Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Roger-that flow: um uso aprimorado do git-flow

Daniel Almeida Luz

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Roger-that flow: um uso aprimorado do git-flow

Daniel Almeida Luz

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio (Orientador)
CIC/UnB

Prof. Dr. Edna Dias Canedo
CIC/UnB

Prof. Dr. Edson Alves da Costa Júnior
FGA/UnB

do Bacharelado em Ciência da Computação

Brasília, 9 de julho de 2018

Dedicatória

Ao “Bora trancar”, que, contrário ao nome, fez com que eu não trancasse o curso. Às pessoas da Central, que criaram experiências pessoais que não podem ser descritas aqui, por diversos motivos... Às minhas famílias (de sangue e não), que me suportaram durante todo o período, mais do que elas tem noção.

Agradecimentos

Agradeço à equipe do SEPAS, meu primeiro estágio no Tribunal de Contas da União, que me proporcionou o contato com assuntos avançados e aprendizados que valem até hoje, não só para este trabalho, mas para grande parte das minhas experiências profissionais e pessoais.

Resumo

Neste trabalho buscamos aperfeiçoar o ciclo de desenvolvimento de software voltado para o desenvolvimento de produtos. Por meio de um incremento ao git-flow, um modelo de organização para o versionamento de código, utilizando uma parte da cultura DevOps, além de gerar ganhos técnicos no que diz respeito à instabilidade do sistema, conseguimos um processo mais efetivo entre times de desenvolvimento e negociais. Adotando um modelo que faz uso de ferramentas e processos existentes em um cenário isolado, mas bastante replicável, criamos um processo simples e com baixo custo de implantação. Por meio de uma metodologia de pesquisa baseada em *action research* e uma análise de grupos focais no cenário aplicado, validamos uma ideia que a princípio foi voltada para ganhos puramente técnicos, e encontramos benefícios consideráveis até o time gerencial. Esse trabalho possibilita uma ótica diferente para o termo DevOps, que passa de uma cultura criada para preencher o espaço entre times de desenvolvimento e operações, para um facilitador de comunicação que alcança o time gerencial da empresa.

Palavras-chave: git, git-flow, software, *deploy* contínuo, DevOps

Abstract

In this work, we improved the software development cycle for product development. By using an increment to the git-flow, a branching model for code versioning, and using a subset of the DevOps culture, we gained not only technical quality regarding system instability, but created a more effective process between development and business teams. Adopting a model that uses existing tools and processes in an isolated scenario, but highly replicable, we created a simple workflow, with little to no overhead added to the team. By using an *action research* methodology, and an analysis of focus groups, we validated an idea that was aimed at purely technical gains, and found considerable benefits to the business team. This work opens a new optic to the DevOps term, that goes from a culture created to fill the gap between development and operations teams, to a culture that also brings the business team closer to the technical communication also.

Keywords: git, git-flow, software development, continuous deployment, DevOps

Contents

1 Introduction **1**
1.1 Problem definition . . . 2
1.2 Objectives . . . 3
1.3 Methodology used . . . 3

2 Background theory **5**
2.1 Version control . . . 5
2.1.1 Centralized Version Control Systems . . . 5
2.1.2 Distributed Version Control Systems . . . 6
2.2 Git . . . 6
2.3 Git-flow . . . 7
2.4 Continuous Integration/Deployment . . . 7
2.5 Devops . . . 7

3 Methodology **9**
3.1 Roger-that flow . . . 9
3.2 Application process . . . 10
3.3 Implementation . . . 11
3.3.1 Initiating . . . 11
3.3.2 Iterating . . . 11
3.3.3 Closing . . . 12
3.3.4 Quantitative analysis . . . 13
3.3.5 Qualitative analysis . . . 15
3.4 Suggested tool specification . . . 18

4 Conclusion **20**

References **22**

List of Figures

1.1 Scenario pictured after both features are merged into develop 2

3.1 First method of detecting unstable commit 13
3.2 Second method of detecting unstable commit 14
3.3 Resolution of unstable commit using Roger. Scenario pictured in Figure 3.1 14
3.4 Resolution of unstable commit by using Roger. Scenario pictured in Figure 3.2 15
3.5 Code that implements the scenarios covered by Roger 19

Chapter 1

Introduction

The git-flow model is a valuable framework for developers. By adopting it, the code base can be properly organized using git branches for different purposes. Integrated with a continuous deployment tool, we can use one branch (e.g. **develop**) as a staging area where business logic can be validated by people other than developers.

However, more than often, business requirements needs to be adapted during the development cycle, and that leaves your **develop** branch in a not production-ready state.

Imagine some company on it's early stages. By relying on constant feedback from it's users, their product is evolving quickly and new features needs to deployed and adapted every week. On a given week, it was defined that two new features were highly required by their product's users, for example: (1) an historic visualization of some user data and (2) a way of downloading the data from a specific table on the system. Each feature was developed by different developers.

The developers are using the git-flow branching model, and as soon as possible, each one starts the development of their defined features. After two days, the download feature (2) was done and the branch was merged to the **develop** branch so it could be tested by the company's business person. The day after this first merge, the historic visualization feature (1) was also done and merged. Both features are now on the **develop** branch waiting to be validated.

Approaching the end of the week, when this new batch of features should be deployed to the end user, the business person began testing if all the logic was done correctly. The historic visualization feature (1) was perfect, but unfortunately, a *bug* was found on the download feature (2). The develop branch is now in a not production-ready state, meaning that it cannot be deployed to production, and producing a commit history that can be pictured like Figure 1.1.

