DevOps Principles And Version Control

1.0 Contents [Heading3,14, Arial]

2.0 General Information: [Heading3,14, Arial]

2.1 Document Jira/ Github Ticket(s) [Heading4,12, Arial]

Ticket(s) Name	Url
DevOps Principles And Version Control [Normal text,10, Arial]	Jira / github url

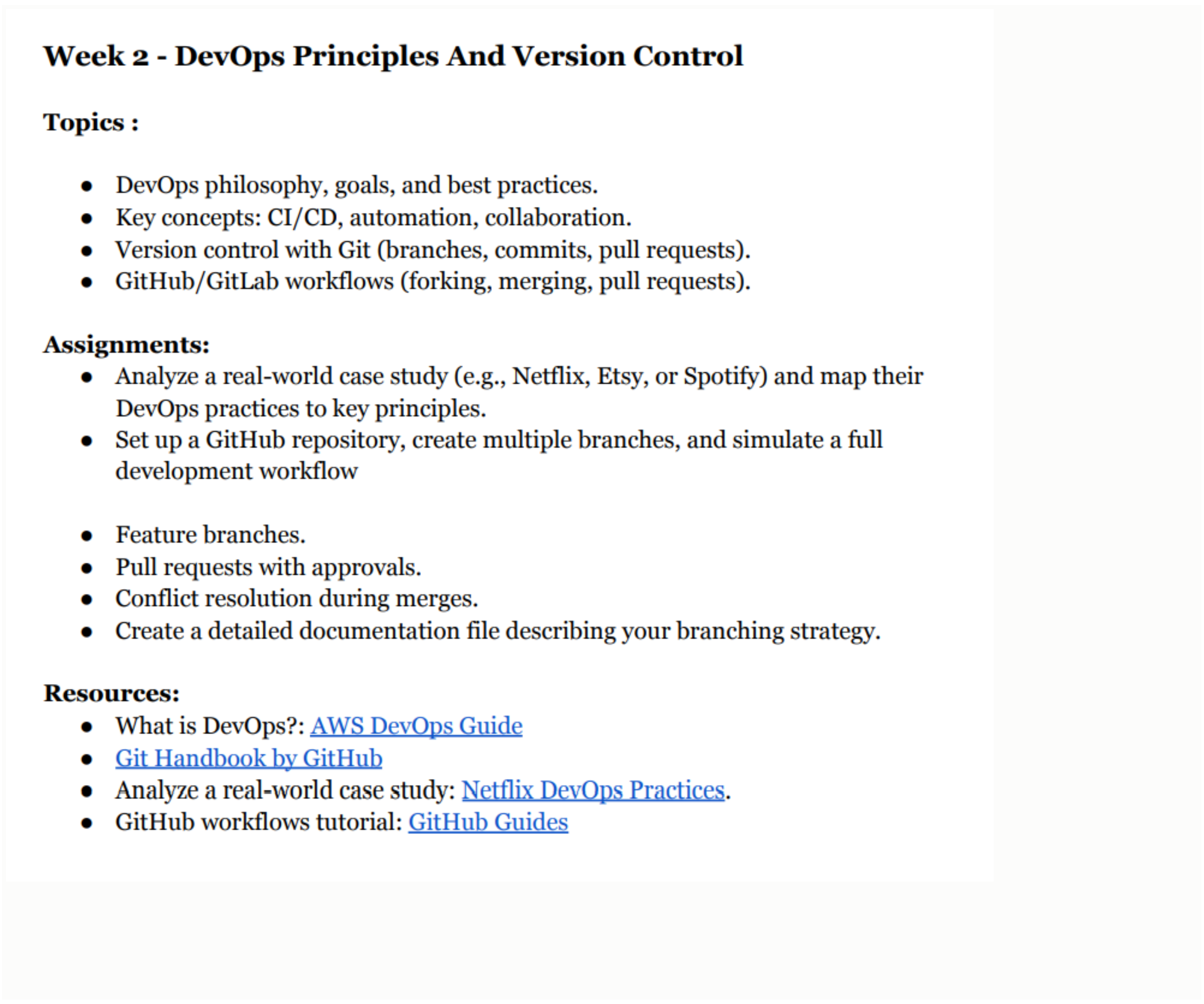
2.2 Document Purpose

This document aims to analyze the DevOps practices of Netflix and map these practices to key DevOps principles. Additionally, the document will demonstrate setting up a GitHub repository, creating branches, and simulating a full development workflow, including pull requests and code reviews, to illustrate DevOps in action. [Normal text,10, Arial, Justify Alignment]

2.4 Document References

The following artifacts are referenced within this document. Please refer to the original documents for additional information.

Date	Document	Filename / Url
12-05-2025	Netflix system design blog	https://www.linkedin.com/pulse/system-design-netflix-narendra-l/?published=t
12-05-2025	Netflix DevOps Practice blog	https://www.bunnyshell.com/blog/how-netflix-does-devops/ https://www.simform.com/blog/netflix-devops-case-study/
13-05-2025	Merge conflict	https://www.youtube.com/watch?v=FyAAIHHClqI
13-05-2025	Branching Strategies	https://www.youtube.com/watch?v=JIVMLZp-SI0&t=2422s



Week 2 - DevOps Principles And Version Control

Topics :

- DevOps philosophy, goals, and best practices.
- Key concepts: CI/CD, automation, collaboration.
- Version control with Git (branches, commits, pull requests).
- GitHub/GitLab workflows (forking, merging, pull requests).

Assignments:

- Analyze a real-world case study (e.g., Netflix, Etsy, or Spotify) and map their DevOps practices to key principles.
- Set up a GitHub repository, create multiple branches, and simulate a full development workflow

- Feature branches.
- Pull requests with approvals.
- Conflict resolution during merges.
- Create a detailed documentation file describing your branching strategy.

Resources:

- What is DevOps?: AWS DevOps Guide
- Git Handbook by GitHub
- Analyze a real-world case study: Netflix DevOps Practices.
- GitHub workflows tutorial: GitHub Guides

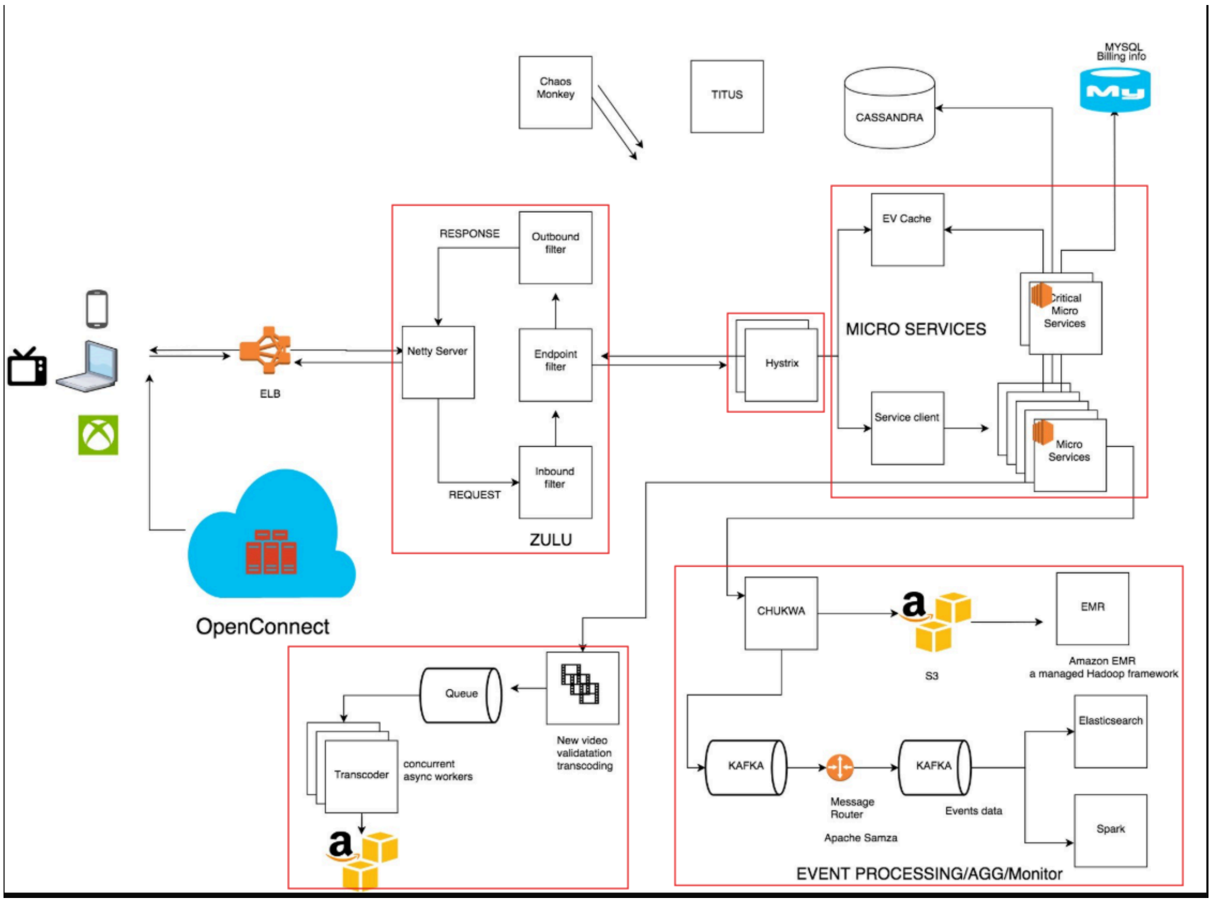
3.0 Document Overview:

This document, prepared by Expert Cloud Consulting, provides a comprehensive guide to DevOps principles through a case study of Netflix’s microservices architecture and practical implementation details of Git version control. It blends theoretical principles with hands-on instructions for setting up infrastructure and managing code using GitHub.

The key highlights include:

- Analysis of Netflix's DevOps strategies.
- Git branching strategies.
- Conflict resolution techniques for collaborative development.

4. Netflix Microservices Architecture



4.1. Core System Architecture

4.1.1. Client Layer

4.1.1.1. Devices: Smart TVs, mobile apps, web browsers, and gaming consoles.

4.1.1.2. Functions: UI rendering, playback initiation, and adaptive video streaming control.

4.1.2. Backend Services (AWS-Based)

4.1.2.1. Microservices handle authentication, recommendations, billing, etc.

4.1.2.2. AWS Services: EC2, S3, ELB, RDS, DynamoDB, Lambda, Auto Scaling, CloudWatch.

4.1.3. Open Connect (OC)

4.1.3.1. Proprietary Netflix CDN.

4.1.3.2. Delivers video content from geographically distributed Open Connect Appliances (OCAs).

4.2. Adaptive Streaming and Media Transcoding

4.2.1. Why It's Required:

4.2.1.1. Supports 2200+ devices.

4.2.1.2. Matches video quality to bandwidth and device type.

4.2.2. Workflow

4.2.2.1. **File Ingestion**: Upload HD master file.

4.2.2.2. **Validation:** Quality checks.

4.2.2.3. **Segmentation**: Break into chunks.

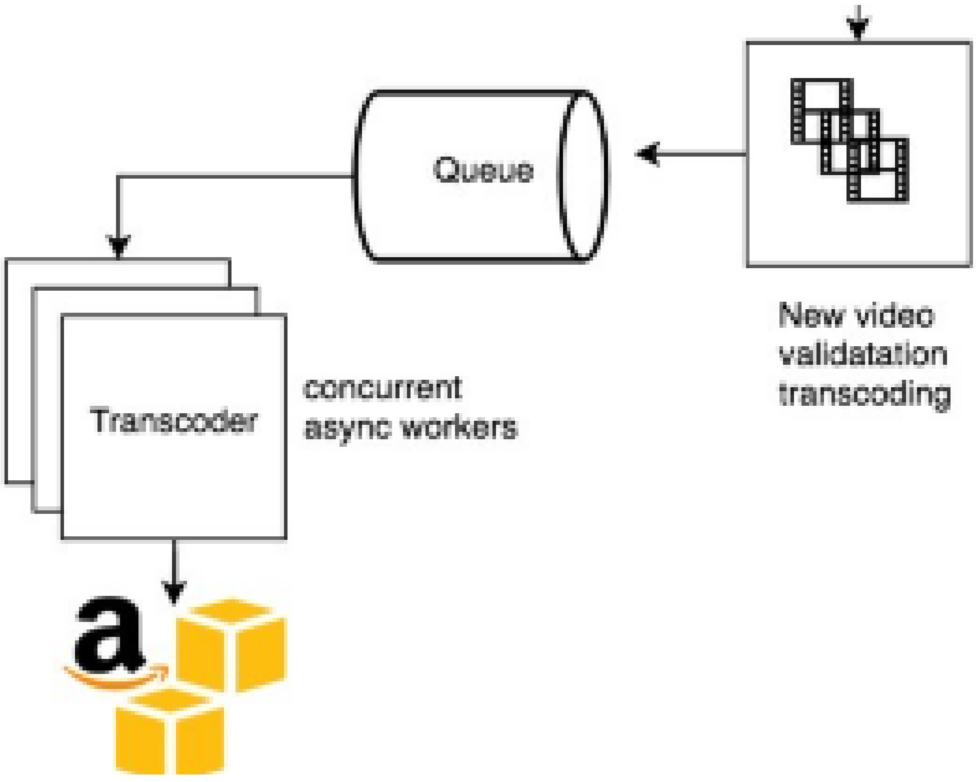
4.2.2.4. **Transcoding**: Create 1200+ adaptive versions (resolution + bitrate).

