment we are deploying to.

6.3.2 Enabling a version

To enable a new version, we need to first sync files from the source bucket to the one served by CloudFront and then invalidate the files CloudFront is serving.

```
aws s3 sync ${SOURCE_BUCKET}/${TAG} ${DEST_BUCKET}
aws cloudfront create-invalidation –distribution-id ${ID} –paths "/*"
```

Snippet 2. Simplified version of the enabling script

Source and destination buckets are again evaluated based on the environment passed to the script as a variable. If no tag is given as argument, the latest version is queried from S3 by sorting the deployed versions by date.

6.3.3 Cleaning up

Since there is a small time-window where users could receive an outdated version of an HTML file, we cannot remove old versions of files right away. To get rid of the left-over files later, we can sync the buckets again, this time using the --delete flag. This removes any files from the destination bucket that did not exist in the source bucket.

7 Backend deployment

Backend deployment is a more complex process than frontend deployment. It is the riskier since interruptions can cause loss or corruption of data and thus must be done more carefully. This usually means having at least short downtime during deployments. There is also no way of validating that the new version works correctly in production before it is accessed by users. Recovery from failed deployments is also time consuming further adding to the effect of broken releases.

7.1 Common backend architecture

Modern web service backends usually consist of multiple control servers with one or more load balancers directing traffic to them (Figure 2).

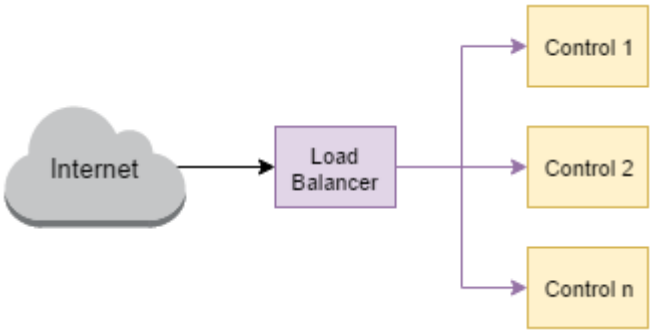

Figure 2. Backend architecture

There are two common ways that systems like this are traditionally updated: updating the servers one by one or all at the same time. The former results in an inconsistent state due to different versions of the software running at the same time. The latter will result in an interruption when the whole service is brought down for the duration of the update.

7.2 Blue-Green deployment

To work around the issues in backend deployment, we can take advantage of being in a cloud environment. Being able to create instances at will makes it possible to create a mirror image of the current production or testing environment running a new version of the server software. The newly created environment can be kept separate from the public serving one and taken to use after validating everything is working as expected. This way of releasing software is often called blue-green deployment [2, 261].

Figure 3. Two identical production environments, blue and green.

In a blue-green deployment, seen in Figure 3, the green environment is serving users, while the blue one is being prepared for replacing it.

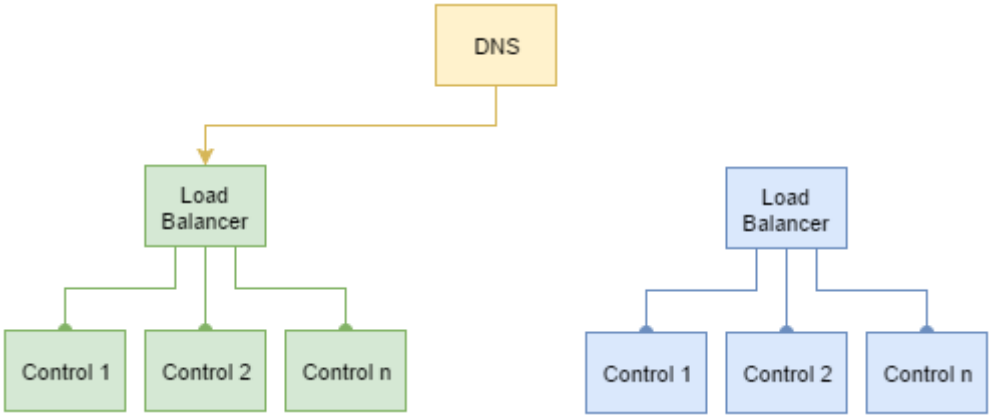

Figure 4. DNS pointing the previous deployment

One way to take the blue environment into use is to point the domain of the service to it (Figure 4 and 5).