The bug was reported, the developer started working on it, and found that it was going to be a bit more complicated to resolve than he thought. The end of the week

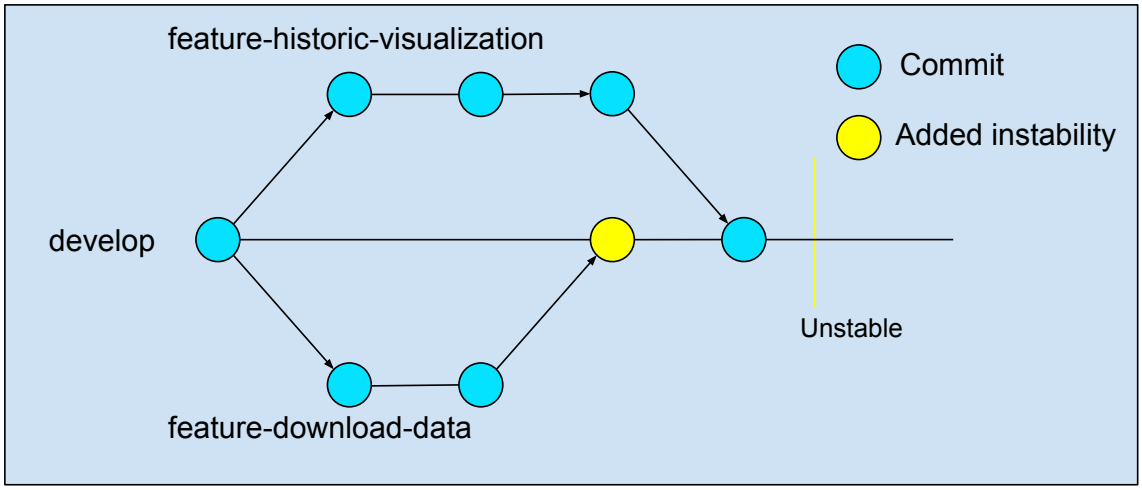

Figure 1.1: Scenario pictured after both features are merged into develop

arrived, and the immediate solution was to delay a deploy to production until **all** features merged into **develop** were validated. Also prevent more feature merges into **develop** while current validations are made, delaying the next development cycle, into a snowball effect.

This work aims to propose a minor adaptation to the git-flow by using a subset of DevOps techniques and tools to increase the effectiveness of the model and the teams productivity. By isolating the feature branch to it's own staging environment, we can fully validate it's business logic before merging it to **develop**, making it a more stable branch, almost always production-ready.

1.1 Problem definition

I currently work at a *startup* that develops a product aimed at processing high loads of data and showing it to the final user in a more pleasant way. Given that, we rely on constant feedback from our users so we can adapt the system properly.

Our development cycle consists on short sprints of development well defined by business necessities. However, as mentioned, user feedback is an important factor when evolving the system and quite often we have to adapt business rules during the sprint.

Being able to adapt a feature development means that, after a feature is apparently done and merged into **develop**, a business person (or someone closer to the end user), can test it and point issues where we, as developers, could not fully address when developing. So we adapt, commit and push, until the feature is satisfied.

The cycle, *code, deploy to develop, validate* can generate a lot of noise into the **develop** branch and leave it too unstable. Unstable enough to delay fully validated features being

deployed to production because other branches are still in the cycle mentioned earlier, **deploy to develop until validated**.

By isolating this cycle into it's own branch, associated to it's own staging environment, we can reduce significantly the number of pull requests of unfinished features into *develop*, and create a more stable environment, almost always ready for production.

For developers, it means more agility and more focus when developing a feature, with less merge conflicts and less stress when deploying to production. For the business as a whole, it means a high throughput of validated features, as well as deadlines less prone to be missed because of instability of features.

1.2 Objectives

The main objective of this work is to improve the software development cycle and feature validation in a *startup* environment by reducing the instability level of the *develop* branch and increasing the throughput of features to the end user. To reach this objective, the following secondary objectives were defined:

- propose a workflow for developers and testers regarding business features when using the git-flow branching model;
- create a tool that automates the process of making features available for testing.

The secondary objectives will be combined aiming to require minimum overhead for a team to adopt this new flow, and by using a subset of DevOps techniques, we will greatly reduce the effort required in the process.

1.3 Methodology used

In this research, we will use a method called *action research*. Action research, according to [1], is a participatory, democratic process concerned with developing practical knowing in the pursuit of worthwhile human purposes, grounded in a participatory worldview which we believe is emerging at this historical moment. It seeks to bring together action and reflection, theory and practice, in participation with others, in the pursuit of practical solutions to issues of pressing concern to people, and more generally the flourishing of individual persons and their communities.

By adopting this method, we focus on the a application of the theory/research to a real problem and ask ourselves some questions like: what are the outcomes of the research? Does it work? Is this application life enhancing?

Based on the study made in [2], we created a simplified model for joining theory and practice that was combined as follows:

- Research theme: the general area of interest was the improvement of the software development process.
- Research framework (F): theory and concepts about the software development using git-flow within agile methodology framed the study, as detailed in the next sections.
- Research methodology (M): the action research methodology used, also adapted from [2], will be detailed on the section related to the application of the research.
- Real world problem situation (A): the research addressed the problem definition section.
- Reflection based on F and M: while working on A, there was a continuous effort to accumulate experiences based on F and M.
- Findings: during and after the application of the research, some points were addressed to be critically reviewed after the conclusion of the process.

