Software development process improvements - *Case QPR Software Plc*

Lidia Zalevskaya

Master's Thesis

Degree Programme in

Information Systems Management 2019

Haaga-Helia
University of Applied Sciences

Abstract

Date: 2019.11.24

Author(s)
Lidia Zalevskaya

Degree programme
Information Systems Management, Master's Degree

Thesis title Software development process improvements - Case QPR Software Plc	**Number of pages and appendix pages 98 + 26**

Initially this study was planned as an effort to improve on a software development process within an existing team using an existing product code and systems. However, the situation changed and a new team (DevApps team) was established and given a new project, which created an opportunity to build a new type of team, product, process, and tools pipeline from scratch utilizing the improvement ideas. An Action Research framework was adopted as the theoretical approach for the study, while the Scrum methodology served as a framework for the development practices.

The study began by summarizing previously identified problems in the software development process at QPR Software Plc and formulating improvement ideas focused on the coding workflow and Scrum practices. These were then tested in practice by the new DevApps scrum team. The research analysis centres on the process of choosing and setting up the new team's development tools, figuring out ways of working, and implementing several iterations to find the best suitable development process.

The most valuable empirical outcomes were the creation of a branching strategy and Git workflow for the DevApps team, the team members' practical experience of working with Git and with the Azure DevOps developer services. A key outcome was the shift in many verification activities to earlier phases. Moreover, verification was enforced automatically by adopting the practice of changes being 'pulled' into the main development branch, rather than the previous practice of 'pushing' changes there. As well as understanding of the importance of integrating smaller code changes at a time, and learning to define and plan smaller work items.

Adopting the Azure DevOps service to manage the development flow enabled much more convenient inline code reviewing. Another important outcome was the understanding of need to improve Product Backlog Grooming, which meant adding and enforcing stricter criteria for user stories, such as Definition of Ready for Implementation, and Definition of Done.

The Agile Software Development is an iterative process, which is consistent with the report structure of Action Research. At the time of writing this report, the practical implementation has been ongoing for 3.5 months. The team were able to set up an initial process and tools, though further improvements are expected to follow in subsequent iterations. The preliminary observations have been shared with the rest of R&D at QPR Software, though it will take more time to crystalize the lessons learnt and to plan for other teams to utilize and apply those findings.

Keywords
Software development best practices, Scrum, software development workflow, git workflow, branching strategy, version control

Abbreviations

CD - continuous deployment

CI - continuous integration

DoD - definition of done

E2E - end to end (usually about UI automatic tests)

MVP - minimal viable product

QA - quality assurance

R&D - research and development, in QPR Software called also Products & Technologies.

SDLC - software development life cycle

SDT - software development task (work item in issue tracking system)

SW - software

TA - test automation

UI - user interface

VCS - Version control system

WIP - work in progress

Table of contents

Table of Figures

Foreword

I started conceiving and planning my thesis a little over two years ago when I arrived at QPR Software. I want to say a few words of gratitude to my colleagues who have inspired me continuously and given me plenty of food for thought. I want to thank all my team-mates, especially Guðni Ólafsson and Pekka Kyllönen for their support and ideas. I'm also deeply thankful to Heikki Sjöblom for his supervision, thorough reviews, and advice.

I would also like to thank my tutors and supervisors at Haaga-Helia, Kasper Valtakari and Jouni Soitinaho, who have so ably supported and encouraged me.

I also couldn't complete this work without a jillion discussions with my husband Den - thank you!

And I want to give my warm gratitude to Mark Phillips for polishing my English.

1 Introduction and background

1.1 The company and the product

QPR Software Oyj is a Finnish software company that provides management software products (QPR Software, 2019). Its main products are solutions for strategy execution, process mining, enterprise architecture, and performance and process management. QPR UI and QPR UI 2.0 are web applications for visualizing business analytics data to help customers gain insights to inform their decision-making and improve their competitive advantage (QPR Software Plc, About QPR - Overview | QPR, 2019).

QPR UI 2.0 is an application serving the same purposes as QPR UI, but is being built from scratch within the scope of a technology renewal project. The project is implementing an application framework that is new to this UI product and is targeted at a more efficient delivery of custom QPR UI applications.

1.2 Personal context

I work inside a new software development team (created in the summer of 2019 and adopting the name DevApps) at QPR Software (https://www.qpr.com/) as a QA lead and Scrum master. My role is to facilitate software development, to support programmers, and to control product quality. I also work with the code base, so I’m fully involved on a daily basis in the development process.