4.2.2.5. **Packaging**: Encrypted in HLS/MPEG-DASH format.

4.2.2.6. **Distribution**: Pushed to OCAs.

4.2.3. AWS Services:

4.2.3.1. S3, Elastic Transcoder, EC2, Titus, CloudFront (optional).



4.3. Load Balancing Strategy

4.3.1. **ELB Two-Tier Model**:

4.3.1.1. **DNS (Tier 1)**: Route to ELB endpoints via Route 53 based on AZ's.

4.3.1.2. **Zone-Level (Tier 2)**: Internal ELBs distribute requests among EC2s.

4.3.2. AWS Services:

4.3.2.1. ELB, ALB, Route 53

4.4. API Gateway: **ZUUL / ZULU**

4.4.1. Why It's Required

4.4.1.1. Controls routing, A/B testing, traffic segmentation.

4.4.2. Components:

4.4.2.1. **Inbound Filters**: Auth, validation.

4.4.2.2. **Endpoint Filters**: Route to service.

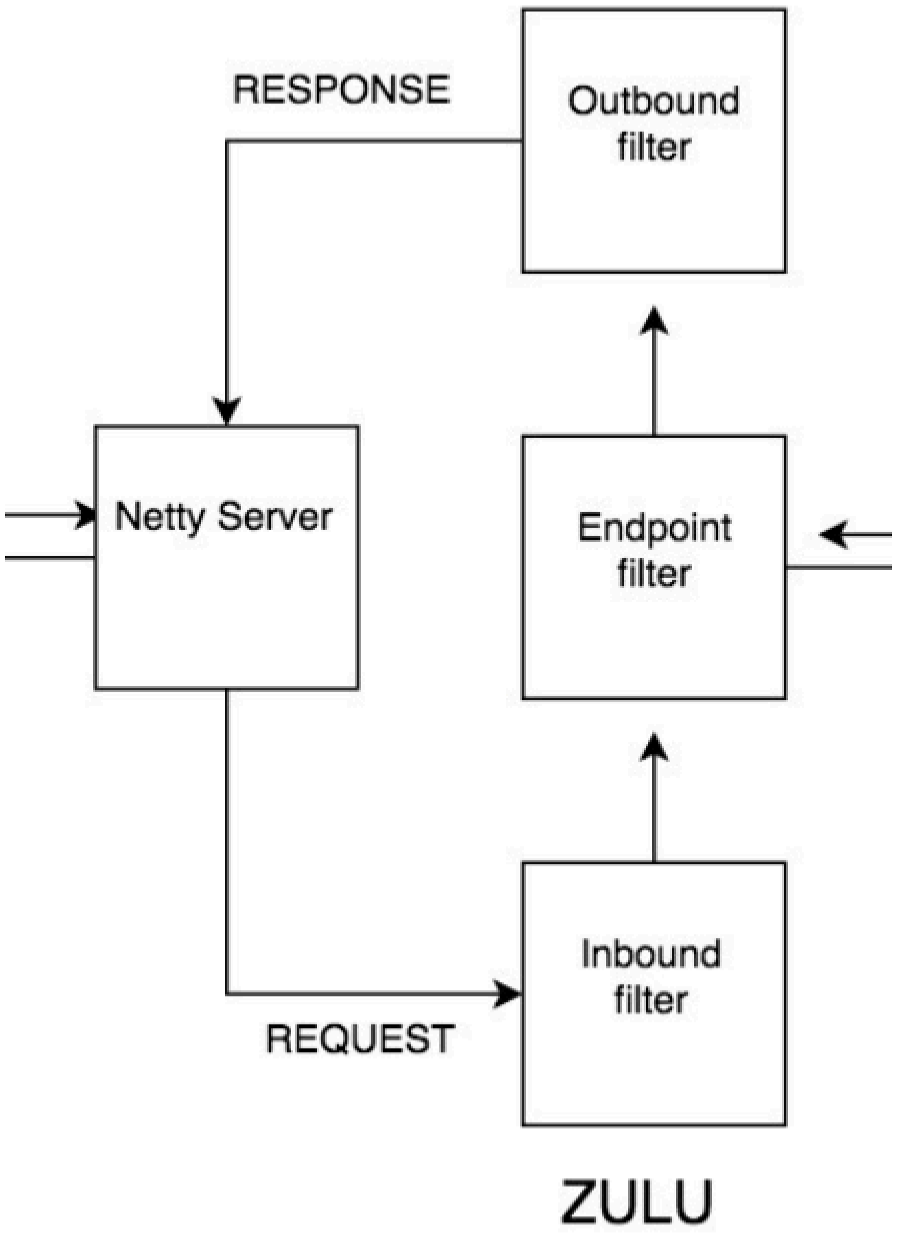
4.4.2.3. **Outbound Filters**: Gzip, headers, logging.

4.4.3. Features:

4.4.3.1. HTTP/2, mTLS, sticky sessions, canary support.

4.4.4. AWS Services:

4.4.4.1. EC2, CloudWatch, Route 53, ELB



4.5. Fault Tolerance Mechanisms: **Hystrix**

4.5.1. Why It's Required:

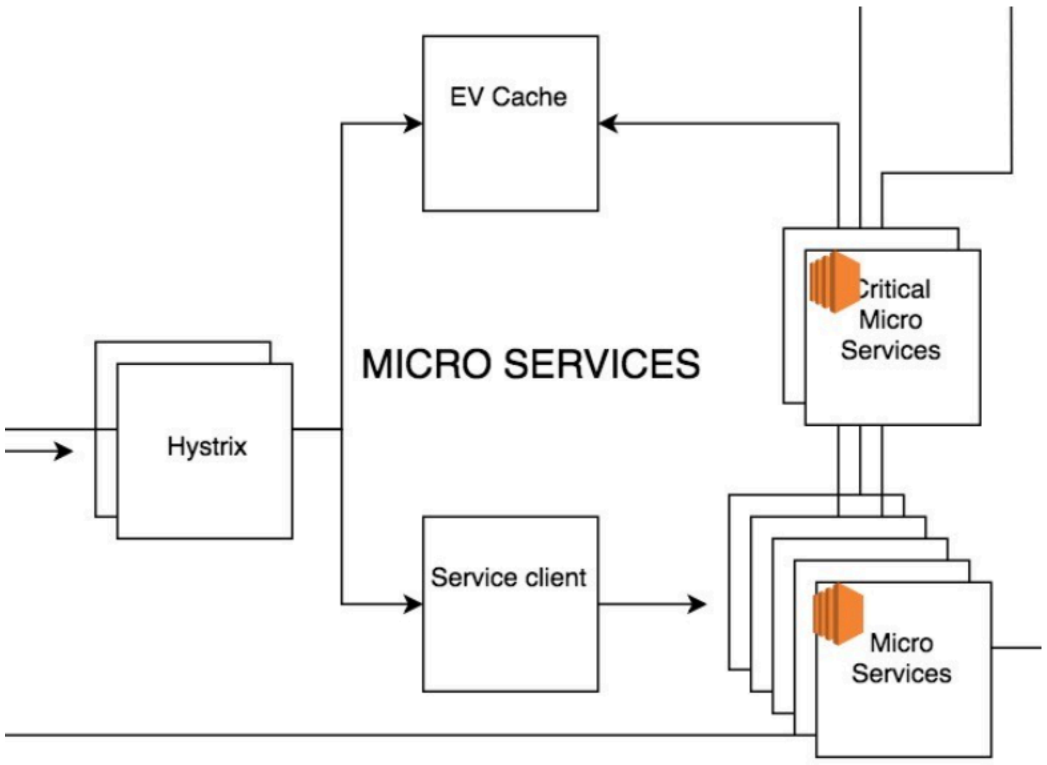
4.5.1.1. Prevent cascading failures in a distributed system.

4.5.2. Components:

4.5.2.1. Circuit Breaker, Fallback, Request Collapsing, Monitoring

4.5.3. AWS Services:

4.5.3.1. EC2, CloudWatch, optional SNS/SQS



NOTE- Netflix Hystrix was officially announced as deprecated in 2018 and is no longer maintained. Developers are encouraged to migrate to modern alternatives like **Resilience4j** or use service meshes such as **Istio** for resilience and fault tolerance in microservices.

4.5.4. Resilience4j

4.5.4.1. A lightweight, standalone library inspired by Hystrix but designed for **Java 8 and functional programming**.

4.5.4.2. Provides modules like CircuitBreaker, Retry, RateLimiter, Bulkhead, and TimeLimiter.

4.5.4.3. Easy integration with **Spring Boot** via annotations or programmatically.

4.5.4.4. Better performance and lower overhead than Hystrix.

4.5.5. Istio (Service Mesh)

4.5.5.1. Works at the network level (sidecar proxies) rather than within the application code.

4.5.5.2. Handles Circuit breaking, Retries, Timeouts, Traffic shaping, Observability.

4.5.5.3. Language-agnostic and ideal for polyglot microservices environments.

4.5.6. Migration Consideration:

4.5.6.1. If you're already on **Spring Boot**, switching to **Resilience4j** is straightforward.

4.5.6.2. If you want **centralized control** over service communication without code changes, consider a **service mesh** like Istio.

4.6. Microservices Strategy

4.6.1. Design Principles:

4.6.1.1. Independent, **stateless** services.

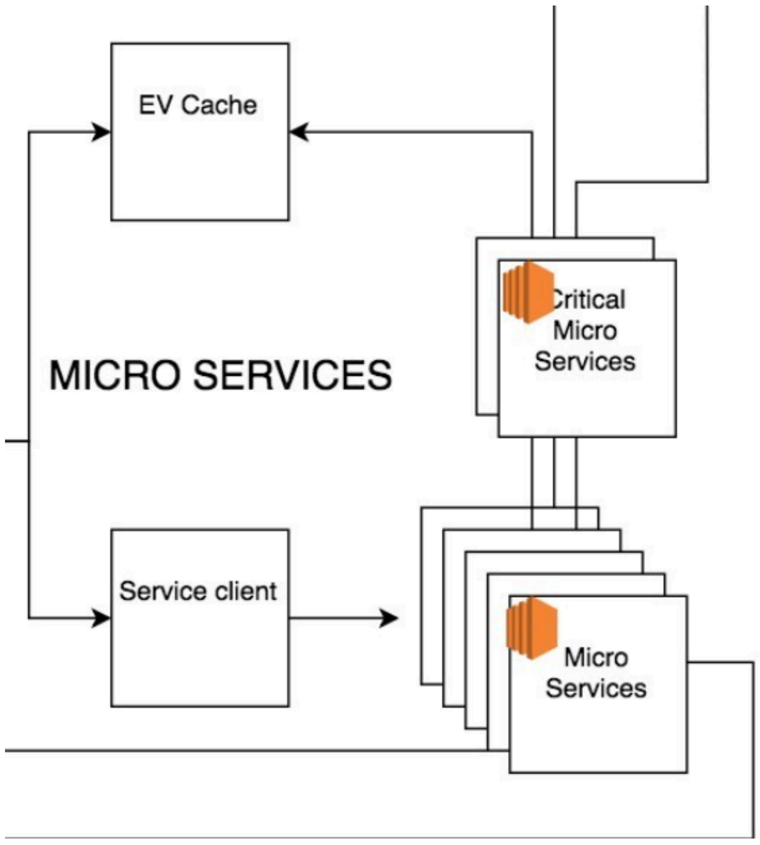
4.6.1.2. Critical services are highly resilient.