Chapter 2

Background theory

This chapter aims to review and provide clarifications of some key concepts related to this proposal. Since we are using an *action based research*, every now and then there will some examples of a concept applied in a real world situation. This application will be done so we can have not only a stronger base on the theory, but a focused and clearer view on how or when a concept is applied into our problem.

2.1 Version control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later [3]. There are several ways that one can implement a version control system (VCS). It can be local, such as copying a file to another directory that is timestamped, and can be remote, every time that a new version is made, you push it to a server that will handle all versions.

Local VCSs is still used today, mostly when the work is done individually. As soon as the necessity to collaborate with other people arrives, local VCS lose it's value, and since our scenario consists in developers collaborating with each other, we will ignore this model and focus on other types of VCS.

The are two main kinds of version control systems: Centralized Version Control Systems (CVCSs) and Distributed Version Control Systems (DVCS).

2.1.1 Centralized Version Control Systems

To deal with the need to collaborate with developers on other systems, CVCSs were developed. These systems (such as CVS[4] and Subversion[5]) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

This system offers certain advantages regarding administration of a unique database of versions and who can do what. However, this setup has one serious downside: the server represents an obvious single point of failure. If the server is down, no one can contribute or save new versions to it. If the server files become corrupted, and there are no proper backups, a great part of the work can be lost.

2.1.2 Distributed Version Control Systems

On the other hand, there are Distributed Version Control Systems (DVCS). In a DVCS (such as Git, Mercurial), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

2.2 Git

Git is a very popular tool among developers used to make version control. It is a DVCS that was designed to be simple, fast, and to strongly support non-linear development (multiple branches of development). Maybe the main difference which makes Git so advantageous over other DVCS is the simplicity when it comes to branching.

The main idea of Git that we will explore is the *branching support*[6]. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In software development terms, it means that we can have a branch that contains only production ready code from the latest release, another one that contains the current release being developed, and several ones for new features on the way.

Of course, in any project that relies on this branching system, some sort of model must be used to organize how branches will be created, what will be the naming convention and how they will be merged, for example. A quite popular model is the git-flow, a model that encourages that each feature is developed on it's own branch and then is merged back into the main branch, basically. Although it is criticized for creating a lot of conflicts between branches because of this isolation, it's really easy and straightforward to use.

2.3 Git-flow

Git-flow[7] is a branching model that consists in separating each feature into it's own development branch. We will focus on a simplified version of the model[8] composed of three main type of branches:

- **master**: main branch where the source code reflects a production-ready state.
- **develop**: branch where the source code reflects a state with the latest delivered development changes.
- **feature branches**: used to develop new features for the upcoming or a distant future release.

After a feature is completed, it is merged into the **develop** branch so the other developers can synchronize their work and keep their own feature branches up-to-date with the rest of the system.

When the develop branch reaches a defined state (e.g. all features that should be developed are already done) and is stable enough, the team or the release manager merges it into the master branch, so proper build and deploy routines are ran and it reaches the user.

2.4 Continuous Integration/Deployment

In 2010, Jez Humble and David Farley defined the term Continuous Delivery (CD). They describe it as a set of practices and principles to release software faster and more frequently. CD extends the workflows and techniques for build and test automation already known as continuous integration (CI). While CI focuses on the automation of the build process on a central server, continuous delivery extends this approach to all workflows needed for the test and deployment of a new build. CD aims to simplify the release of software and allows shorter feedback cycles between developers and customers. Among these benefits, CD enables teams to continuously track the current build state of the software, it reduces integration and configuration errors, lowers stress when dealing with releases and increases the deployment flexibility[9].

2.5 Devops

Coined in Belgium during a 2009 data center migration, the term "devops" sprang from an attempt to apply agile techniques to operations ("ops") activities [10]. We will define

it briefly since we have more focus towards the CI/CD techniques, that are a subset of the DevOps world.

Basically, instead of treating the operations or infrastructure team as a service that exists to supply the development necessities, it will now become a part of the development process and the business as a whole. So the infrastructure behind the business is properly taken into account together with the development planning.

Chapter 3

Methodology

The proposed process is new, created along the evolution of this work. It was fully developed by the researchers.

As said before, the main objective is to improve the software development cycle and feature validation, and we aim to do that by reducing the develop branch instability. For now, we will briefly define that a branch is unstable as: it is not production ready, and thus cannot be deployed to the end user.

3.1 Roger-that flow

This proposal, while it can be applied to different scenarios with minor adjustments, was highly based on the use of existing tools that drastically reduces the overhead of infrastructure operations like build an application, deploy it to a server and expose it to be accessed by some user over the internet.

We will be using the git-flow as our base model, and make a minor change to it. In addition to the simplified model used in [8], we will introduce another type of branch:

- **Staging branches**: A staging branch is very close to a feature branch, the main difference is that once a staging branch is created on the repository, it will trigger a set of automated scripts and provide a way to access the application in the stage it is being developed. So any change can be tested and adjusted on a single branch, isolated from work that is not related to that development.