For me personally, this study was a great opportunity to learn more about contemporary software development best practices and to try to apply those to a specific company’s situation and to generate significant know-how. In undertaking this project, the aim was to build experience and knowledge that could be applied in subsequent projects to improve or initiate from scratch a software development process. The theoretical learning and practical know-how gained can be used as an aid in similar future projects aimed at software development improvements.

1.3 Aims and objectives

As a background to this study, information from previously conducted surveys in the company, from team sprint retrospective meetings (in my previous team), and from individual discussions with team members had highlighted difficulties and obstacles in the software development workflow, leading to frustrations and disappointment among engineers, with some practices clearly showing room for improvement. Hence, the aim of the study was:

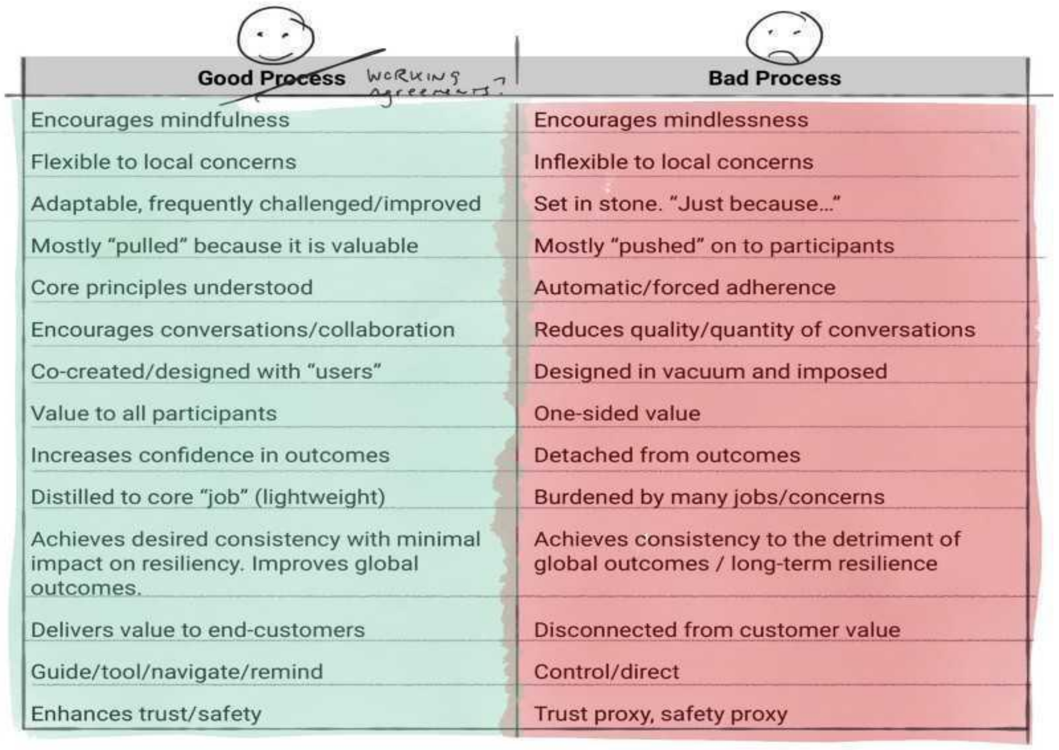
Firstly, to gather all available information about previously known issues with the software development process at QPR Software. Information about known issues was gathered from R&D teams in the company and was complemented with the author’s extensive experience working in one of QPR’s development teams prior to working in DevApps. During the period August–September 2019, an external consultant performed a “QPR 360 assessment” and the final report made numerous suggestions for improving the productivity around the business value creation process within QPR’s product development. The consultant’s report was used to cross-check with the issues and improvement ideas already highlighted in previous data gathering and to add more weight to the process change that was being planned to address those issues.

And **secondly** to define priorities - what processes and action plans need to be changed or improved. The aim was to iteratively implement the highlighted priorities in a newly established development team. Experience from working with the previous team was brought to bear in setting up a new pilot team, which was considered experimental and if successful would to serve as a positive example to other teams to change and improve their software development process. The new team were given more freedom to experiment with different tools and processes, thus paving the way for greater changes.

The study included an assessment of the current tools in use - how well do they serve or support the desired development process. Current tools included: a task and issue tracking system (Axosoft), tools and systems serving the development processes (dealing with coding, code review, version control etc.) and development documentation (e.g. working instructions or product technical documentation).

Alternative tools were assessed and compared with those currently in use within QPR Software so as to identify the best way to support the desired new practices.