4.6.1.3. Scales containers based on traffic.

4.6.1.4.

4.6.2. AWS Services:

4.6.2.1. EC2, Elastic Beanstalk, API Gateway, CloudWatch



4.7. Caching Strategy

4.7.1. EVCache (Elastic Volatile Cache):

4.7.1.1. Based on Memcached.

4.7.1.2. Sharded, replicated, fault-tolerant.

4.7.2. SSD Tiering (storage optimization technique that automatically moves data between different types of storage media):

4.7.2.1. Hot data in RAM, cold data on SSD. (Based on the frequency of data accessed, it is categorized as hot data, which is frequently accessed, and cold data, which is rarely accessed.)

4.7.3. AWS Services:

4.7.3.1. EC2, EBS (SSD)

4.8. Data Management

4.8.1. MySQL on EC2:

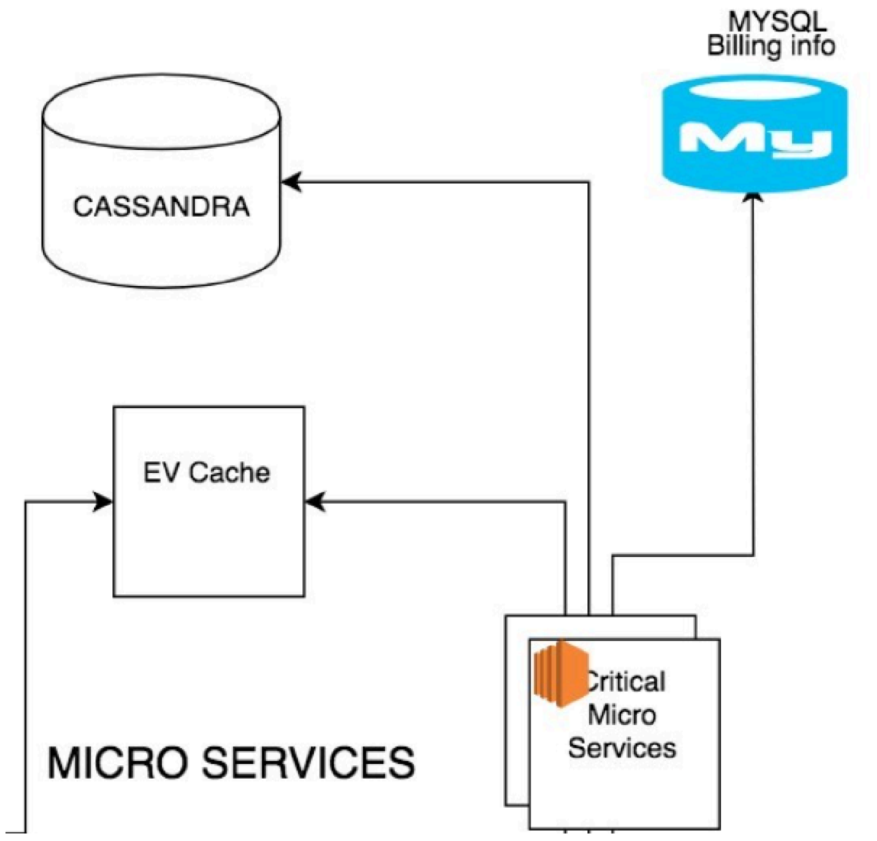
4.8.1.1. Billing, profiles with master-master setup, and replicas.

4.8.2. Cassandra (distributed NoSQL database):

4.8.2.1. Viewing history in LiveVH and CompressedVH formats.

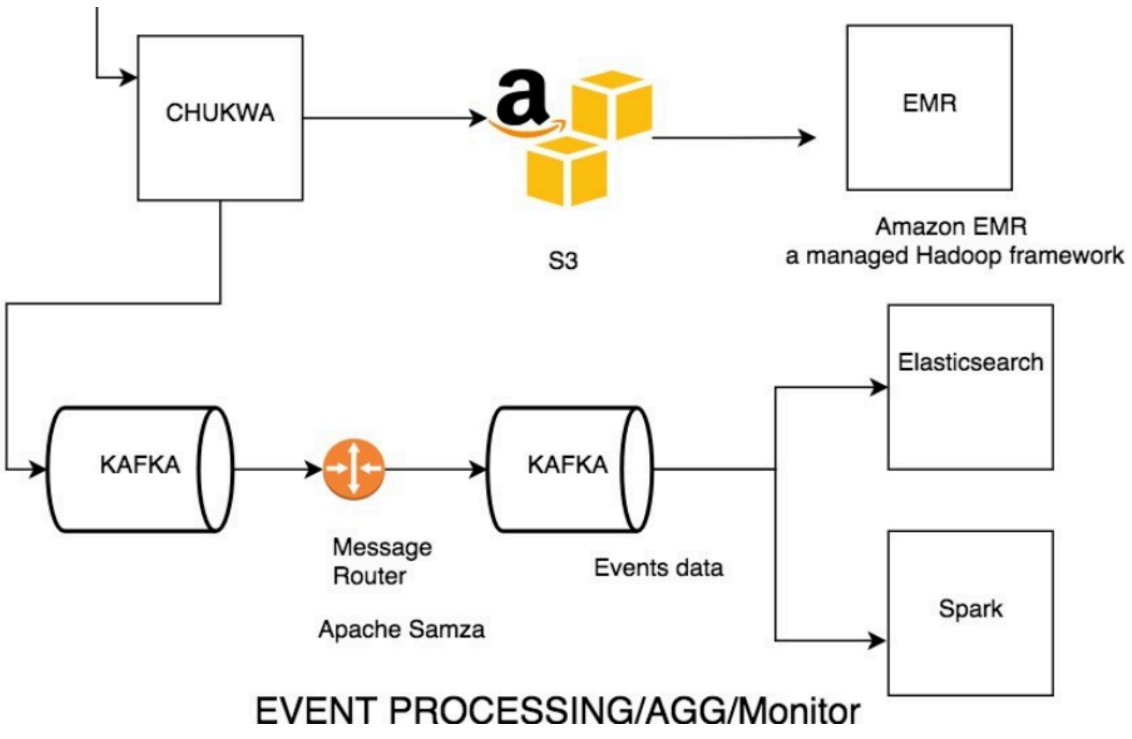
4.8.3. AWS Services:

4.8.3.1. EC2, Route 53, CloudWatch



4.9. Observability and Monitoring

- 4.9.1. Components:
  - 4.9.1.1. **Chukwa**: Event ingestion.
  - 4.9.1.2. **Kafka + Samza**: Stream processing.
  - 4.9.1.3. **Elasticsearch**: Query and diagnostics.
- 4.9.2. AWS Services:
  - 4.9.2.1. CloudWatch, CloudTrail, S3, IAM



4.10. DevOps Practices in Netflix

- 4.10.1. Netflix don't prevent engineers from accessing the production environment.
- 4.10.2. Netflix don't enforce using specific programming languages and frameworks
- 4.10.3. They don't focus on processes and procedures. That's because it's difficult for such a large organization to move as quickly if engineers are tied down by specific policies they need to follow
- 4.10.4. They don't enforce using specific programming languages and frameworks.
- 4.10.5. If their engineers can come up with new features and ideas, they have the freedom to implement them even if they affect uptime.
- 4.10.6. Netflix uses **Chaos Monkey**, it randomly terminates virtual machine instances and containers that run inside of your production environment. Exposing engineers to failures more frequently incentivizes them to build

resilient services. It is an example of a tool that follows the Principles of Chaos Engineering.

4.10.7. After the success with Chaos Monkey, Netflix engineers wanted to test their resilience to all sorts of inevitable failures, detect abnormal conditions. So, they built the **Simian Army**, a virtual army of tools

4.10.7.1. **Latency Monkey**
It creates false **delays in the RESTful client-server** communication layers, simulating service degradation and checking if the upstream services respond correctly. Moreover, creating very large delays can simulate an entire service downtime without physically bringing it down and testing the ability to survive. The tool was particularly useful to test new services by simulating the failure of dependencies without affecting the rest of the system.

4.10.7.2. **Conformity Monkey**
It **looks for instances that do not adhere to the best practices and shuts them down**, giving the service owner a chance to re-launch them properly.

4.10.7.3. **Doctor Monkey**
It detects unhealthy instances by tapping into health checks running on each instance and also monitors other external health signs (such as CPU load). **The unhealthy instances are removed from service and terminated after service owners identify the root cause of the problem**.

4.10.7.4. **Janitor Monkey**
It ensures the cloud environment runs **without clutter and waste**. It also searches for **unused resources and discards them**.

4.10.7.5. **Security Monkey**
An extension of Conformity Monkey, it identifies **security violations or vulnerabilities** (e.g., improperly configured AWS security groups) and **eliminates the offending instances**. It also ensures the SSL (Secure Sockets Layer) and DRM (Digital Rights Management) certificates were valid and not due for renewal.

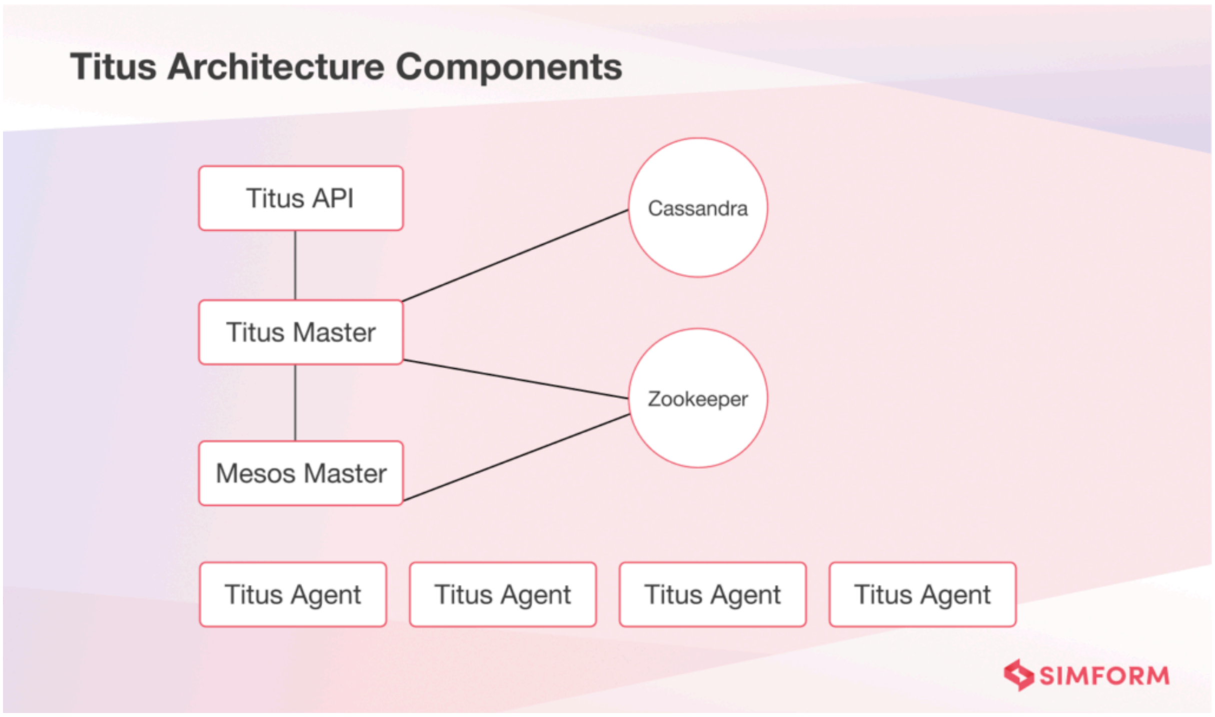
4.10.7.6. **10-18 Monkey**
Short for Localization-Internationalization, it identifies configuration and runtime issues in instances serving users in multiple geographic locations with different languages and character sets.

4.10.7.7. **Chaos Gorilla**
Like Chaos Monkey, the **Gorilla simulates an outage of a whole Amazon availability zone** to verify if the services automatically re-balance to the functional availability zones without manual intervention or any visible impact on users.