The whole process was designed to create minimum overhead as possible for developers and testers, and once the tools are properly installed and configured, the majority of the tasks are done automatically so there is close to no difference to working with our base git-flow model. For clarity on how simple it is to be used, the steps to work using the *roger-that* flow from the developer perspective are:

1. define if the new feature will need business validation;
2. create a new **staging** branch;
3. work on it until it's validated;
4. merge it back to the base branch.

Looking closely, it is just like any other **feature** branch, the difference here is what we call a **staging** branch, defined earlier. What defines a **staging** branch can be as simple as using a prefix (e.g. "stg-") on it to mark it different from a feature branch, and that will specify that a set of operations need to run related to that branch.

3.2 Application process

Aiming to reduce the instability level of the **develop** branch, and with that, increasing the throughput of features available to the end user, the main idea of this proposed flow is to fully validate a feature branch while it is being developed. The process was conducted in a collaborative and iterative manner so the team could gradually adopt the new flow.

Our application process was defined as follows:

1. **Initiating**:
   (a) appreciate problem situation;
   (b) study CI/CD solutions that could be applied;
   (c) select simplest solution;
   (d) manually apply the simplest solution whenever possible.
2. **Iterating**:
   (e) improve and automate one step of the solution;
   (f) communicate the new improvement to the team;
   (g) verify acceptance.
3. **Closing**:
   (h) exit;
   (i) assess usefulness;
   (j) check results with the team.

3.3 Implementation

3.3.1 Initiating

Based on the defined application process, (a) is well defined in the previous sections of this thesis. Following (b) and (c), we selected two existing tools to be used during the research that were not only simple to use, but were already known to the team, which drastically reduced the learning overhead. The tools were the following:

- **CircleCI**[11]: CircleCI is a solution that integrates with a *code versioning system* (e.g. Bitbucket) via webhooks and provides continuous integration and continuous deployment capabilities with fine grained control over building, testing and deploying your code.
- **Heroku**[12]: Heroku is a cloud platform that encapsulates most of the infrastructure related tasks for deploying an application. From managing an application server, to auto-scaling and making it accessible over the internet, it greatly reduces the effort required to this procedures.

With tools and solution clear, the application objective was as follows (d): deploy what we called a **staging** branch to his own environment to be accessible by a tester. So when a developer needed a new environment for a feature, the researcher would manually create and provide it using Heroku, and the developer would then push the built project to this new infrastructure. The new environment was also passed over to the tester, who would then validate it together with the developer. Once the feature was validated and merged into the **develop** branch, the researcher would delete the environment created previously.

3.3.2 Iterating

Although some manual actions were required, little overhead was applied and the next step was to improve and automate most of the solution steps. With our main flow running, next iterations were designed to remove the manual tasks of the developer and researcher.

In the **first iteration**, the objective was defined as: (e) automate the environment creation in Heroku. For that, a convention was defined (f) that every branch that would require it's own staging environment would be prefixed with **staging-**. A webhook was attached to the versioning system with the following purpose: every time a new branch with a **staging-** prefix was created, it would automatically make a request to Heroku create a new environment. Since the convention was simple enough, with no negative effect to the process (g), the developers accepted the idea with no objections.

The **second iteration** objective was defined as: (e) automate the build and deploy made by the developers. By using the convention created in the previous iteration, researcher and developers combined to (f) add CircleCI scripts that would automatically build and push every change made by the developer in a **staging** branch to it's proper Heroku environment. When scripts were added, developers had one manual task less than before (g), so the acceptance was also without objections.

A nice technical detail to be noted is that since names were a convention, the webhook previously added and the CircleCI's build and deploy scripts did not necessarily had to interact, which greatly reduced the implementation complexity. For example, if a new branch was created called ***staging-****feature-1*, the webhook would request an environment creation called **validate-feature-1** while the CircleCI scripts, following the convention, would try to deploy the code to an environment called **validate-feature-1**.

Similar to the second iteration, the **third iteration** had the objective to (e) automate the deletion of the environment once the feature was validated. A similar convention was defined that (f) once a staging branch was merged, it means that it was fully validated, and by this it could be merged into **develop**. By using the same webhook responsible for the environment creation, we attached an action to every time that a branch prefixed with **staging-** was merged into **develop**, it would automatically make a request to Heroku to delete the environment related to that branch name. Since the git-flow branching model was being used, this process was also already in the development process. Again, the acceptance was subtle, with little to no overhead.

After the three iterations, the process was complete and being applied fully by the team. The only manual part of this new flow was the notification that a new environment was created to be validated. Once the developer created and pushed a new **staging** branch and the triggers for build/deploy and environment creation were fired, he needed to manually pass the access URL to the tester responsible for that feature.

Since leaving this part manual was not critical to the process success, we opted to leave it for future works.

3.3.3 Closing

With the process fully applied and all the steps properly automated whenever possible, the researcher (h) stopped the iterations and exited the application. The process kept being applied by developers and testers, and since it required minimal manual effort, the team rapidly internalized this new proposed flow. One key point that must be noted here is that *the proposed flow is being used by more than **6** weeks without any kind of modification, adjustments, or researcher interference.*