The experimental team set up a project to develop a new product, for which we defined principles for the development process and followed the “best-practice” processes identified, with a view to paving the way for their application within the company’s entire R&D. The aim was to minimise the reliance on self-discipline in following the best practices and the desired process, so we set up automatic constraints to ensure everyone followed the agreed and desired workflow. We also defined criteria for critical events, defined core guidance principles, and minimised as much as possible the reliance on self-discipline in following what the process dictates. As an example of how this work is being extended, the new team is setting up quality gates for selected best practices and coding standards, by setting up Azure DevOps and code linting for Visual Studio Code. These ideas were inspired by a blog by John Cutler (Cutler, 2018) and Figure 1 from that blog summarises the good and bad traits of the process.



Good ~~Process~~ WORKING agreements?	Bad Process
Encourages mindfulness	Encourages mindlessness
Flexible to local concerns	Inflexible to local concerns
Adaptable, frequently challenged/improved	Set in stone. “Just because...”
Mostly “pulled” because it is valuable	Mostly “pushed” on to participants
Core principles understood	Automatic/forced adherence
Encourages conversations/collaboration	Reduces quality/quantity of conversations
Co-created/designed with “users”	Designed in vacuum and imposed
Value to all participants	One-sided value
Increases confidence in outcomes	Detached from outcomes
Distilled to core “job” (lightweight)	Burdened by many jobs/concerns
Achieves desired consistency with minimal impact on resiliency. Improves global outcomes.	Achieves consistency to the detriment of global outcomes / long-term resilience
Delivers value to end-customers	Disconnected from customer value
Guide/tool/navigate/remind	Control/direct
Enhances trust/safety	Trust proxy, safety proxy

Figure 1. Good Process vs Bad Process (Cutler, 2018)

During the various iterations of the process change, best practices from the software development industry were studied to elicit suggestions on what could be improved and how. I decided to write the report in a Zipper data structure format to reflect how the study was going, while seeking theoretical insights and trying ideas in practice on the fly. The aim was to test the improvement ideas through applications in practice within a new experimental development team working on a QPR UI renewal project. To this end, I set up metrics for the experimental team’s software development process. After having completed the first practical changes and assessing the development process improvements, future steps will be to plan for sustainability in the changes implemented by this team and, where it seems useful, to start encouraging other teams to adopt the same tools and process modifications (beyond the scope of this study).

Below is given an overlay matrix to help match the research questions to the chapters in this thesis, matching the theory to outcomes.

Table 1. Overlay matrix

Research questions	Theory	Results
What are the key changes needed in the development process to gain more value from the use of the Scrum framework?	"Initial Scrum practices" "Organize issue tracking and work management (backlog, epics, stories, tasks)"	"Scrum: What to change and how?" "Practical plan: Initial Scrum process setup" "Daily cafe, improving daily stand-ups"
What key initial changes need to be made to the development workflow?	"Theory: Main principles leading the process change effort" "Theory: Version control system and development workflow"	"Branching: Why, What and How to change?" "Coding Workflow: What to change and how?"
Assessment of current tools in use		"Choosing a version control system and provider, CI/CD system (tools pipeline)." "Organize issue tracking and work management (backlog, epics, stories, tasks)"
What are the most popular coding workflows used in the software development industry?	"Theory: Available Git workflow options"	

What would be the optimal and simplest coding workflow for this team and this product?	“Agree on coding workflow and branching strategy”	“Practical outcome: Initial suggestion for git flow in the DevApps team” “Step-by-step Gitflow How-to”
Which metrics would be most useful at the initial stage when the new team is setting up tools and processes and working on a new product prototype?	“Theory: Goal - Question - Metric framework”	“Practical application: DevApps team metrics - an initial proposal”
How to minimise the reliance on self-discipline in following the best practices and desired process? Or how to make our tools developer-friendly, so the process is followed naturally and cannot be violated?	“Main principles leading the process change effort”	“Discussion: How to achieve a better process that is less reliant on self-discipline?”

1.3.1 Initial plan - main tasks (08.2019)

A very rough preliminary idea about initial tasks when starting a new team and new product development:

Kickstart: Create Scrum Team Cultural Manifesto and Team Contract (working agreement).

Plan Scrum practices (start with following the Scrum guide and iterate to adjust & improve).

Organize a hierarchy of work items and issue tracking: backlog, epics, stories, tasks.

Choose a version-control system and provider.

Choose CI/CD and building system (tools pipeline).
Agree on coding workflow and branching strategy.

Define metrics for development team and for business objectives.

Beyond the scope of this study: Integrate code version control with CI&CD and automated testing.
Partially beyond the scope of this study: Define and document best practices, guidelines, and standards for application engineering.