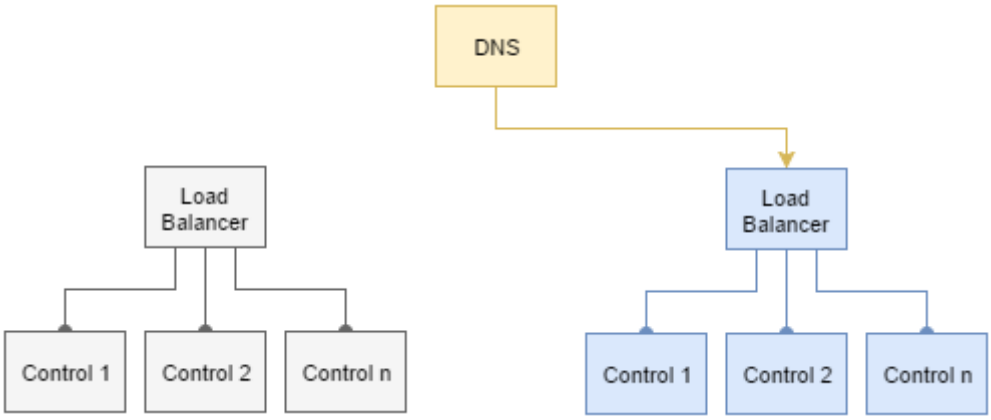

Figure 5. Traffic is directed to the new environment using DNS

This method has the advantage of being very simple to implement, which makes it low risk as well. Once the DNS switch is done, requests will be handled by the new environment with no interruption. The downside is that the instant when the new version will be accessible by users is limited by DNS propagation speed. Precise deployment timing to synch for example frontend and backend deployments is thus not possible.

Another way of making the switch is to replace the instances on the load balancer.

Figure 6. Instances on the load balancer are replaced with new ones

The switch seen in Figure 6 is more complex but results in the new instances becoming active immediately. This makes rollbacks faster and more predictable as well. When evaluating these two methods, there was no noticeable interruption to service in either of them. Depending on the load balancer in use, it can be worthwhile to use the latter.

As seen in Figures 5 and 6, the old instances are left standing by. This makes it possible to roll back the deployment very quickly by either reverting the DNS to the old load balancer or switching the instances on the load balancer.

In the current implementation, old instances are left running for two hours. In some cloud environments, like AWS, stopped instances do not cost anything and can be brought up quicker than completely new instances. This makes it cost effective to leave old environments standing by for a longer time.

7.3 Automating backend deployment

The deployment process consists of scripts for creating, and managing instances and the chatbot controlling the scripts.

7.3.1 First phase – Creating a new environment

Figure 7 shows the first phase of backend deployment. This phases is started by creating new instances matching the environment we are deploying to. In terms of performance, this is not a demanding task on the deployment machine; thus, we can build the server package in parallel while configuring the instances. With the new instances in place, we deploy the server package with an SSH utility.