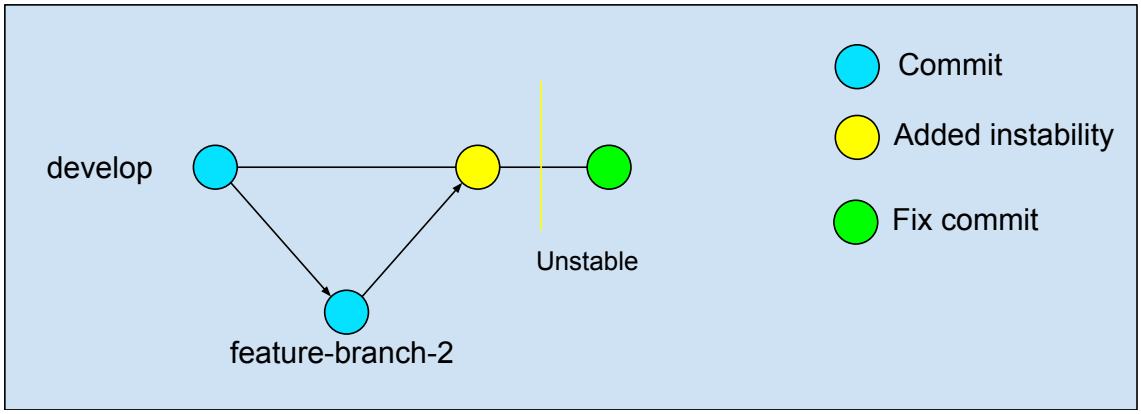

Figure 3.1: First method of detecting unstable commit

3.3.4 Quantitative analysis

Following steps (i) and (j) of the application process, we verified the applied process usefulness and asked ourselves if it really asserted the problem it was the designed to assert: reduce the **develop** branch instability.

For our specific scenario, we need to answer one question: how do we check if the develop branch was unstable at some point? Can we do that by looking on the git commit history? Can we assert it in a quantitative measure?

To answer that question we need to bring it closer to our scenario. We found two ways to verify that some instability occurred. Either (1) there is a commit (not a merge commit) directly into the develop branch addressing some fix on it's message (Figure 3.1), or (2) there are two merge commits of the same branch, which means that the fix was made on the same branch that added an unstable commit, and it was re-merged into the main branch (Figure 3.2). *Note* that a direct commit to the develop branch is not a good practice, but a common practice for small teams, such as ours, when something must be fixed as fast as possible.

Since our method gives us a way to isolate a branch, when applied to the given scenarios, (1) and (2), it outputs a commit history that can be pictured like Figure 3.3 and Figure 3.4 respectively. The majority of commits into the develop are unique merge commits, that is, branches once merged did not need to be further adjusted.

By running the `git log`[13], we can verify all the commit history of a branch, and now we will focus on the develop branch. By adding some parameters do the `git log` command, we analyzed the commit history between March 1st and May 31th, both dates in 2018. The results are the following:

- **159** total commits from all branches, excluding merge commits, only feature related;
- **38** commits made to the develop branch, including merge commits, from which:

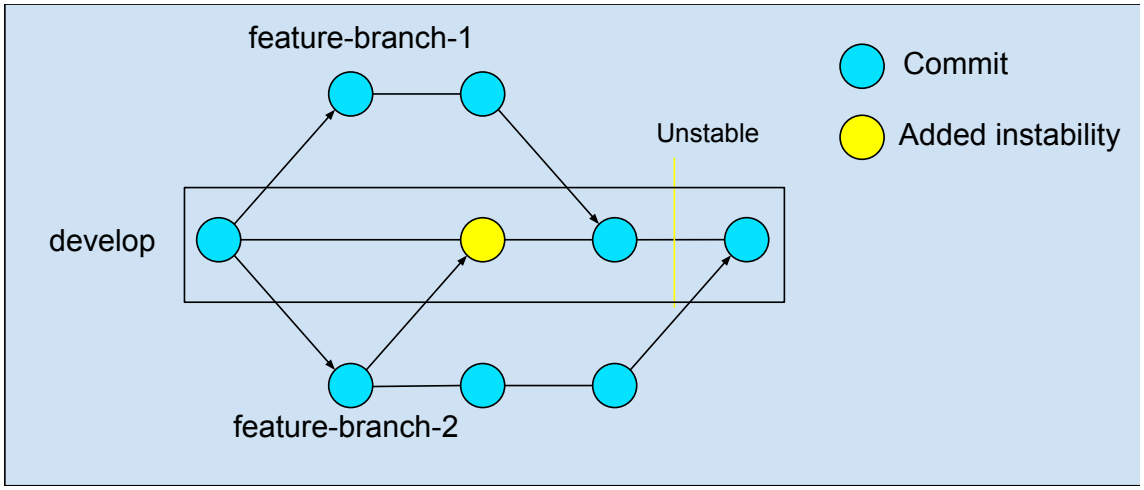

Figure 3.2: Second method of detecting unstable commit

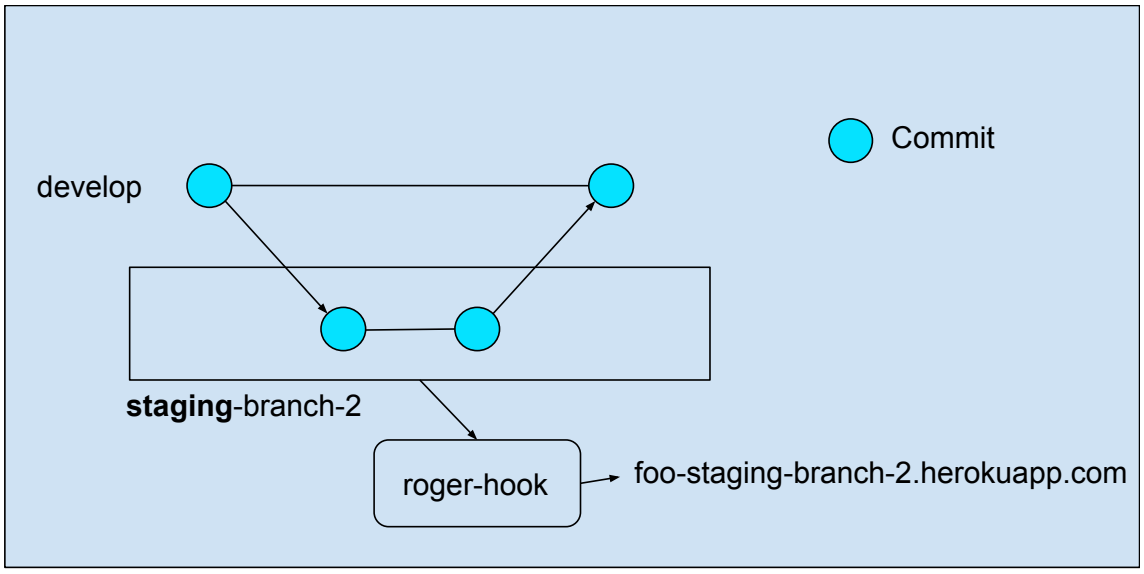

Figure 3.3: Resolution of unstable commit using Roger. Scenario pictured in Figure 3.1

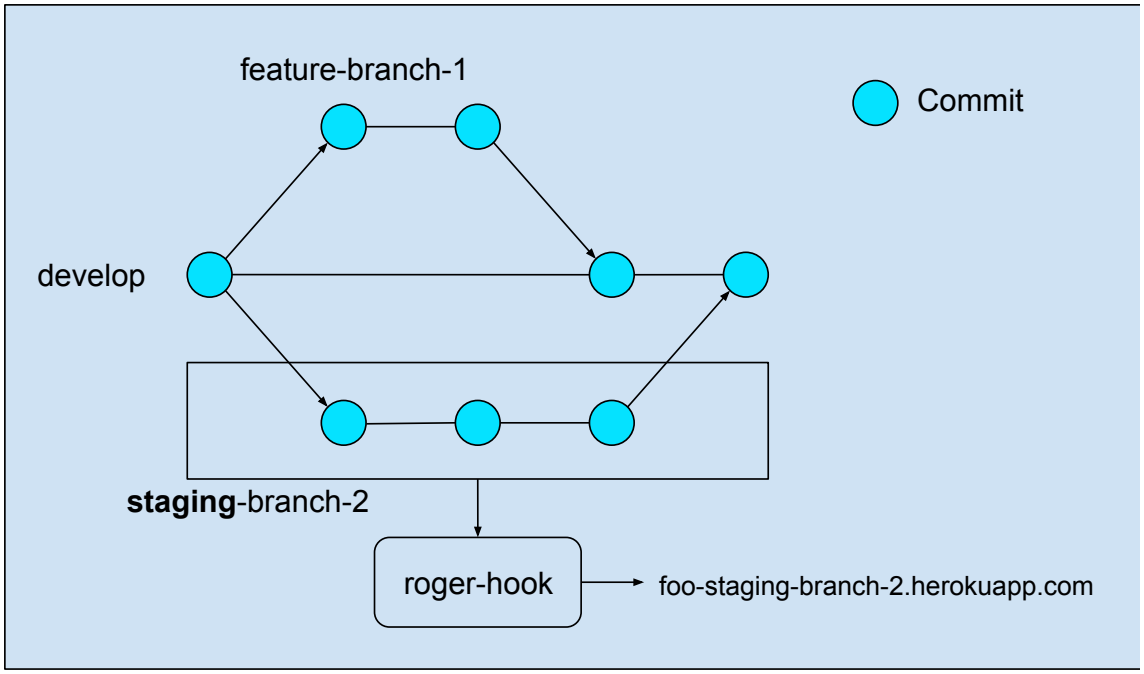

Figure 3.4: Resolution of unstable commit by using Roger. Scenario pictured in Figure 3.2

- **32** merge commits,
- **6** direct commits.

A closer analysis on the direct commits only, **6 total**, showed that **only 2 fix commits** were made to the develop branch. The other direct commits were not fix related, either infrastructure or code formatting related.

There was a total of **5** staging branches used to develop major features that required constant business validation.

Note: comparative results with commit history prior to these dates were not valuable. The development process of the team was not mature enough, the team was new to the project or the git-flow model was not used by that time.

3.3.5 Qualitative analysis

To qualitative analyze the results, we used parts of the focus group method defined in [14], since it's a fast and cost effective method to obtain experiences from practitioners and users. Although it's was designed to be applied in groups of 3 to 12 participants, we could extract some points to be used on one-on-one interviews. One of the focus group method's use cases is to obtain feedback on how models are presented or documented", which is just our case. So, following the steps defined by the method, we started our qualitative analysis.

Defining the research problem. The objective was to obtain feedback on application of the *roger-that* flow inside the company development cycle.

Selecting the participants. Since our application was restricted to one company, we selected the people that had contact with the proposed model. To obtain different points of view, we selected **1** from the business team and **1** from the developers team.