4.10.8. Today, Netflix still uses Chaos Engineering and has a dedicated team for chaos experiments called the Resilience Engineering team (earlier called the Chaos team).

4.10.9. For Container Orchestration Netflix uses **Titus** that manages containers and provides integrations to the infrastructure ecosystem. This repository contains the Titus API Swagger/Protobuf/GRPC IDLs.

4.10.10. Titus provided a scalable and reliable container execution solution to Netflix and seamlessly integrated with AWS. In addition, it enabled easy deployment of containerized batches and service applications.



4.10.11. Titus served as a standard deployment unit and a generic batch job scheduling system. It helped Netflix expand support to growing batch use cases.
Batch users could also put together sophisticated infrastructure quickly and pack larger instances across many workloads efficiently. **Batch users** could immediately schedule locally developed code for scaled execution on Titus.
(A user account created to run automated tasks, scripts, or background jobs (also called batch jobs) usually on a schedule or based on triggers without requiring manual intervention.)

4.10.12. Netflix adopted the "**Operate what you build**" model, traditional Model consist of siloed teams which focused on specific aspect of SDLC (opration team focus on deplyment).
In **Operate what you build** model, The teams developing a system were responsible for operating and supporting it. Each team owned its own deployment issues, performance bugs, alerting gaps, capacity planning, partner support. They also introduced centralized tooling to simplify and automate dealing with common development problems of the teams

4.11. Lessons learned from Netflix's DevOps strategy

4.11.1. Don't build systems that say no to your developers
4.11.2. Focus on giving freedom and responsibility to the engineers
4.11.3. Don't think about uptime at all costs

4.11.4. Prize the velocity of innovation
4.11.5. Eliminate a lot of processes and procedures
4.11.6. Practice context over control
4.11.7. Don’t do a lot of required standards, but focus on enablement
4.11.8. Don’t do silos, walls, and fences
4.11.9. Adopt “you build it, you run it” culture
4.11.10. Focus on data
4.11.11. Always put customer satisfaction first
4.11.12. Don’t do DevOps, but focus on the culture

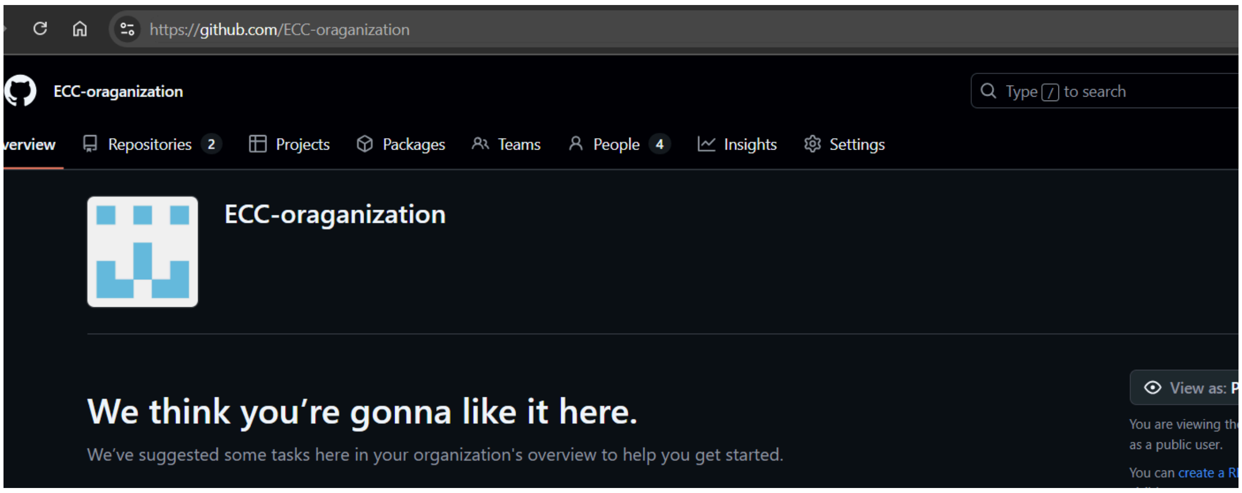
5. Git Branching Strategy

Common commands used with git:

Basic	
git init	- Initialize a new Git repository
git clone <repo>	- Clone a repository
git status	- Show working tree status
git add <file>	- Stage file(s) for commit
git commit -m "msg"	- Commit staged changes
git log	- View commit logs
git diff	- Show changes
Branching & Merging	
git branch	- List branches
git branch <name>	- Create a new branch
git checkout <branch>	- Switch to a branch
git merge <branch>	- Merge a branch into current
git rebase <branch>	- Reapply commits on top of another branch
Remote Repo	
git remote -v	- Show remote URLs
git remote add origin <url>	- Add a new remote
git push	- Push changes
git pull	- Pull latest changes
git fetch	- Fetch remote changes
Undo & Reset	
git checkout -- <file>	- Discard file changes
git reset --hard	- Hard reset
git revert <commit>	- Revert a commit
GitHub Workflow	
git fork	- Fork a repo (on GitHub)
git clone <forked_repo>	- Clone your fork
git remote add upstream <url>	- Add original repo
git fetch upstream	- Fetch latest from original
git pull upstream main	- Sync fork
git push origin <branch>	- Push feature branch
Create PR on GitHub	- Open a Pull Request

5.1. Create a GitHub Organization and Add Members

5.1.1. Go to github.com → click on your profile → "Your organizations" → "New organization".
5.1.2. Choose a plan (Free/Pro/Enterprise).
5.1.3. Add team members with roles: Owner, Admin, or Member.



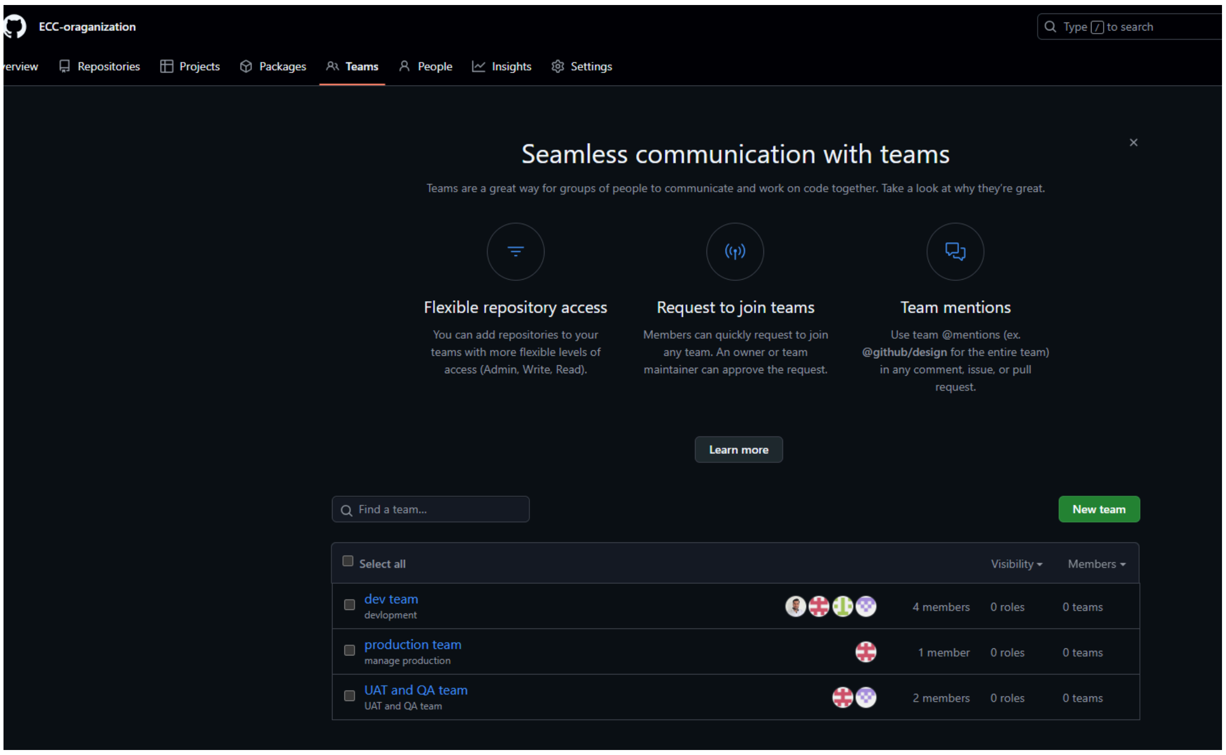
ECC-oraganization

We think you're gonna like it here.

We've suggested some tasks here in your organization's overview to help you get started.

You are viewing th
as a public user.
You can create a R

Create organization on github.



Seamless communication with teams

Teams are a great way for groups of people to communicate and work on code together. Take a look at why they're great.

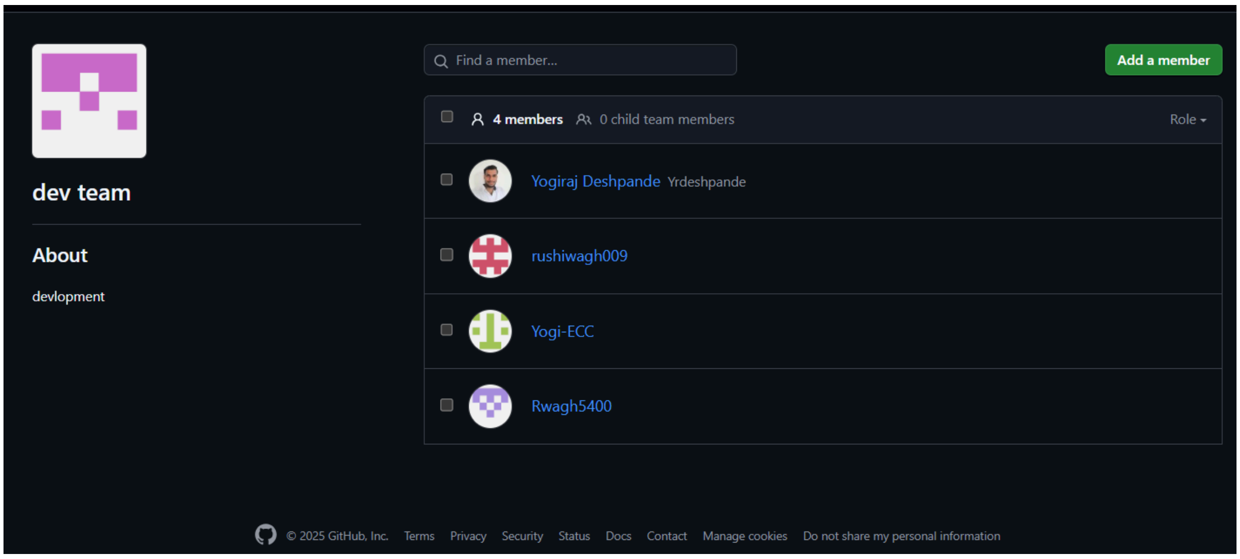
Flexible repository access
You can add repositories to your teams with more flexible levels of access (Admin, Write, Read).

Request to join teams
Members can quickly request to join any team. An owner or team maintainer can approve the request.

Team mentions
Use team @mentions (ex. **@github/design** for the entire team) in any comment, issue, or pull request.

Select all	Visibility ▾	Members ▾		
dev team devlopment	4 members	0 roles	0 teams	
production team manage production	1 member	0 roles	0 teams	
UAT and QA team UAT and QA team	2 members	0 roles	0 teams	

Create teams like dev, production, UAT and QA.


dev team

About

devlopment

Find a member...