1.4 Challenges

The first challenge faced was ensuring that developers and others would buy into the process change and the change process itself: to some people, the need for change is obvious, while others may be more comfortable with familiar ways of working and existing tools.

Too many changes happening too quickly brings with it a risk of people becoming change weary, from having to deal with uncertainties and the greater need to assess every little step every day when making many decisions.

Many simultaneous operational changes divert resources away from product development, while business pressures may still be high, creating a risk of not meeting the change objectives fully. Moreover, it may be difficult to evaluate the success of an individual change when so many elements change: Scrum adjustments, new team building, new processes and tools, as well as systems change.

Extensive competence development is also likely required, as well as continuous coaching for team members in how to change work habits. Training is necessary for some people to learn about the Git workflow and good Git practices. Internal resources can be utilised for this if there are volunteers who know Git well enough and have appropriate practical experience. Longer-term support is needed to ensure the desired work habits really do stick and the change is sustainable.

Changing the development workflow leads to changes in several tools at once (version control, build and TA systems), hence some time is needed to familiarise with these tools and find effective ways of working. Migration to a new tool also implies a need to assess the future of tools, to consider future expectations: which tools will be developed and supported in the future, and support for which products is likely to be terminated or minimized? For other teams who would move their existing product code and development processes to new systems, the costs of data migration need to be considered.

A geographical split between the teams in Helsinki and Oulu can lead to a risk of disconnect, with the possibility for tighter collaboration within sites and weaker interaction and information-sharing between sites - the so-called space barrier. Virtual teams face challenges involving trust, effective communication, cohesiveness - as well as the bond between team members (Ale Ebrahim et al., 2009). Cascio (Cascio, 2000) declared that there are five main disadvantages to a virtual team: a lack of physical interaction, the loss of face-to-face synergies, a lack of trust, a greater concern with predictability and reliability, and a lack of social interaction.

On a personal level, my main difficulty has been keeping the study within a reasonably limited scope, so that the study would not become too broad.

1.5 Structure of the study

Previously collected feedback was analysed, workshops were organized to identify further problems, improvement suggestions were collected, and further actions planned, with selected improvements then utilized by the new experimental team.

The text is organized according to a zipper structure, meaning the theoretical framework and the empirical part are intertwined, reflecting the action research paradigm used in the study. Over the course of this study, the new team went through several sprint iterations, represented in the “iteration” chapters (1–3). The team worked through 3-week sprint iterations, matched approximately to a calendar month for simplification. At the end of every sprint, the team held a sprint retrospective meeting, where team members reflected together and planned improvements.

The first iteration started with the initial research data collected over the course of 2018 and 2019: which included the results of questionnaires implemented in the R&D department (some organised through an external consultant, some carried out internally by the department during the previous two years), and the outcomes of workshops organized during an internal R&D conference at QPR called Developers Days. The data also included the author’s observations gathered from daily work in one of the development teams. A light analysis of the existing development processes and internal documentation was done to help compare the current state with the desired state. More focused measurements were made during the autumn of 2019 to set benchmarks for the new development team and to support the team in seeking appropriate and optimal best practices and processes.

2 Iteration 1: Methodological framework, literature review, and initial plan

This chapter provides background for setting up the new development team and the new product. It covers the methodology of the study, the principles guiding the change in processes, the theoretical background, and a preliminary list of the most critical changes to be implemented.

2.1 Methodological framework and strategy

Agile software development happens by means of iterations, where actions are planned, implemented, evaluated and analysed and lessons learned along the way are then used in the next iteration. Given that the aims of the study were to seek out optimal best practices and to identify a process change for developing a new software product, action research was a suitable choice as a methodological framework.

2.1.1 Theory: action research

The purpose of action research is to solve a particular problem and to produce guidelines for best practices (Action research, 2019). It was chosen in this organizational development context because it also serves as a framework for iterative improvements (spiral of steps: planning, action and fact-finding about the results of actions).

Action research implies that all stakeholders actively participate, which should support adoption and reinforce usage of new tools and practices. The action research methodology (see Figure 2) in my opinion also aligns very well with the Scrum improvement cycles: “planning action” matches with sprint planning, “taking action” corresponds with Scrum’s sprint phase, and “evaluating action” corresponds well with the sprint retrospective.