Planning and conducting the interview session. We held only one interview with each person, each made separated so participants did not know about the response from each other. Each session started with a brief review of the model to clarify the context and what was the purpose of the session: to elucidate if the model was useful. This initial review was also to ensure that the participant's opinion should represent the real world situation or feeling regarding the model presented, not concerning about it being good or bad, this was explicitly reinforced from time to time during the interview.

We kept the format of the interview the same for both teams to ensure the research construct was aligned with our research goals, and the points of view obtained were over the same objects of analysis. We tried to elaborate questions that were generic enough and asked them for both business and developer side. The questions were the following:

1. How was the feeling when using staging environments? Did you felt any change, for better or worse, to your development cycle?
2. Do you think it changed the interaction between developer/business?
3. Did this new flow changed regarding deadlines? Were they easily met?
4. There was any overhead or simplification added to your workflow?
5. **(business only)** What do you think about being able to approve a feature yourself and it be deployed to develop/production?
6. What do you think that could be improved regarding staging environments?

Analysis. The sessions were audio recorded so we could reduce instrumental errors. We interviewed Gustavo, the team's business person responsible for testing the staging features, and Marcus Vinícius, one of the developers that used staging environments to develop new features.

The overall team's felling was that this new model really improved the development cycle, and made the system mode robust and less prone to instabilities.

From Marcus, we could verify and assert the ideas raised before. We extracted from the interview that the technical scenarios pictured in the quantitative analysis were real and addressed by the proposed flow. For Gustavo, the isolated environment made it

"more clear which feature we were testing and the access to that feature was easier". This showed that different areas enjoyed different points of the solution.

When asked about any changes in the communication between teams, Marcus said that now they talk in terms of **staging** environments, when he finishes some feature, the tester can validate it in an specific environment and they can deploy it. From the other side, Gustavo's response was that he knew exactly which feature he should test and the dialog with the developer is open the moment the link to access a staging environment is sent, so it facilitates both communication ant testing. Here we can see that we have the different perspectives converging to a specific improvement on the communication and integration between teams, both know what is a staging environment and what is it's purpose.

The question number (3), about deadlines, briefly shows that the participants were not biased into evaluating the model as good. Both promptly responded that there was not change regarding deadlines being met, but raised some interesting points. Gustavo said that the new methodology helped because everything becomes more controlled, instead of having an environment where he would test multiple features, he has a link (URL) associated with a feature and I can set a specific deadline that, and the whole process becomes more coordinated. Similar for Marcus, he said that sometimes they have a deadline smaller than the end of the end of the cycle to show some feature for a partner or client, and it really helped.

For both Marcus and Gustavo, when asked about any overhead added to how they were used to work, the answer was clear and simple: there was **no** overhead added at all, which is a key point for the success of this model. Marcus pointed out that the only thing he has to do is create a new branch with a specific nomenclature that begins with **staging**, **just that**, for him it's not an overhead that will disturb the development process.

Trying to expand some future uses for this new model, I asked Gustavo about question 5, he asserted an interest in being able to deploy features by himself, since the test was more complete and closer to the end user. Here we can see this new flow opened a door between business and technology. Not in a way to make the technology team *controlled* by the business person, but work together and exchange experiences from both worlds on a new communication channel.

Finally, when asked about how the method could be improved, Gustavo said that maybe if they had a better methodology to control the links to staging environments would benefit their inner process. While Marcus could not point any improvements by the time of the interview.

An interesting point that we can see from the interviews is that although we initially aimed at the technical problem of a unstable and polluted develop branch, this was not

the final result at all. The main point that we hear from the interviews is about **communication**. Mainly for Gustavo, the business side, he clearly stated in many answers that the interaction with the developer was more active and he became closer to technology team.

3.4 Suggested tool specification

As the workflow was defined and applied, to make it even more easily adoptable to any company, we need to define and provide a clear way to apply the step (5) of the application process mentioned before. The "automate one step of the solution" part was done by a tool developed by the researcher, and in this case it consists of a simple web application that receives and makes HTTP requests. For now, we will name the tool **Roger**. Roger works as a bridge between the code versioning system (Bitbucket) and the environment provider (Heroku).

We created an application Express[15], a framework for web application development on top of NodeJs[16]. It attaches as a webhook[17] to Bitbucket and listens for events related to staging branches, when an event of interest is detected, we make use of the Heroku API[18] execute the proper actions related to staging environments. The two scenarios we are interested are:

- **When** a staging branch is created or updated in Bitbucket **Roger** requests an environment **creation**
- **When** a staging branch is merged in Bitbucket **Roger** requests an environment **deletion**

From Bitbucket's documentation on webhooks, we can register one webhook to each scenario defined. For the first scenario, when a commit is pushed, Bitbucket will make an HTTP request to our specified endpoint in Roger with a payload data we can use to verify if that commit is related to a staging branch. If positive that the commit corresponds to a staging branch creation, Roger then makes an HTTP request to the Heroku API to create a staging environment for that branch, following the naming convention defined by the team. For the second scenario, when a pull request is merged, Bitbucket will also make an HTTP request with different payload data we can verify that it is indeed a staging branch being merged. If so, Roger makes an HTTP request to Heroku requesting that environment deletion.