Add a member

4 members 0 child team members Role

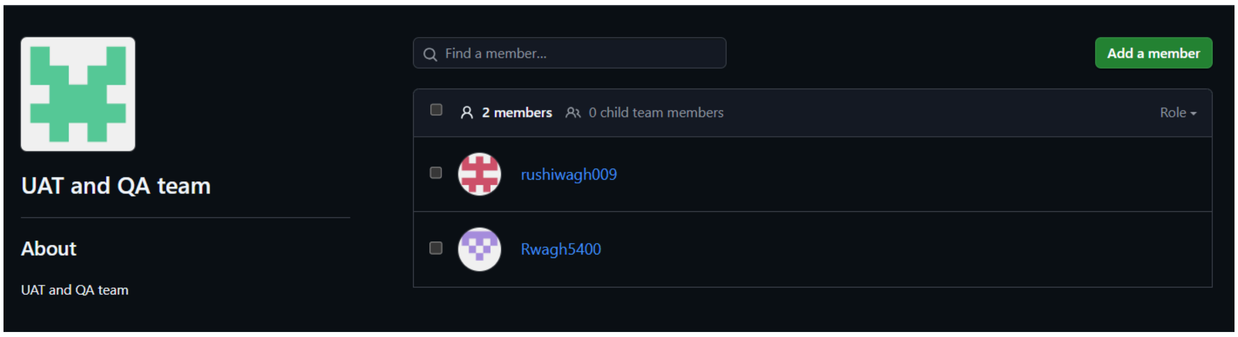
Yogiraj Deshpande Yrdeshpande

rushiwagh009

Yogi-ECC

Rwagh5400

© 2025 GitHub, Inc. Terms Privacy Security Status Docs Contact Manage cookies Do not share my personal information


UAT and QA team

About

UAT and QA team

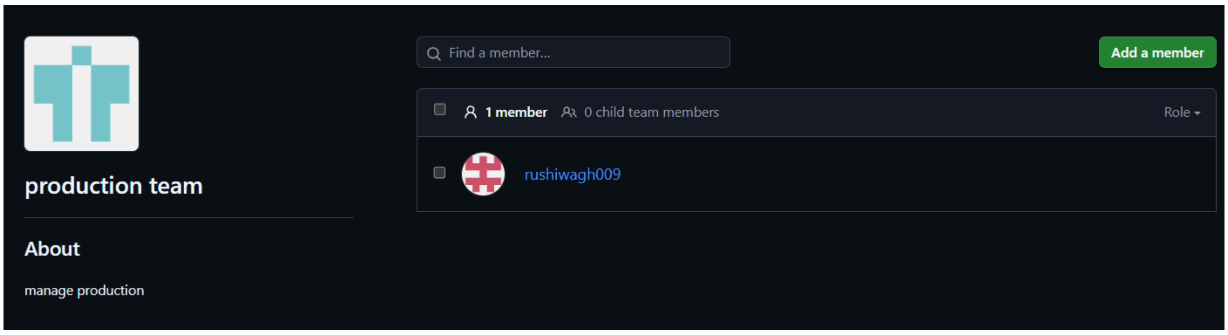
Find a member...

Add a member

2 members 0 child team members Role

rushiwagh009

Rwagh5400


production team

About

manage production

Find a member...

Add a member

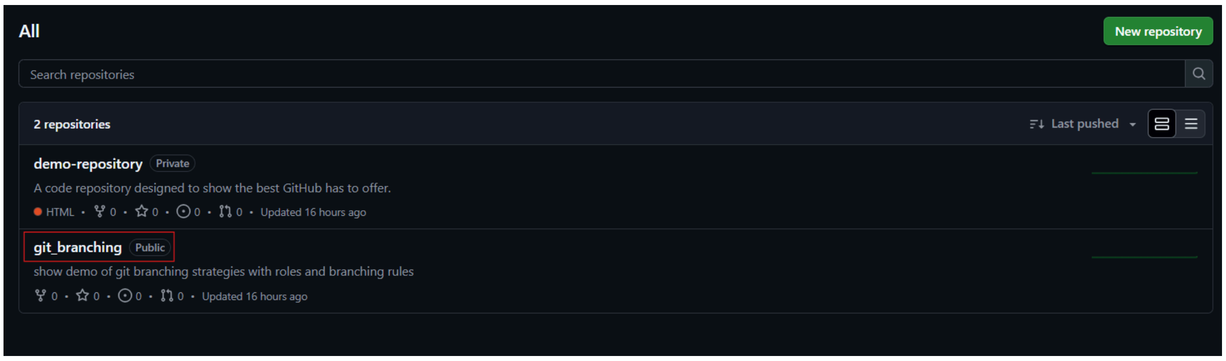
1 member 0 child team members Role

rushiwagh009

Assign members and maintainers to associated teams

5.2. Create a Repository in the Organization

- 5.2.1. Navigate to your organization and click "New Repository".
- 5.2.2. Choose visibility (public/private).
- 5.2.3. Initialize with a README.md or .gitignore if needed


All

New repository

Search repositories

2 repositories

Last pushed

demo-repository Private

A code repository designed to show the best GitHub has to offer.

HTML · 0 · 0 · 0 · 0 · Updated 16 hours ago

git_branching Public

show demo of git branching strategies with roles and branching rules

0 · 0 · 0 · 0 · Updated 16 hours ago

Create a repository in the organization, either public or private, based on the requirement.

5.3. Define and Create Branches from the main

5.3.1. Establish your branching model. A common strategy is a simplified version of Git **Flow**:

5.3.1.1.	**Main -**	Production-ready code
5.3.1.2.	**Develop-**	Integration of all features. Used by devs.
5.3.1.3.	**QA -**	Code is ready for QA testing.
5.3.1.4.	**UAT-**	Code ready for User Acceptance Testing.
5.3.1.5.	**feature/*-**	Individual features (e.g., feature/home, feature/login).

5.3.1.6. Command for local host :

# repeat for QA, UAT, etc

git checkout main

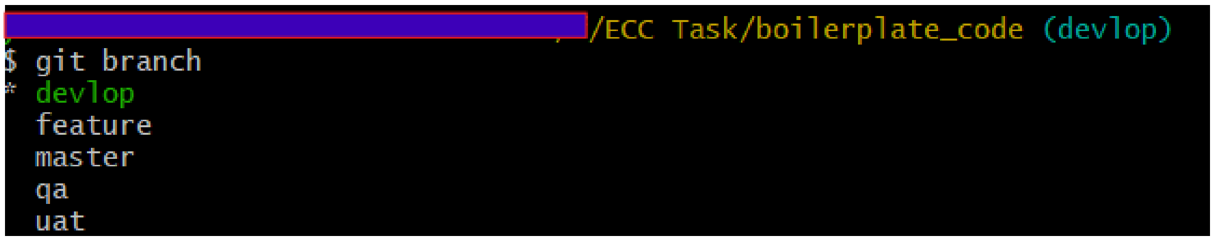
git checkout -b develop

git push -u origin develop

Alternative:

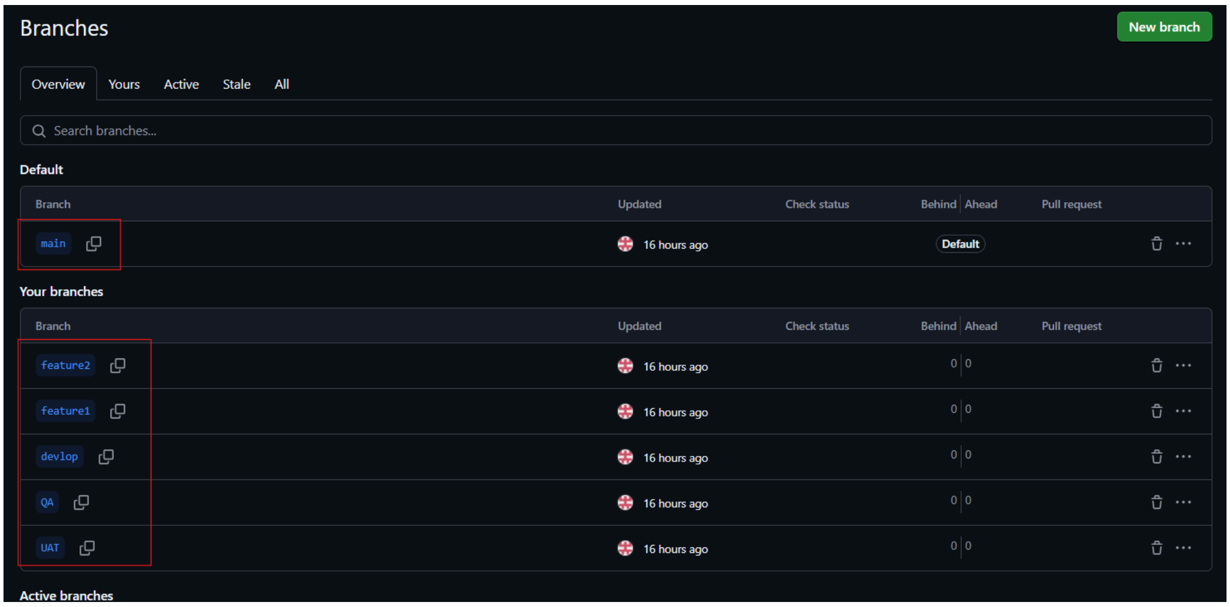
Create branches through GitHub

And clone the repo on localhost



```
[redacted]/ECC Task/boilerplate_code (devlop)
$ git branch
* devlop
  feature
  master
  qa
  uat
```

Create feature, develop, qa, UAT, master branch in git.



Branches

New branch

Overview Yours Active Stale All

Search branches...

Default

Branch	Updated	Check status	Behind | Ahead	Pull request
main	16 hours ago		Default	

Your branches

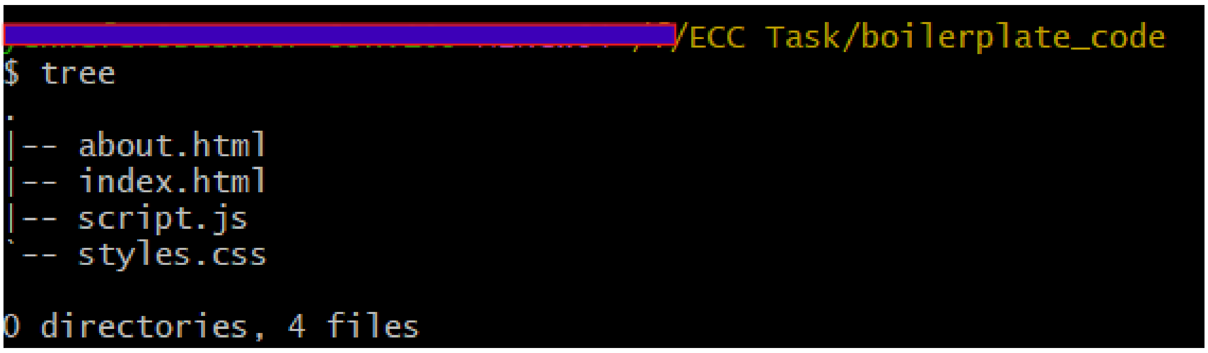
Branch	Updated	Check status	Behind | Ahead	Pull request
feature2	16 hours ago		0 | 0	
feature1	16 hours ago		0 | 0	
devlop	16 hours ago		0 | 0	
QA	16 hours ago		0 | 0	
UAT	16 hours ago		0 | 0	

Active branches

Create feature, develop, qa, UAT, master branch, use main branch as source branch in GitHub (if required).