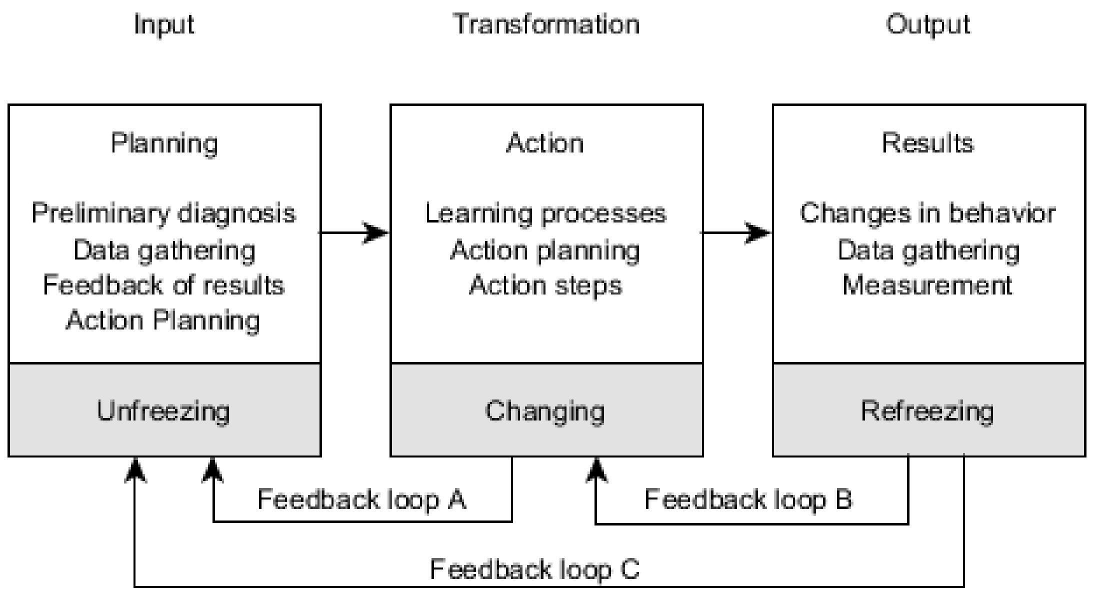


Figure 2. Action Research (Action Research, 2019. Wikipedia) CC BY-SA 4.0

A change is most successful when done with everyone involved: Kurt Lewin believed that the motivation to change is strongly related to action: If people are active in decisions that affect them, they are more likely to adopt new ways. "Rational social management proceeds in a spiral of steps, each of which is composed of a circle of planning, action and fact-finding about the result of action" (Lewin, 1958). This connects well with a key aim of this study, which was to affect habits and culture so that they would drive development processes from the bottom up, such that the process would be frequently challenged and improved by the team, rather than being imposed from the top down.

“Action research is about taking action, researching the action, and learning from the process” (Jean McNiff, 2016). McNiff sees action research as evaluating your practice to check whether it is as good as you would like it to be, identifying any areas that you feel need improving, and finding ways to improve them. Action research can be summarised in 3 steps: defining what we want to improve, doing it, and describing (and making it public) what has been done, how, and why. You discover existing knowledge and create new knowledge that people may not have been aware of before. The application of existing knowledge to new practices—while at the same time establishing a process and tool pipeline for a new team and new development project—is itself a research process and it generates procedural subjective knowledge: that is, know-how for a specific team, project, or product in a specific company. Learning a skill is also a form of subjective knowledge acquisition. Claiming that you can explain what you are doing in your practice implies that you have generated a theory of practice (Jean McNiff, 2016). As Horton and Freire describe, “[s]ometimes you create the road by walking it” (Horton and Freire, 1990).

2.1.2 Methods used in the study

Many development teams who are trying to follow the Scrum framework cannot take full advantage of Scrum (What is ScrumBut?, n.d.). In keeping with the iterative nature of the research and the goal of seeking continuous improvement, Scrum events were chosen as a key unit of analysis for understanding the effects of process changes (includes planning and retrospective stages, and daily status checks, i.e. inspect and adjust).

Workshops and reviews were also organized. Questionnaires were used to further clarify specific elements that arose during the research and to obtain feedback from the team.

2.2 Leading Change

This section describes the main ideas and principles that inspired and guided this work.

2.2.1 Theory: Leading Change. John Kotter (1995)

Typically, each change phase requires a considerable length of time to fully implement, but it is a natural human tendency to move too quickly, impatiently, and to want to achieve rapid results. The change process goes through a series of phases that would typically require a considerable length of time to be fully realised. Skipping steps creates the illusion of speed, but not a satisfying result (Kotter, 1995).