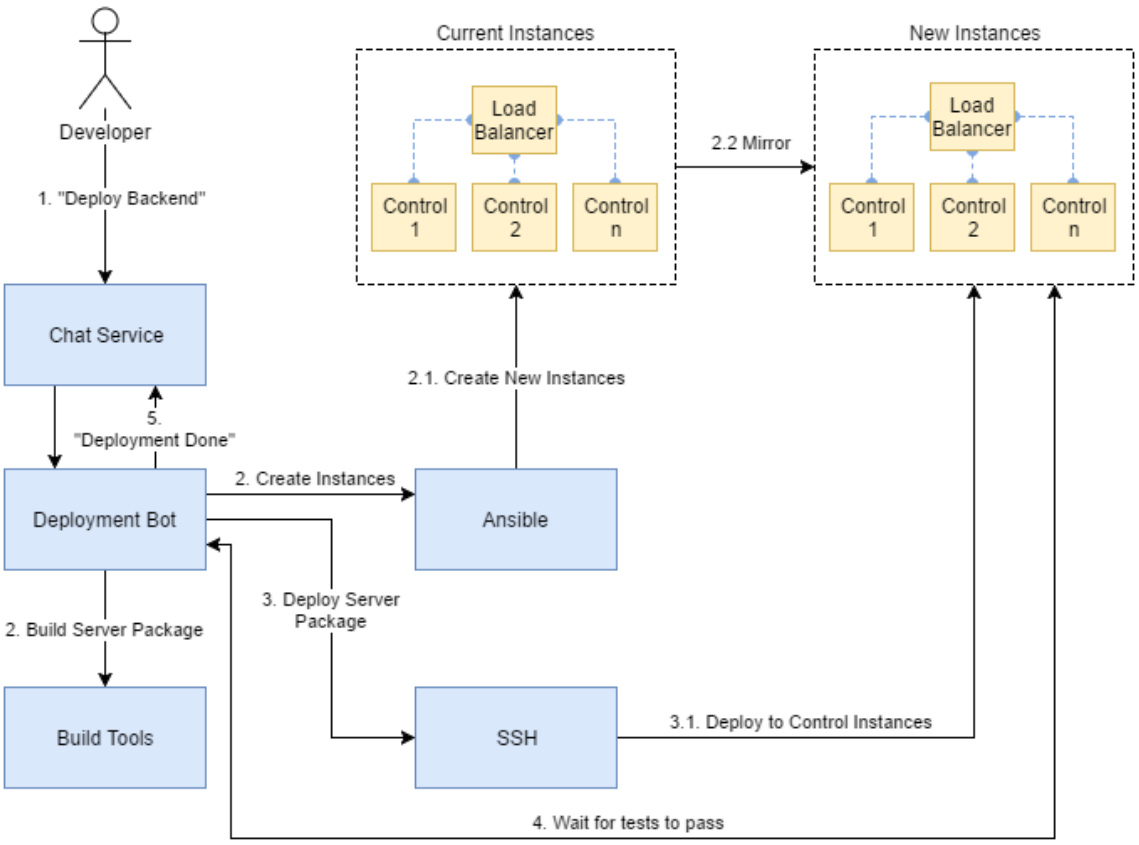

Figure 7. First phase of the backend deployment process

Once the new environment is up, we can validate that the service is working correctly by running automated tests against it. After these checks pass we can be quite confident that it is safe to proceed with the deployment. At this stage, we can also test any other feature by hand if needed.

7.3.2 Second phase – switching to new environment

In the second phase of the deployment, the new version is taken in to use by switching instances on the load balancer. The old environment is scheduled to terminate two hours after this phase is finished.