5.4. Push Boilerplate Code

5.4.1. Boilerplate code refers to standardized, reusable pieces of code that are commonly used as a starting point in new projects.
(choco install tree — install tree command for Windows)

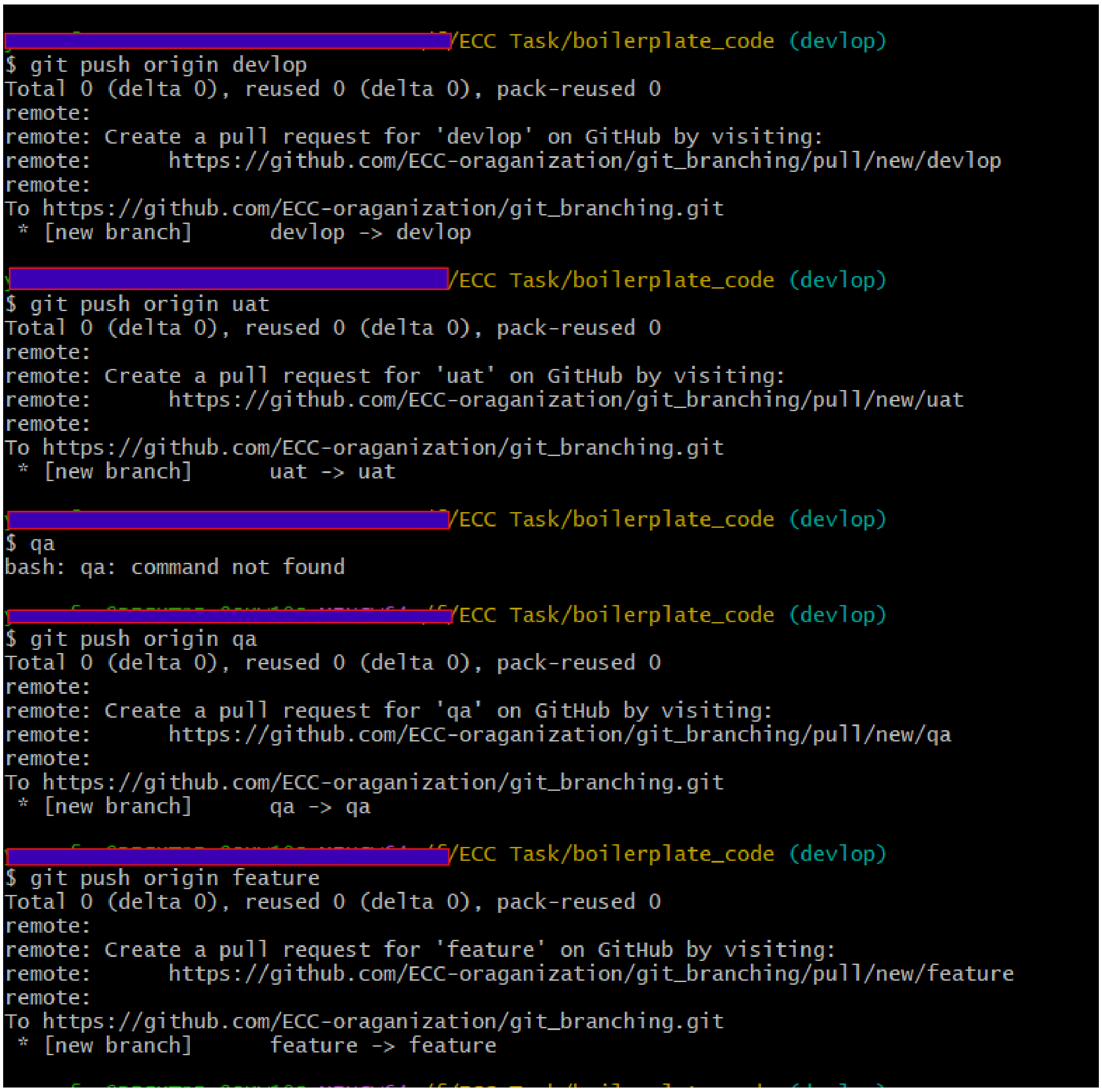


```
[redacted]/ECC Task/boilerplate_code
$ tree
.
|-- about.html
|-- index.html
|-- script.js
`-- styles.css

0 directories, 4 files
```

File system of boilerplate code

5.4.2. Add the project’s initial codebase
Command:

```
cd ./boilerplate_code
git init
git add .
git commit “boilerplate_code push”
git push origin feature
git push origin develop
git push origin qa
git push origin uat
git push origin master
```



```
[redacted]/ECC Task/boilerplate_code (devlop)
$ git push origin devlop
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'devlop' on GitHub by visiting:
remote:      https://github.com/ECC-oraganization/git_branching/pull/new/devlop
remote:
To https://github.com/ECC-oraganization/git_branching.git
 * [new branch]      devlop -> devlop

[redacted]/ECC Task/boilerplate_code (devlop)
$ git push origin uat
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'uat' on GitHub by visiting:
remote:      https://github.com/ECC-oraganization/git_branching/pull/new/uat
remote:
To https://github.com/ECC-oraganization/git_branching.git
 * [new branch]      uat -> uat

[redacted]/ECC Task/boilerplate_code (devlop)
$ qa
bash: qa: command not found

[redacted]/ECC Task/boilerplate_code (devlop)
$ git push origin qa
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'qa' on GitHub by visiting:
remote:      https://github.com/ECC-oraganization/git_branching/pull/new/qa
remote:
To https://github.com/ECC-oraganization/git_branching.git
 * [new branch]      qa -> qa

[redacted]/ECC Task/boilerplate_code (devlop)
$ git push origin feature
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'feature' on GitHub by visiting:
remote:      https://github.com/ECC-oraganization/git_branching/pull/new/feature
remote:
To https://github.com/ECC-oraganization/git_branching.git
 * [new branch]      feature -> feature
```

Push boilerplate code to each branch.

6. Resolve Merge Conflicts

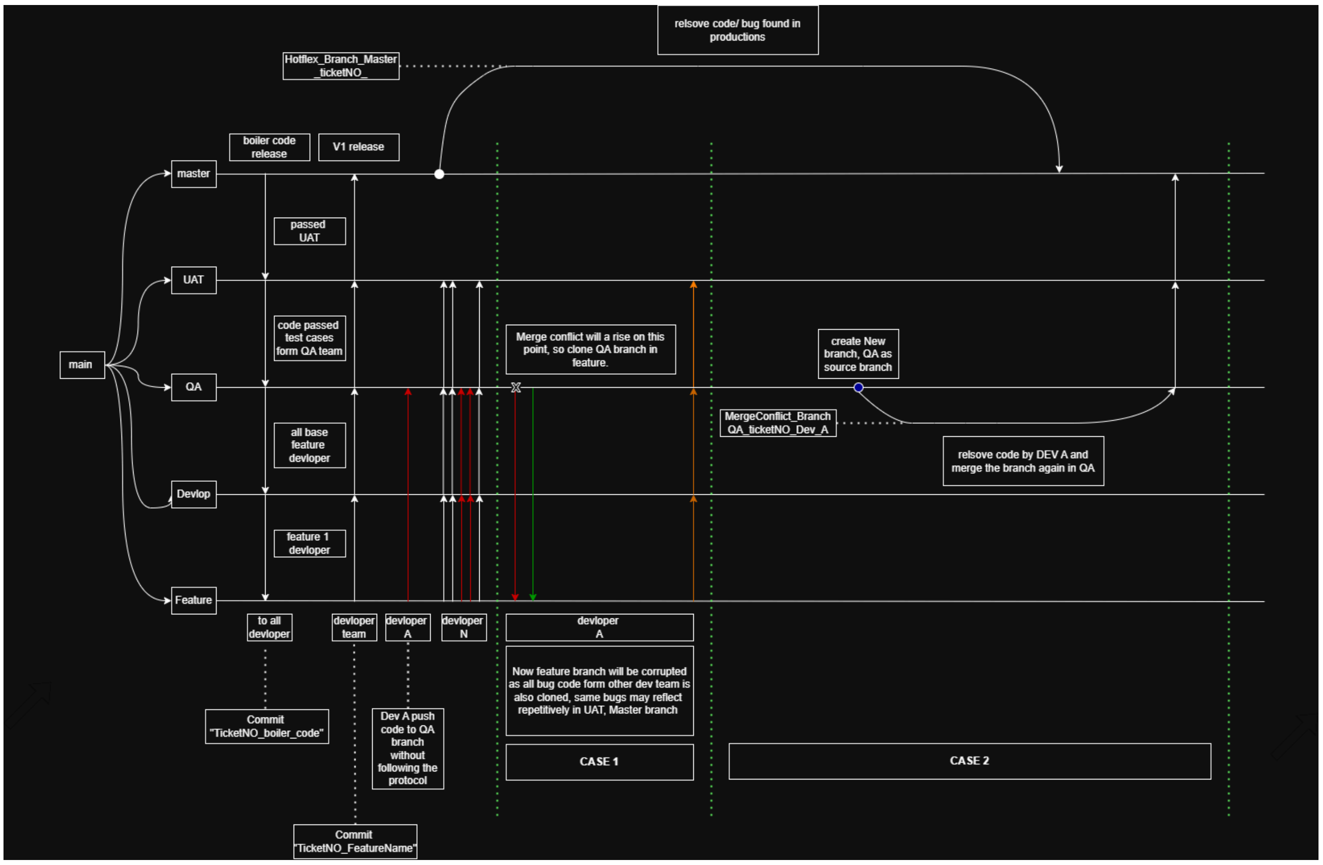
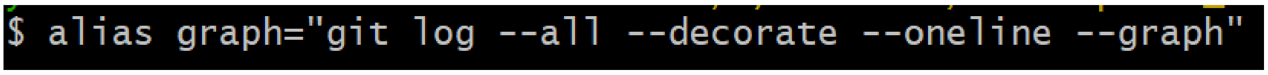

relsove code/ bug found in productions
Hotflex_Branch_Master _ticketNO_
boiler code release
V1 release
master
passed UAT
UAT
code passed test cases form QA team
main
QA
all base feature devloper
Devlop
feature 1 devloper
Feature
to all devloper
devloper team
devloper A
devloper N
Commit "TicketNO_boiler_code"
Dev A push code to QA branch without following the protocol
Commit "TicketNO_FeatureName"
Merge conflict will a rise on this point, so clone QA branch in feature.
devloper A
Now feature branch will be corrupted as all bug code form other dev team is also cloned, same bugs may reflect repetitively in UAT, Master branch
CASE 1
create New branch, QA as source branch
MergeConflict_Branch QA_ticketNO_Dev_A
relsove code by DEV A and merge the branch again in QA
CASE 2

Diagram of branching strategies and merge conflict cases.

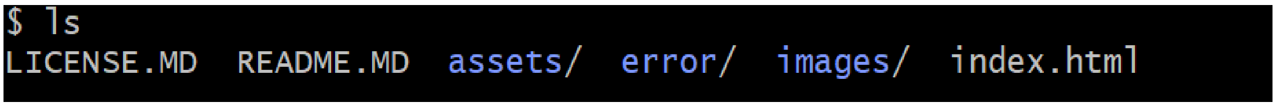
6.1. CASE 1

6.1.1. **Situation:** DEV A has pushed changes to the QA branch directly, simultaneously, other developers (i.e, DEV N) have pushed code to the develop branch and then to the QA branch.
Now DEV A gets a merge conflict while merging the Branch QA and Feature.
Solution: DEV A clone QA branch to Feature Branch and resolve the code, and again push it to Devlop Branch and QA branch
Observation: Bugs present in the QA branch due to DEV N development in code are cloned in the Feature branch, due to which the Feature Branch is corrupted for DEV A, In this situation, we have to pick Feature/ Branch, which consists of minimum bugs, which leads to a delay in production.



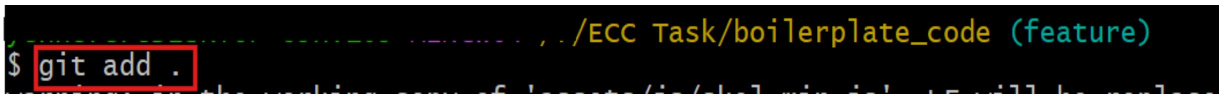
```
$ alias graph="git log --all --decorate --oneline --graph"
```

Assign graph as command “git log –all –decorate –oneline –graph”
We can directly type this command or just type “graph”, both will give the same results.