John Kotter’s eight steps for a successful change were used to inform this study:

1. Establish a sense of urgency

2. Form a powerful guiding coalition

3. Create a Vision (+ Strategy)

4. Communicate the Vision (use every possibility)

5. Empower others to act on the Vision (remove obstacles, make changes to systems, encourage risk-taking, consider non-traditional ideas and actions)

6. Plan for and create short-term wins (recognize and reward)

7. Consolidate improvements and produce still more change

8. Institutionalize new approaches.

2.2.2 Theory: “Island of freedom”

The QPR UI renewal project constituted an experiment carried out inside an established company with a long history and with long-lived products. The newly created team (DevApps) were given freedom to experiment with their work practices and development processes and tools, such that it closely resembled a start-up inside a corporation (see Ries’s work on lean start-ups) (Ries, 2011).

Ries highlights the importance of the minimal viable product (MVP) and how to develop it using fast feedback cycles of “Build - Measure (customer response) - Learn” (Ries, 2011). In applying this principle to developing our new QPR UI 2.0 prototype, we decided to seek customer feedback as soon as we had something to show, to experiment as much as possible, and to use validated learning. The suggestion was to start from a simple basic solution and to automate it when it is clear what creates value.

Ries emphasised the importance of learning what attributes customers care about. Customers do not care how much time it takes to build something, they care only if it serves their needs. To this end, it is good to set quantitative targets for recording progress and to utilize qualitative research to find ways to improve. Questions need to be formulated about the product and empirical tests carried out to find answers to those questions; with a "just do it" approach, a product is shipped and you wait to see what happens. Behind this approach is the idea that if you cannot fail, you cannot learn! In other words, encourage trials and failures. Customers often do not know what they want, so we should not take what customers think they want for granted, but rather test and find out what they really need. Value is providing benefit to the customer, and anything else is a waste, so when in doubt - simplify. Avoid overbuild and overpromise.

2.2.3 Theory: Main principles leading the process change effort

A good process is said to be adaptable, frequently challenged and improved, flexible to local concerns, and guided by core principles. It encourages conversations and collaboration, is co-created by the team, increases confidence in outcomes, achieves the desired consistency with minimal impact on resiliency (Cutler, 2018). With this co-creation in mind, the new team explored automating the workflow to follow good practices and to pave the way for good work habits. We also explored best practices to identify the good habits that we wanted to support. As Scrum master, I wanted to minimise the reliance on self-discipline in following the new best practices and the desired process. The aim therefore was to change the process such that it is not the process that dictates what and how things should be done, but the toolchain that leads the developer as much as possible, making it easy to follow the process.

Within the DevApps team, the process was defined and written down, yet there may still be the possibility for deviations (“Unauthorized process mutation”) in daily work practices. An improvement would be to make deviations impossible (for example, reviewing before merging into the main development branch) or to make following the process the easiest way to go, so people naturally do things the right way, without trying to find workarounds. We therefore sought to "automate" the best practices as far as possible by setting up a tool pipeline to lead our workflow (guide, navigate, remind), and to set up gates that would not allow us to move to the next step until conditions were met. The idea that people often choose the easiest or more convenient path when not restricted by a “fence” is illustrated by Figure 3 (below).

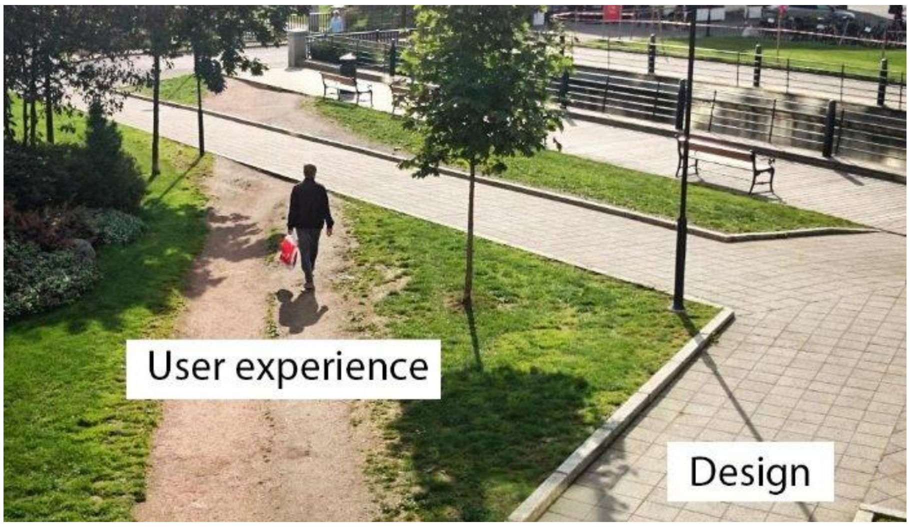


Figure 3. Intended path vs path people actually take (Chuwa (Francis) flickr.com) (CC BY-SA 2.0)

It is useful to consider how developers deviate from the most-desired process and why, and then try to address the causes of those deviations, but not by relying entirely on self-discipline. It is better to create a cultural shift and encourage the team to acquire new good habits, and to let go of bad habits. The idea that “culture eats strategy for breakfast” can also be applied to software development’s daily work practices, since it is the work culture that drives behaviour (Teasdale, 2002,195-196.)