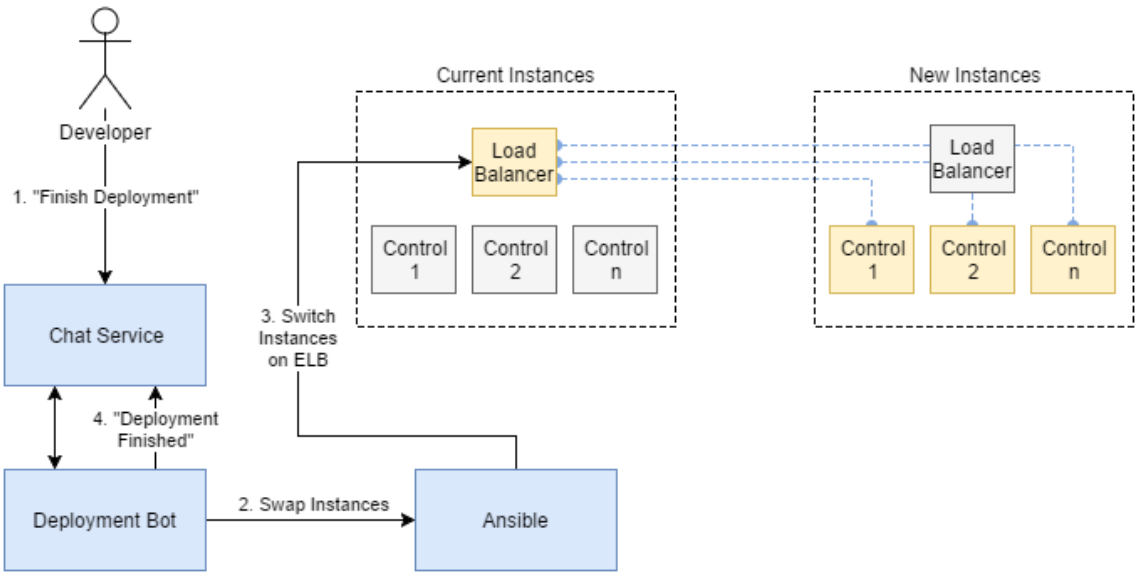

Figure 8. Second phase of the backend deployment process

The most complex task of the new deployment process is the creation, configuration and termination of instances. Several tools exist for this purpose. Ansible was chosen for this project for extensive feature coverage as well as its declarative syntax.

7.4 Ansible

Ansible is an open-source automation tool for managing and configuring computers. It is developed by Red Hat and the open-source community. It is designed to manage complex infrastructure rather than single instances. Dynamic inventory of instances, which is necessary when instances come and go, is supported with extra packages for common cloud providers.

Ansible uses YAML as its language. YAML is a human-readable data serialization language commonly used in configuration files. Ansible tasks are in fact not so much actions as they are descriptions of the desired result.

```
site.yml                # master playbook
roles/
    common/             # this hierarchy represents a "role"
        tasks/          #
            main.yml    #  <-- tasks file can include smaller files if warranted
        handlers/       #
            main.yml    #  <-- handlers file
        templates/      #  <-- files for use with the template resource
            ntp.conf.j2 #  <------- templates end in .j2
        files/          #
            bar.txt     #  <-- files for use with the copy resource
            foo.sh      #  <-- script files for use with the script resource
        vars/           #
            main.yml    #  <-- variables associated with this role
        defaults/       #
            main.yml    #  <-- default lower priority variables for this role
        meta/           #
            main.yml    #  <-- role dependencies
```

Figure 9. Common Ansible project structure [11]

The main component of an Ansible task is called a playbook. Playbooks can contain simple tasks or multiple tasks combined into roles. Roles can contain multiple tasks and have their own variable definitions. They can also have files and templates that can be copied to remote servers and handlers for common actions. Figure 11 shows the structure of a common Ansible project.

7.4.1 Creating and configuring instances

Instances are created with the Ansible ec2 module. To keep track of existing instances, tags can be used to distinguish instances in different environments or states. Tag counts can be used to ensure a specific number of instance in an environment regardless of how many times the script is executed.

To mirror an existing environment for deployment, the count and type of each instance role can be calculated with their tags. This calculation can be done in the variable definition of a playbook.

The internal inventory of Ansible can be accessed through the maplike data structures *groups* and *hostvars*. The *groups* variable contains general information about instances and can be used to query for a list of instances with a specific tag. The *hostvars* variable has more concrete information about specific instances including the type of the instance and its public IP-address.

```
- vars:
    control_instances: "{{ groups['tag_role_control'] }}"
    control_instance_count: "{{ control_instances | length }}"
    control_instance_type:
      "{{ hostvars[control_instances[0]]['ec2_instance_type'] }}"

  roles:
    - {
      role: create_instances,
      ami_image: control,
      ec2_instance_type: "{{ control_instance_type }}",
      number_of_instances: "{{ control_instance_count }}",
      instance_role: control,
      instance_state: deployment
    }
```

Snippet 3. Ansible playbook for creating new control instances

Snippet 3 shows how a role to create instances is passed variables containing the count and type of the instance it should create.

```
- name: Create instances
  ec2
    instance_type: "{{ ec2_instance_type }}"
    region: "{{ aws_region }}"
    instance_tags: {
      "role": "{{ instance_role }}",
      "state": "{{ instance_state }}"
    }
    exact_count: "{{ number_of_instances }}"
    count_tag:
      role: "{{ instance_role }}"
      state: "{{ instance_state }}"
```

Snippet 4. Ansible task that creates an instance using variables passed to the role

In Snippet 4 the *ec2* module is used to create an instance with the variables passed from the playbook. Tags are used to specify which existing instances Ansible should look for to decide if more instances need to be created to reach the specified count.