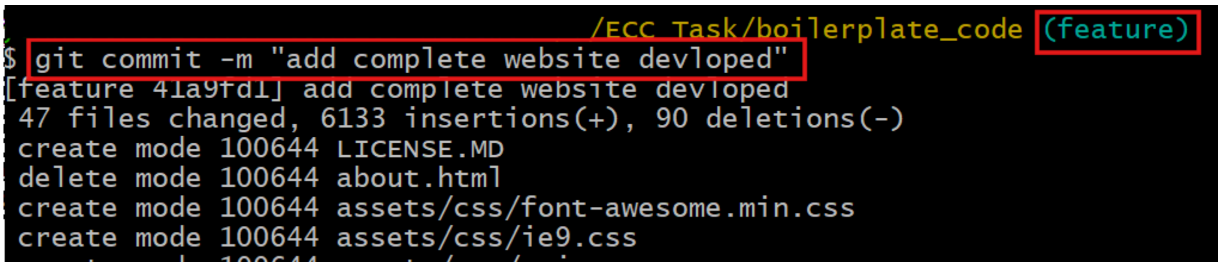
```
$ ls
LICENSE.MD  README.MD  assets/  error/  images/  index.html
```

Complete website code in feature branch



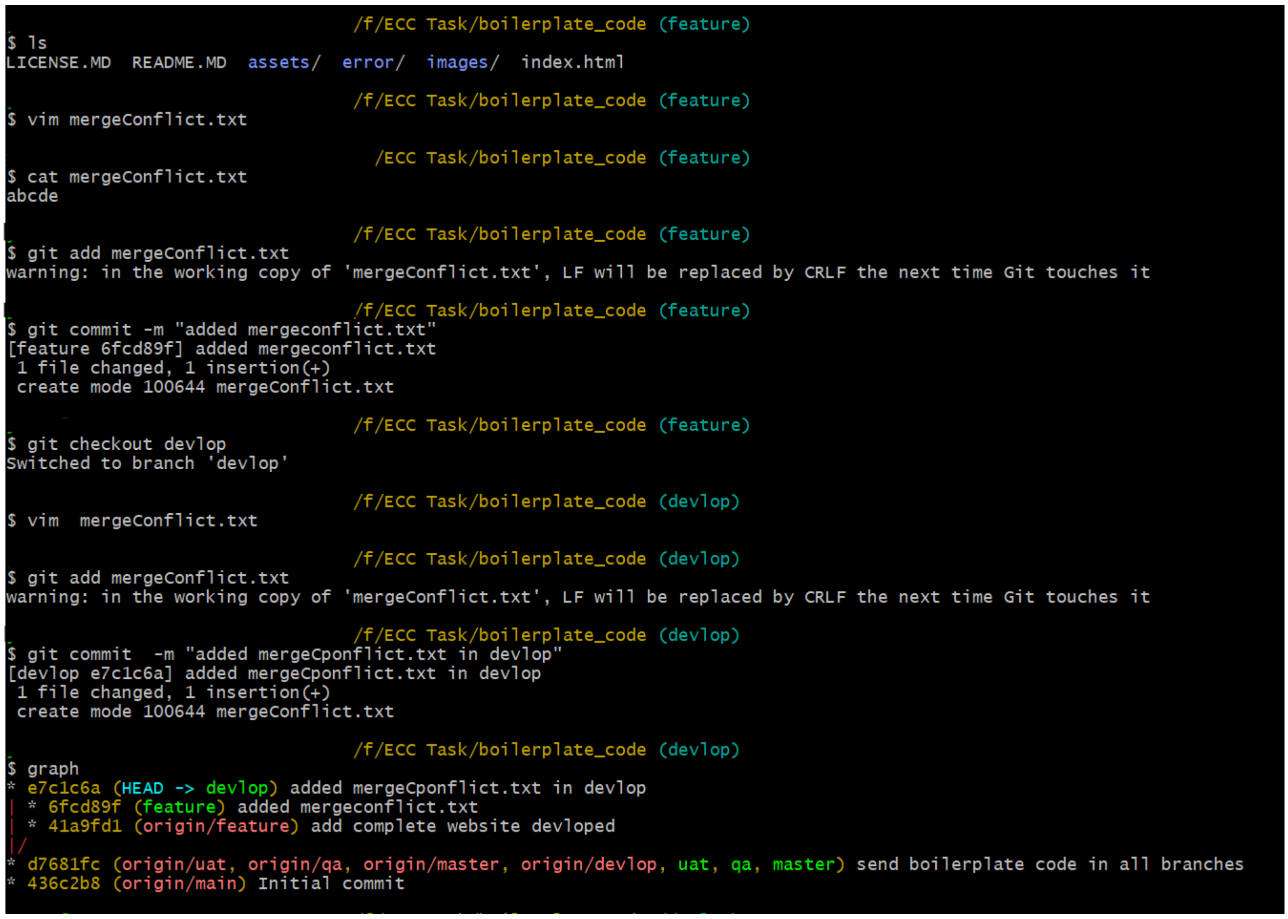
```
                                        /ECC Task/boilerplate_code (feature)
$ git add .
```

Added all website code in staging area.



```
                                        /ECC Task/boilerplate_code (feature)
$ git commit -m "add complete website devloped"
[feature 41a9fd1] add complete website devloped
 47 files changed, 6133 insertions(+), 90 deletions(-)
 create mode 100644 LICENSE.MD
 delete mode 100644 about.html
 create mode 100644 assets/css/font-awesome.min.css
 create mode 100644 assets/css/ie9.css
```

Committed code in local git repo (feature)

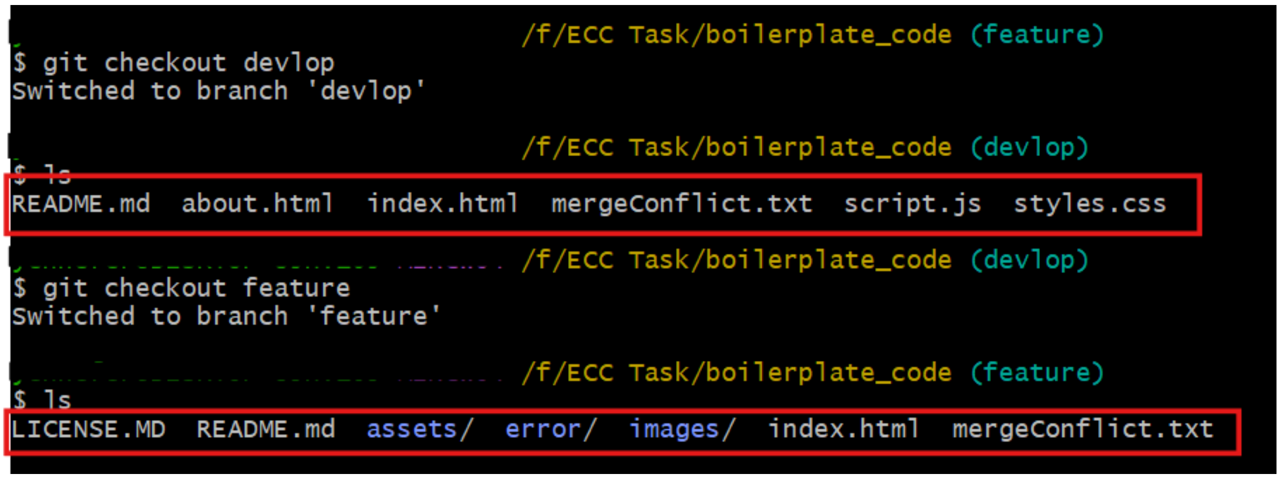


```
                                  /f/ECC Task/boilerplate_code (feature)
$ ls
LICENSE.MD  README.MD  assets/  error/  images/  index.html

                                  /f/ECC Task/boilerplate_code (feature)
$ vim mergeConflict.txt

                                    /ECC Task/boilerplate_code (feature)
$ cat mergeConflict.txt
abcde

                                  /f/ECC Task/boilerplate_code (feature)
$ git add mergeConflict.txt
warning: in the working copy of 'mergeConflict.txt', LF will be replaced by CRLF the next time Git touches it

                                  /f/ECC Task/boilerplate_code (feature)
$ git commit -m "added mergeconflict.txt"
[feature 6fcd89f] added mergeconflict.txt
 1 file changed, 1 insertion(+)
 create mode 100644 mergeConflict.txt

                                  /f/ECC Task/boilerplate_code (feature)
$ git checkout devlop
Switched to branch 'devlop'

                                  /f/ECC Task/boilerplate_code (devlop)
$ vim  mergeConflict.txt

                                  /f/ECC Task/boilerplate_code (devlop)
$ git add mergeConflict.txt
warning: in the working copy of 'mergeConflict.txt', LF will be replaced by CRLF the next time Git touches it

                                  /f/ECC Task/boilerplate_code (devlop)
$ git commit  -m "added mergeCponflict.txt in devlop"
[devlop e7c1c6a] added mergeCponflict.txt in devlop
 1 file changed, 1 insertion(+)
 create mode 100644 mergeConflict.txt

                                  /f/ECC Task/boilerplate_code (devlop)
$ graph
* e7c1c6a (HEAD -> devlop) added mergeCponflict.txt in devlop
| * 6fcd89f (feature) added mergeconflict.txt
| * 41a9fd1 (origin/feature) add complete website devloped
|/
* d7681fc (origin/uat, origin/qa, origin/master, origin/devlop, uat, qa, master) send boilerplate code in all branches
* 436c2b8 (origin/main) Initial commit
```

Created a mergeConflict.txt file in the feature branch local.
Content of mergeconflict.txt = abcde
Added the mergeConflict.txt file in the staging area
Committed mergeConflict.txt file in feature branch local.

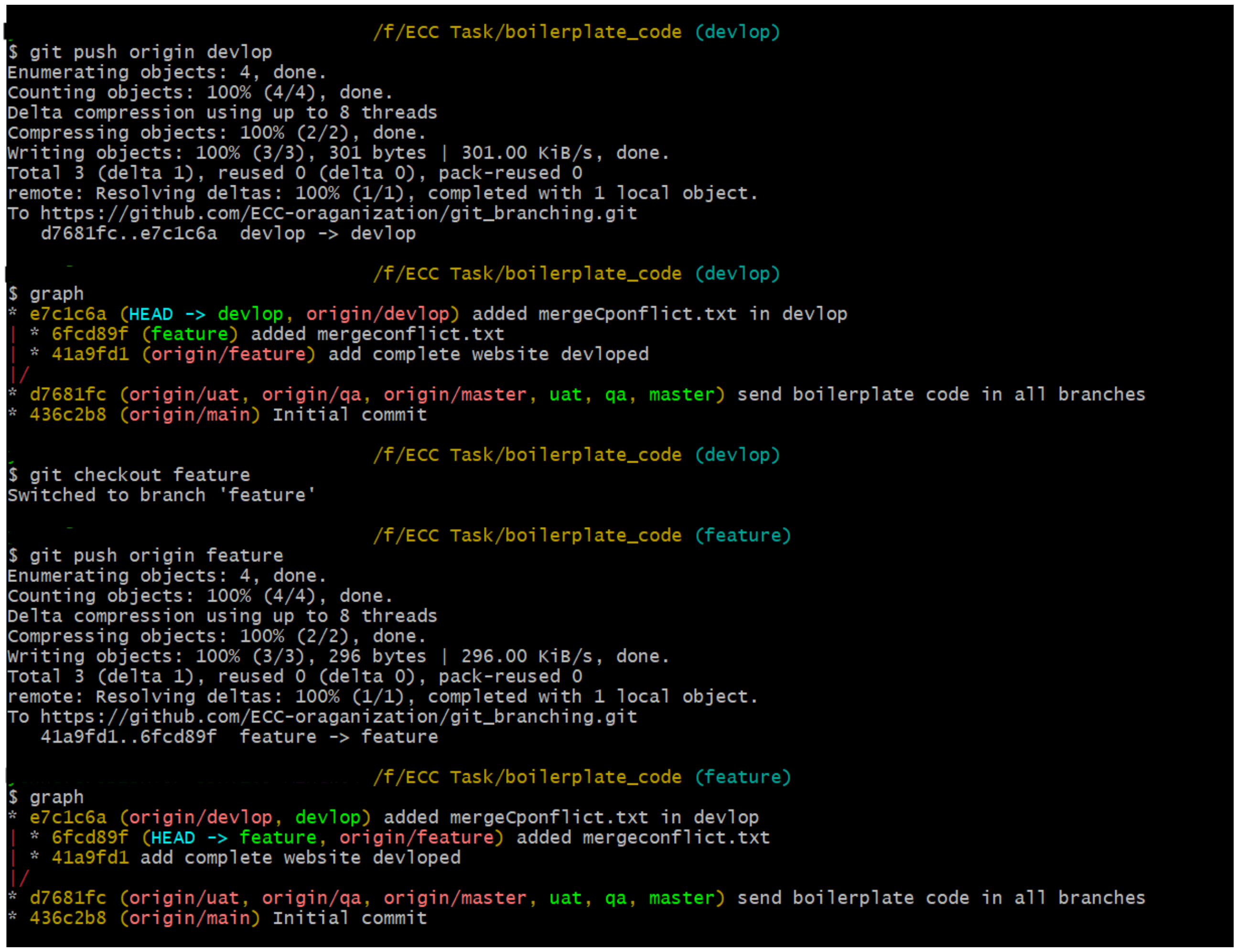
Created a mergeConflict.txt file in the devlop branch local.
Content of mergeconflict.txt = efghi

Added the mergeConflict.txt file in the staging area
Committed mergeConflict.txt file in Devlop branch local.