Another critical and potentially beneficial change identified within QPR software was to shift quality assurance (QA) efforts leftwards in the process chain (to earlier stages) to make the main development branch stable and mostly functional (unlike previously). We identified another major goal to be ensuring automated tests and environments were stable, so as to provide developers with valuable feedback every day, although this improvement remains outside of the scope of this study.

Having frequent iterations that customers can use and comment on by means of an active feedback-loop helps team members to build an understanding of the customer.

2.2.4 Current process, workflow and branching strategy

In existing R&D teams at QPR Software, we used one "main development branch" in the workflow named “trunk” – a branch that serves as the base for all current development work. It included all the latest code changes that had been prepared for the next release as well as some unfinished features still under construction and earmarked for future release. All teams tracked the code changes via a central SVN repository. By default, new functionality was implemented through local copies of “trunk” and an SVN commit was used to publish a change to the common mainline. Running any kind of verification before publishing was entirely the developer’s responsibility and decision, so it was common enough that no tests were run or the test build was not run locally before new code was published. The main testing and quality assurance activities were carried out after the code was published to the “trunk”, so all necessary fixes were added to the already published, and possibly reused, code.

Anyone could commit changes to the trunk branch without barriers, and the release-branch policy was that the QA-lead either gave permission to merge a feature for release or the QA-lead would do that themselves.

To promote code to the release branch, all revisions needed to be cherry-picked from the trunk branch and added to the release branch, which created a second (re)integration stage (additional to integration in trunk). This second stage happened significantly later in the process, which led to merge conflicts (large deviations between code in the trunk and the release branch) and unforeseen dependencies would quite often be revealed at this point, which is arguably too late in the process. To illustrate, the trunk would have, for example, features A, B and C, where A and C are ready for release, while work on feature B is still in progress. In this case, the trunk and release branches would have different integration results: in the trunk branch are A, B and C, and in the release branch, there are A and C only. Figure 4 illustrates how the mentioned features would be promoted to the release branch.