Instances on the load balancer are swapped by first adding the new instances and then immediately removing the old ones.

```
- name: Add new instances to ELB
  local_action:
    module: ec2_elb
    state: present
    region: "{{ aws_region }}"
    ec2_elbs:
      - "{{ elb_name }}"
    instance_id: "{{ hostvars[item]['ec2_id'] }}"
    wait: no
  with_items: "{{ deployment_controls }}"
```

Snippet 5. Ansible task that adds deployment instances to the active load balancer

Snippet 5 shows how instances can be added to a load balancer. To remove the old ones the state parameter will be changed to absent.

8 ChatOps with Hubot

Hubot is a chatbot developed by GitHub, Inc. It is a Node.js application written in CoffeeScript, a language which compiles into JavaScript and can be used with either language. Hubot supports third party plugins that can be installed as JavaScript modules with the node package manager npm. [6]

Hubot works by listening to chat channels and performing actions when a message matches a listener. Middleware can be used to intercept messages before they reach the listeners. This can be used, for example, to check if the sender of a message is authorized to command the bot.

8.1 Integrating with Flowdock

Support for various chat services is handled with adapters. Integrating with Flowdock is done by installing the adapter with NPM and configuring its options. Adapter options are set using environment variables. For Flowdock, the only mandatory variable is the API token for the bot's Flowdock account [7].

This approach makes it possible to develop bot functionality without committing to a single service provider. If we were to switch to a different chat service, the only change needed to the bot would be installing and configuring a new adapter.

8.2 Authentication

The bot will be on a chat channel used only by the development team, but since it can make major changes to production environments, a form of authentication is necessary. Hubot has a role based authentication plugin that uses user IDs as its authentication method. Roles are used to authorize certain actions. For this project, only a single role for deployments is necessary.

Figure 10. Granting a role to a user

With the authentication plugin, roles can be granted to users through the bot by users with an admin role. Figure 10 shows an admin user giving another user a role which enables them to initiate deployments.

Like most chat services, Flowdock can be accessed from outside of office network. This raises a security concern in the case that a device of an authorized person with Flowdock logged in ended up in the wrong hands. To solve this issue all actions that were considered dangerous were made to require confirmation from another authorized person before the bot would execute the action.

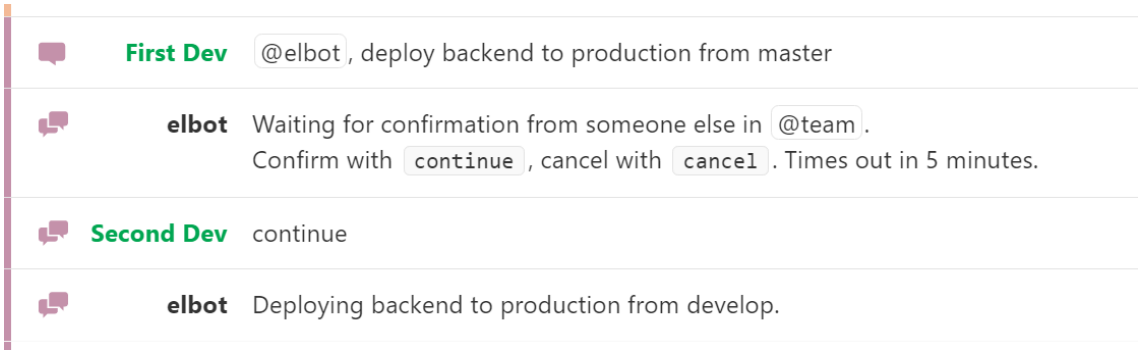

Figure 11. Deployment bot requests confirmation for a production deployment.

As seen in Figure 11, the confirmation is implemented as a conversation. The bot will alert the development team and wait for five minutes for someone to confirm the initiated action. If no confirmation is received the action is cancelled.

8.3 Persisting information

Hubot has an in-memory map for keeping track of information such as user roles. This data storage is called the bot's brain. By default, any data stored in the brain is lost when the bot is shut down. For data to persist through restarts, the bot can be connected to a database.