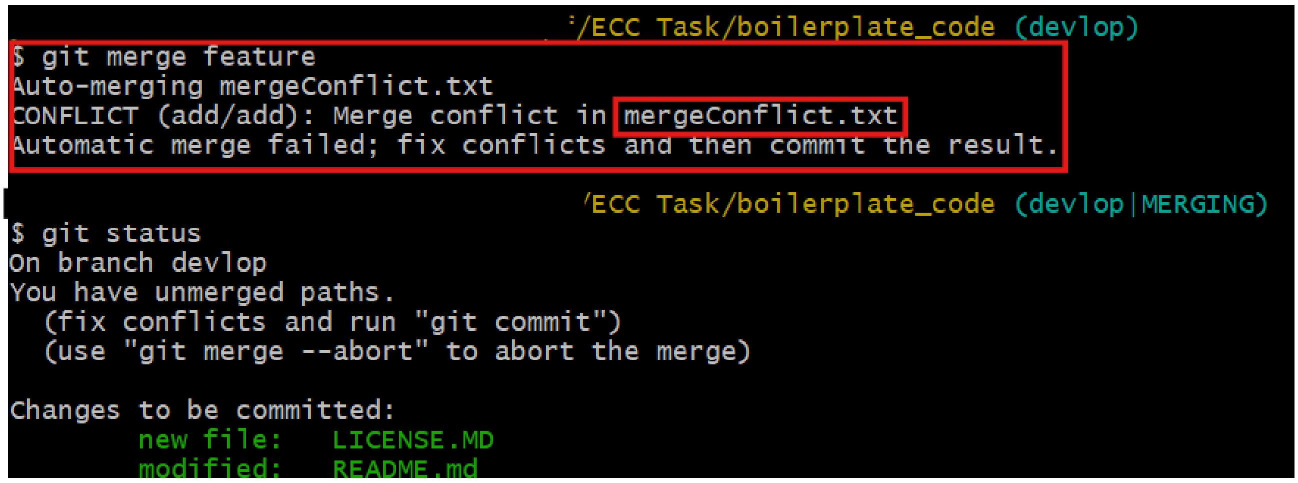
```
[redacted]                    /f/ECC Task/boilerplate_code (feature)
$ git checkout devlop
Switched to branch 'devlop'

[redacted]                    /f/ECC Task/boilerplate_code (devlop)
$ ls
README.md  about.html  index.html  mergeConflict.txt  script.js  styles.css

[redacted]                    /f/ECC Task/boilerplate_code (devlop)
$ git checkout feature
Switched to branch 'feature'

[redacted]                    /f/ECC Task/boilerplate_code (feature)
$ ls
LICENSE.MD  README.md  assets/  error/  images/  index.html  mergeConflict.txt
```

File system of branch develop and feature are different



```
[redacted]                    /f/ECC Task/boilerplate_code (devlop)
$ git push origin devlop
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 301 bytes | 301.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ECC-oraganization/git_branching.git
   d7681fc..e7c1c6a  devlop -> devlop

[redacted]                    /f/ECC Task/boilerplate_code (devlop)
$ graph
* e7c1c6a (HEAD -> devlop, origin/devlop) added mergeCponflict.txt in devlop
| * 6fcd89f (feature) added mergeconflict.txt
| * 41a9fd1 (origin/feature) add complete website devloped
|/
* d7681fc (origin/uat, origin/qa, origin/master, uat, qa, master) send boilerplate code in all branches
* 436c2b8 (origin/main) Initial commit

[redacted]                    /f/ECC Task/boilerplate_code (devlop)
$ git checkout feature
Switched to branch 'feature'

[redacted]                    /f/ECC Task/boilerplate_code (feature)
$ git push origin feature
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 296 bytes | 296.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ECC-oraganization/git_branching.git
   41a9fd1..6fcd89f  feature -> feature

[redacted]                    /f/ECC Task/boilerplate_code (feature)
$ graph
* e7c1c6a (origin/devlop, devlop) added mergeCponflict.txt in devlop
| * 6fcd89f (HEAD -> feature, origin/feature) added mergeconflict.txt
| * 41a9fd1 add complete website devloped
|/
* d7681fc (origin/uat, origin/qa, origin/master, uat, qa, master) send boilerplate code in all branches
* 436c2b8 (origin/main) Initial commit
```

Both the branches' changes are pushed to the remote repo(Github)



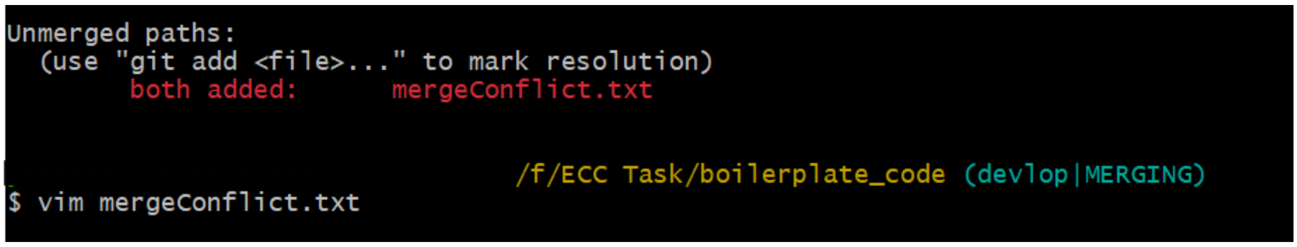
```
                          /ECC Task/boilerplate_code (devlop)
$ git merge feature
Auto-merging mergeConflict.txt
CONFLICT (add/add): Merge conflict in mergeConflict.txt
Automatic merge failed; fix conflicts and then commit the result.
                          /ECC Task/boilerplate_code (devlop|MERGING)
$ git status
On branch devlop
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   LICENSE.MD
        modified:   README.md
```

Now we get a merge conflict if we merge the develop branch and the feature branch.
Because of the mergeconflict.txt file
Note: during merging to branches

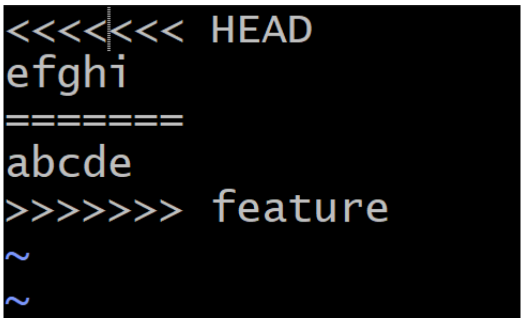
To merge a branch (develop) with no Change file and a branch (feature) with changed files (eg, images, assets), the changed files with be pushed in the develop branch to merge feature and develop.

To merge a branch (develop) and (feature) consisting of the same file (mergeconflict.txt) but in have different content (code) in different branches, then a merge conflict will occur.

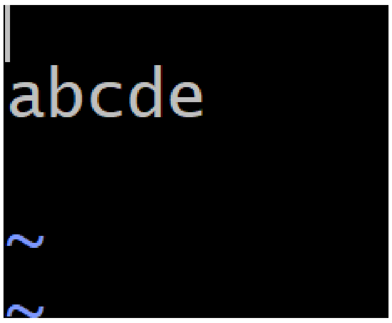
To resolve that conflict, we need to change the content of the files according to the requirement.



```
Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both added:      mergeConflict.txt

                          /f/ECC Task/boilerplate_code (devlop|MERGING)
$ vim mergeConflict.txt
```

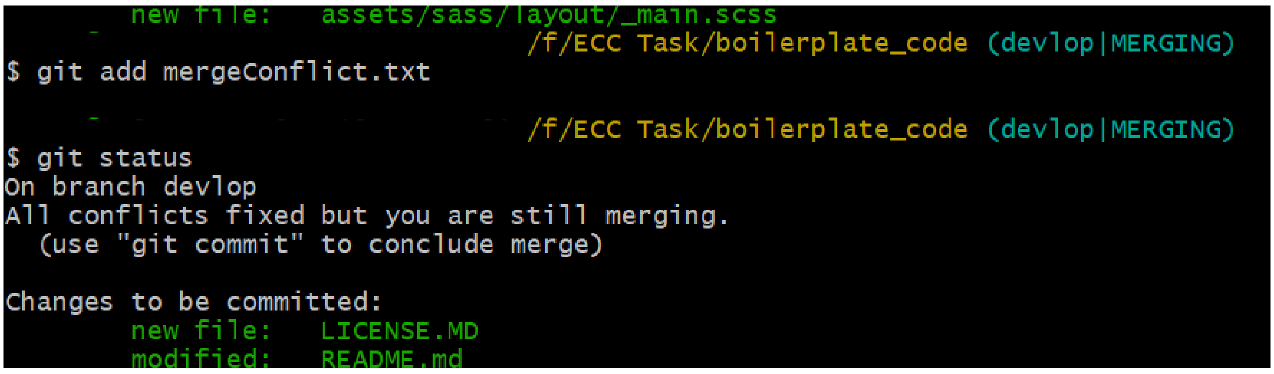


```
<<<<<<< HEAD
efghi
=======
abcde
>>>>>>> feature
~
~
```



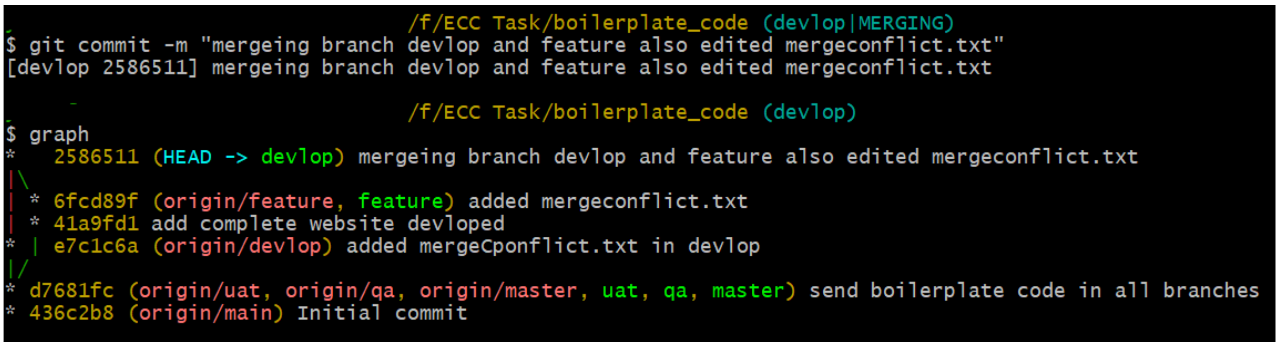
```
abcde

~
~
```

Changed the content of the file according to the requirement in mergeconflict.txt
<<<<<<< HEAD means the content of the develop branch because we are in the develop branch (HEAD pointer points to Develop branch)
<<<<<<< Feature means the content of the feature branch
=========== separate the contents of the file that are different in the branches.



```
        new file:   assets/sass/layout/_main.scss
                                     /f/ECC Task/boilerplate_code (devlop|MERGING)
$ git add mergeConflict.txt

                                     /f/ECC Task/boilerplate_code (devlop|MERGING)
$ git status
On branch devlop
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        new file:   LICENSE.MD
        modified:   README.md
```

Add mergeconflict.txt to the staging area



```
                                   /f/ECC Task/boilerplate_code (devlop|MERGING)
$ git commit -m "mergeing branch devlop and feature also edited mergeconflict.txt"
[devlop 2586511] mergeing branch devlop and feature also edited mergeconflict.txt

                                   /f/ECC Task/boilerplate_code (devlop)
$ graph
*   2586511 (HEAD -> devlop) mergeing branch devlop and feature also edited mergeconflict.txt
|\
| * 6fcd89f (origin/feature, feature) added mergeconflict.txt
| * 41a9fd1 add complete website devloped
* | e7c1c6a (origin/devlop) added mergeCponflict.txt in devlop
|/
* d7681fc (origin/uat, origin/qa, origin/master, uat, qa, master) send boilerplate code in all branches
* 436c2b8 (origin/main) Initial commit
```

Commit mergeconflict.txt, and the merge conflict is resolved.

6.2. CASE 2

6.2.1. **Situation:** DEV A has pushed changes to the QA branch directly, simultaneously, other developers (i.e, DEV N) have pushed code to the develop branch and then to the QA branch.
Now DEV A gets a merge conflict while merging the Branch QA and Feature.
Solution: Create a Branch MergeConflict_Branch_QA_ticketNO_Dev_A using QA as source branch, make required changes on MergeConflict_Branch_QA_ticketNO_Dev_A, once changes are done merge MergeConflict_Branch_QA_ticketNO_Dev_A with QA branch
Observation: bugs present in the QA branch didn't affect the feature branch, the developer team doesn't need to cherry-pick the feature/ branch for fewer bugs.