The code for this implementation is simple, and can be verified in figure 3.5. The main parts of it consists in defining two endpoints called "push" and "prClosed" for first and second scenarios, respectively. The variable "heroku" is a third-part library[19] used

```
// First scenario, after checking if it's related to a staging branch:
// when a commit is pushed
exports.push = (request, response) => {
  // Roger requests an environment creation
  heroku.post('/apps', {
    body: {
      // using the prefix "bx-" for convention
      name: `bx-${request.branchName}`,
    },
  }).then(() => {
    response.send({ status: 200, message: 'App created.' });
  })
};

// Second scenario, after checking if it's related to a staging branch:
// when a pull request is closed, or merged
exports.prClosed = (request, response) => {
  // Roger requests an environment deletion
  // using the prefix "bx-" for convention
  heroku.delete(`/apps/bx-${request.branchName}`).then(() => {
    response.send({ status: 200, message: 'App deleted.' });
  })
};
```

Figure 3.5: Code that implements the scenarios covered by Roger

to interact with the Heroku API mentioned earlier. To check if an event is related to a staging branch, we can simply match the branch name in the payload data against a regular expression and verify if it starts with "staging-", for example.

The final step is to configure the continuous deployment tool being used to actually deploy the code to the created environments when a staging branch is detected. Since it is highly variable from tool to tool, we cannot go into technical details on how it is done.

We can see that it's not some complex tool, and this tasks, while they can be manual, and were in the beginning of the research, carry the heavier part of this flow. If required to be performed by some team member each time a staging environment was needed, it would probably mean a failure of the process in the long run. We could see from the qualitative analysis that there was literally no overhead added to the team, and Roger is the reason for that.

Chapter 4

Conclusion

In this work, there was an attempt to improve the development cycle by applying a minor adjustment to an already consolidated model (git-flow) and using existing technologies (Heroku and CircleCI). Although the scenario is not a completely generic one, it is a common one in product development: to have a relatively lightweight part of the system, a front-end application for example, that can be isolated, deployed and tested separated from the main line of development. With minor adjustments, this new model can be applied to different companies, with different tools and development culture, but keeping the same end goal: improve the team's development as a whole.

When the research started, we were aiming at how we could increase the stability of the system, reduce the noise in the commit history, and make the development branches overall more stable. We focused on resolving technical issues, and although we reached this goal, it wasn't, by far, the end result of the research.

The focus group method brought us an exciting insight for future discussion: DevOps emerges mainly to act as a bridge between development and operation teams, but when we used a subset of it in our model, it clearly showed that the benefits of it sprang all the way to the business, or management, level.

Analyzing the final results of the research, we noted that this new model was not a simple technical improvement to the developer's job, it is an empower to business people since it creates a better visibility of the product's evolution. We opened different communication channels between areas of the company, and as a collateral result, we can have a business area much closer to the technology and teams more aligned towards the development.

From here, and now we are talking about closing a gap that once was distant, we approximate the business to the operations, and vice-versa. It does not mean that we aim to make one area replace tasks of the other, it means that at least one area can *comprehend* more parts of the company. The business can now can have a better notion

on what it means, for example, to “deploy a new application”, “merge a new feature”, “expand a server”, and plan accordingly. And since this door is open, the operations can actively participate on the business because the limitation created by the technical terms begins to dilute.

For future works, we can make the interaction between teams even more closer, for example, making it possible for the business teams to actively participate in feature deployment. By providing a dashboard and a centralized access to staging features, we can control merges and environments without having to teach how to interact with tools “specific”to developers.

References

[1] Reason, Peter and Hilary Bradbury: *Handbook of action research: Participative inquiry and practice*. Sage, 2001. 3

[2] Iversen, Jakob H, Lars Mathiassen, and Peter Axel Nielsen: *Managing risk in software process improvement: an action research approach*. Mis Quarterly, pages 395–433, 2004. 4

[3] Torvalds, Linus and Junio Hamano: *Git: Fast version control system*. URL http://git-scm. com/book, 2010. 5

[4] *cvs - concurrent versions system*. `https://www.nongnu.org/cvs/`. 5

[5] Collins-Sussman, Ben, Brian Fitzpatrick, and Michael Pilato: *Version control with subversion*. " O'Reilly Media, Inc.", 2004. 5

[6] Torvalds, Linus and Junio Hamano: *Git: Fast version control system*. 2010. `https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell#ch03-git-branching`. 6

[7] Driessen, V.: *A successful git branching model*, 2010. `http://nvie.com/posts/a-successful-git-branching-model`. 7

[8] Krusche, Stephan and Lukas Alperowitz: *Introduction of continuous delivery in multi-customer project courses*. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 335–343. ACM, 2014. 7, 9

[9] Humble, Jez and David Farley: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010. 7

[10] DeGrandis, Dominica: *Devops: A software revolution in the making?* 2011. 7

[11] *Circleci*. `http://circleci.com/`. 11

[12] *Heroku*. `https://www.heroku.com/`. 11

[13] *Git - git-log documentation*. 2010. `https://git-scm.com/docs/git-log`. 13

[14] Kontio, Jyrki, Johanna Bragge, and Laura Lehtola: *The focus group method as an empirical tool in software engineering*. In *Guide to advanced empirical software engineering*, pages 93–116. Springer, 2008. 15

[15] *Express*. https://expressjs.com/. 18

[16] *Nodejs*. https://nodejs.org/en/. 18

[17] *Atlassian bitbucket - manage webhooks*. https://confluence.atlassian.com/bitbucket/manage-webhooks-735643732.html. 18

[18] *Heroku platform api reference*. https://devcenter.heroku.com/articles/platform-api-reference. 18

[19] *Heroku client for node applications*. https://github.com/heroku/node-heroku-client. 18