When data is added to the brain, the bot emits a save event with the data stored in the brain. Database persistence is handled by listening to this event and then saving the data to the database. Most common databases have third party plugins for handling this.

8.3.1 Redis

Redis is a lightweight in-memory key-value database. It is often used in high-throughput scenarios, where immediate data persistence is not critical, such as caching [9, 10]. However, for this use case, it was selected for ease of setup and its negligible performance impact with small amounts of data [8].

Redis supports different levels of on-disk persistence. The request rate here is so low that persisting on every update is feasible.

8.4 Updating the bot

The bot is implemented in JavaScript, which is an interpreted language. This makes updates quite simple. To get new features into use, we just need to pull changes from version control and restart the bot.

8.4.1 Automatic updates

Pulling repositories from version control was already implemented for Ansible and server repositories, and the same functionality can be reused here. Node.js does not provide a way to restart itself from within, but we can use the operating system's init system, for example systemd, for this purpose.

```
[Unit]
Description=Hubot

[Service]
ExecStart={{repository_directory}}/bin/hubot-prod
Restart=always
RestartSec=10
User={{ hubot_user }}
Group={{ hubot_user }}

[Install]
WantedBy=multi-user.target
```

Snippet 6. The systemd service script for the bot. Note the Ansible variable syntax with double curly brackets.

The systemd init script seen in Snippet 6 is set to restart the service whenever it goes down. This ensures the bot is restarted if it were to crash as well as bringing it back up if it is terminated on purpose. With this script in place, the update process can easily be completed by exiting the Node.js process after pulling changes.

8.4.2 Accessing GitHub

GitHub offers three ways to authenticate automated deployment scripts: HTTPS with OAuth tokens, deploy keys and machine users. Deploy keys were chosen for this project because they do not require an additional account and require less configuration than using OAuth tokens.

Deployment keys are SSH keys and having multiple keys for one domain causes an issue on how to use the correct key for a given repository when authenticating. We can solve this by using fake subdomains in git and SSH configuration files.

```
Host ansible.<GitHub hostname>
  HostName <GitHub hostname>
  User git
  IdentityFile /home/hubot/.ssh/ansible
  IdentitiesOnly yes

Host server.<GitHub hostname>
  HostName <GitHub hostname>
  User git
  IdentityFile /home/hubot/.ssh/server
  IdentitiesOnly yes
```

Snippet 7. SSH configuration file using fake subdomains to choose correct SSH identity files

The subdomains “ansible” and “server” seen in Snippet 7 are also set as the repositories’ origins in their configuration files. The “HostName” field is the actual host the connection will use. With this setup, we can use the git command line tool in the same way as we would when using a GitHub account.

8.5 Performing deployments

With everything in place, we can start deploying completely through ChatOps. The following shows a complete backend deployment to a production environment.

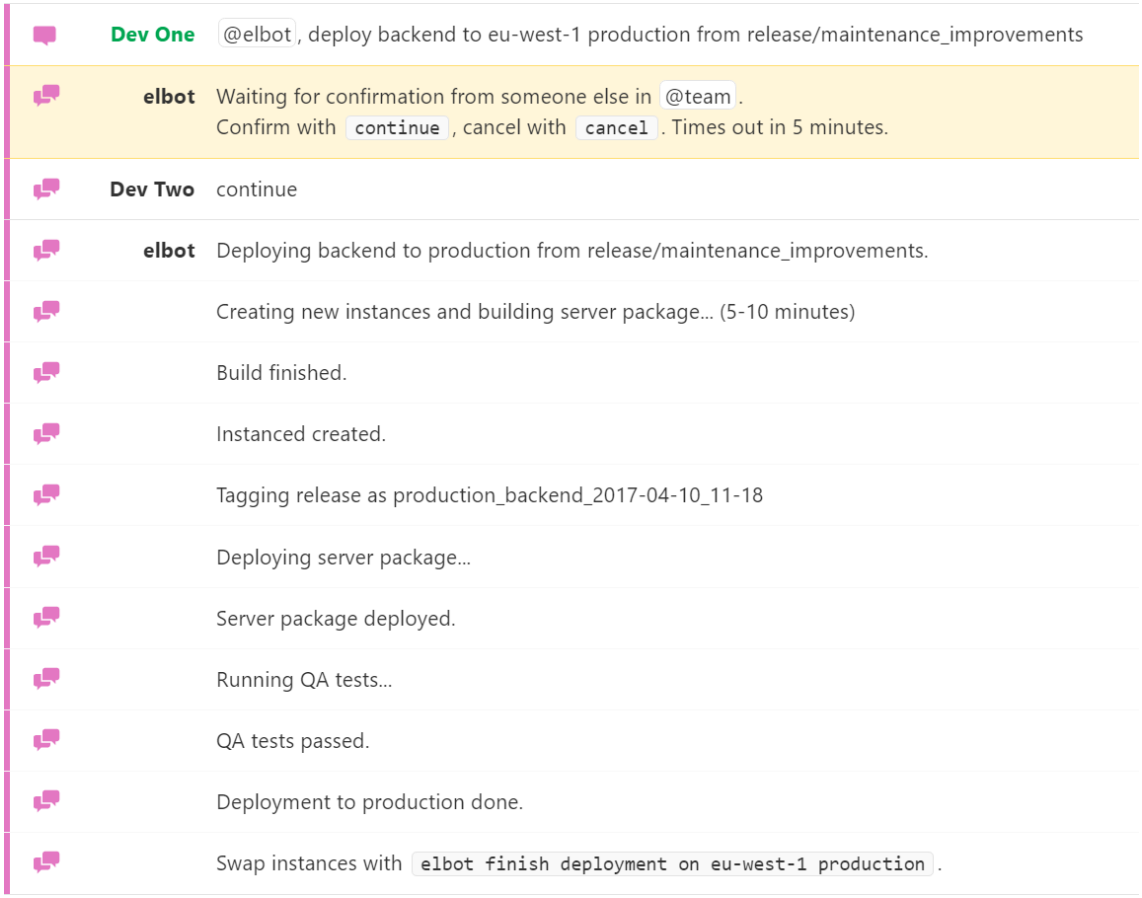

Figure 12. First phase of a production deployment

Figure 12 shows a successful production deployment that has completed the first phase and is waiting for a developer to command the bot to finish the deployment. Since this is a production deployment, permission was required and received from another authorized developer to start the deployment.

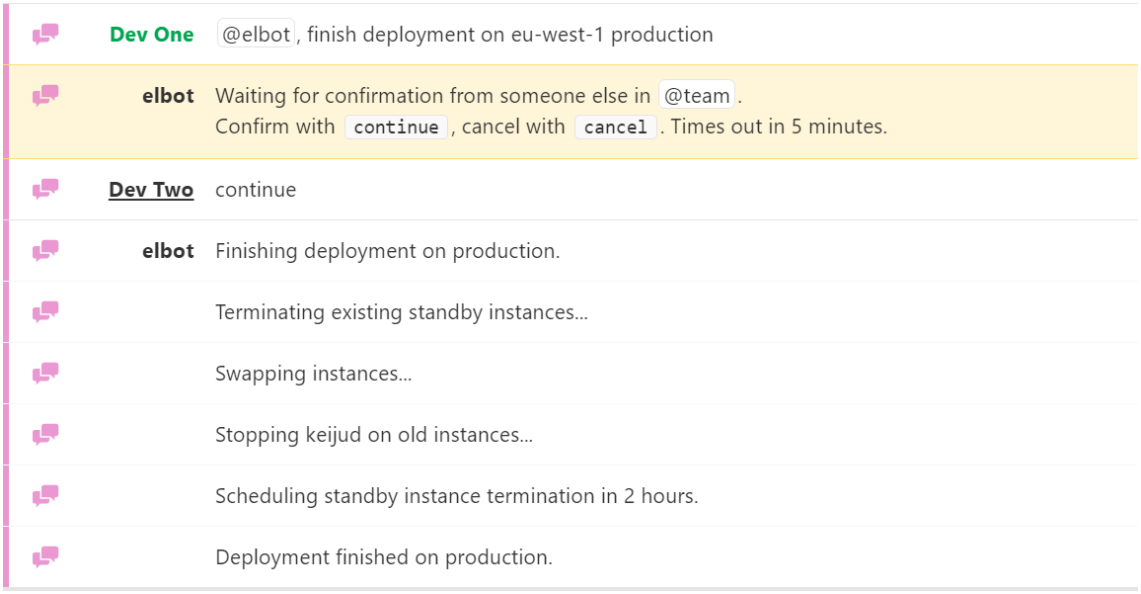

Figure 13. Second phase of a production deployment

In Figure 13, the deployment is finished by taking the new environment into use. Permission is required for this step as well. Termination of the previous environment is scheduled after the deployment is finished. During the two hours before the old instances are terminated we can roll back to the previous environment by reversing the finishing process.

9 Conclusion

ChatOps proved to be an excellent solution for our use case. Especially the blue-green deployments for the backend servers benefitted from the automation and control achieved by using a chat bot. We have also been able to increase the frequency of production deployments because of the reduced service breaks and less development time spent on performing the deployments.

The initial time investment for implementing this deployment system was high compared to ones relying on more manual labour. However, most of the time-consuming challenges faced during development were related to ChatOps being a completely new way of working for us. Further development of the bot, for example adding additional QA testing, has been simple once we had the base structure in place. This makes the original assumption of ChatOps encouraging further automation hold true for us.

The benefit of collaboration through using ChatOps has been especially apparent when team members are working from different locations. The chat channel where the bot is used becomes a natural place for coordinating deployments and discussing possible issues seen in new deployments.

Due to the way chat bots are implemented, they naturally understand only a specific set of commands. This proves to be a powerful security feature against accidental command execution. For example, if the bot is only capable of terminating instances in a specific state, it becomes impossible to accidentally terminate instances that are active in production.

In this implementation, ChatOps has been demonstrated to be a worthwhile option even for small teams without a dedicated DevOps group. In fact, it has, in my experience, the most potential when developed by the whole team.

References

1 Elisa Live. Elisa [online]
URL: https://elisalive.com
Accessed November 7, 2016.

2 Humble J, Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston MA: Pearson Education, Inc; 2011

3 What Is Amazon CloudFront? AWS [online].
URL: http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html
Accessed September 29, 2016.

4 What Is Amazon S3? AWS [online].
URL: http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html
Accessed September 29, 2016.

5 Amazon S3 Bucket Naming Requirements. AWS [online].
URL: http://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html
Accessed September 29, 2016.

6 Hubot. Github [online].
URL: https://hubot.github.com/
Accessed November 16, 2016.

7 Hubot adapter for Flowdock. Flowdock [online].
URL: https://github.com/flowdock/hubot-flowdock
Accessed November 16, 2016.

8 Redis FAQ. Redis [online].
URL: http://redis.io/topics/faq
Accessed November 17, 2016.

9 11 Common Web Use Cases Solved In Redis. High Scalability [online].
URL: http://highscalability.com/blog/2011/7/6/11-common-web-use-cases-solved-in-redis.html
Accessed November 17, 2016.

10 Top 5 Redis use cases. ObjectRocket [online].
URL: http://objectrocket.com/blog/how-to/top-5-redis-use-cases
Accessed November 17, 2016.

11 Best Practices. Ansible [online].
URL: http://docs.ansible.com/ansible/playbooks_best_practices.html
Accessed November 20, 2016.

12 What is ChatOps? A guide to its evolution, adoption, and significance. Regan S [online].
URL: http://blogs.atlassian.com/2016/01/what-is-chatops-adoption-guide/
Accessed November 20, 2016.

13 ChatOps at GitHub. Newland J [video file].
URL: https://youtu.be/NST3u-GjjFw
Accessed November 20, 2016.

14 AWS Command Line Interface. AWS [online].
URL: https://aws.amazon.com/cli/
Accessed November 20, 2016.

15 DNS Support for Load Balancing. Brisco T [online].
URL: https://tools.ietf.org/html/rfc1794
Accessed November 20, 2016.

16 Managing Deploy Keys. GitHub Inc. [online].
URL: https://developer.github.com/guides/managing-deploy-keys/
Accessed November 22, 2016.