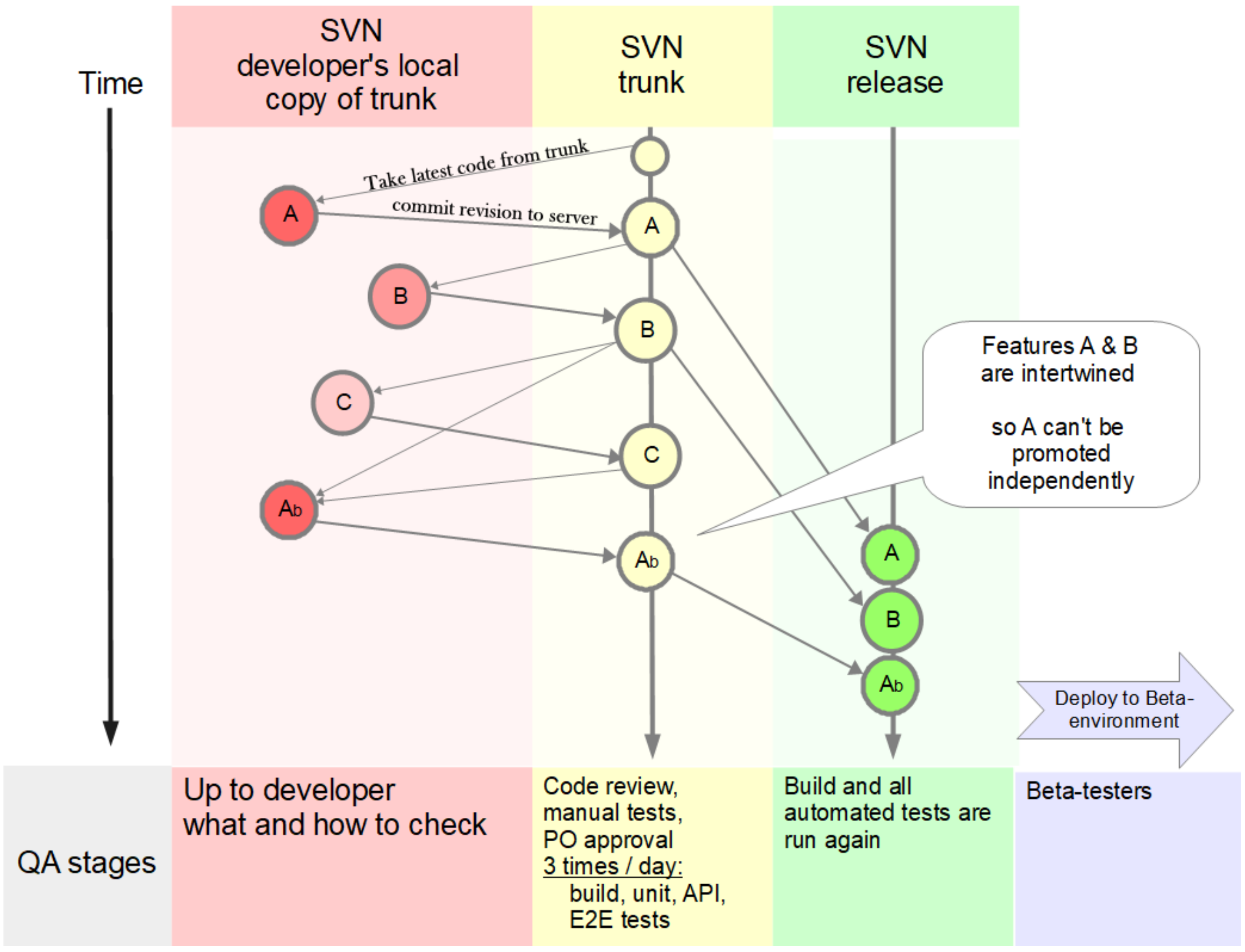

Time
SVN developer's local copy of trunk
SVN trunk
SVN release
Take latest code from trunk
commit revision to server
A
B
C
Ab
Features A & B are intertwined so A can't be promoted independently
Deploy to Beta-environment
QA stages
Up to developer what and how to check
Code review, manual tests, PO approval 3 times / day: build, unit, API, E2E tests
Build and all automated tests are run again
Beta-testers

Figure 4. Previously used SVN branches and workflow

Since features may have been under development for some time (weeks or months), it was not rare to find in QPR's existing process that a long-lived feature like A would become dependent on another big feature B. As a result, they could not be merged with the release branch independently: Either all revisions for both features had to be merged into the release branch together or only in a specific order.

In cases where a branch on the SVN server was created as part of developing some bigger or experimental feature in that branch, we would still have to pick revisions one by one to merge into the trunk branch, and then we would need to do the same again to merge them into the release branch. During development, the feature branch receives updates from the trunk and conflicts are resolved. However, it is a new task again to resolve issues when merging with the release branch.

This is quite similar to how things work in a Centralized Workflow, which according to Atlassian, is great for teams transitioning from SVN, but it does not leverage the distributed nature of, for example, Git (Atlassian, Git Workflow, n.d.). Atlassian was not suitable for us because it did not change the process and so the same problems and conflict resolutions would continue to be a bottleneck as the team gets bigger. The current approach where multiple teams are working with the same code leads to merge conflicts and a bottleneck is caused by the heavy workload during periods when a lot of merges from the trunk to the release branch take place.

2.2.5 Empirical: What needs to change and why

All components in the software development process are extremely intertwined, so it is very difficult to draw lines between different areas of the whole process - between tools and systems. Many changes affect several systems or aspects, which is why it was decided that the current study should cover not just the branching strategy and coding workflow, but also Scrum practices, product (features) planning, and work tracking processes. The following sections present a selection of the most important problems to solve, excluding some that fall outside the scope of this study, such as test automation stability. Study data were collected over the course of 2018 and 2019 in the form of questionnaire data gathered from personnel in the R&D department, as well as data from workshops organized during Developers Days (an internal R&D conference at QPR). The data were further augmented by the author’s observations from previous roles with the company’s existing development teams.

2.2.5.1 Branching: Why, What and How to change?

This section deals with the arguments and justifications for suggesting a change in the branching strategy (model).

When multiple developers from several teams work on the same code base and are not actively working on integrating their changes with other developers’ changes, complicated merge conflicts are inevitable, which leads to delays in code being promoted from the development stage to release. This also makes it hard to identify bugs due to concurrent refactoring. It also matters how often integration happens - it is poor practice to make changes in large chunks and for them to accumulate in a local branch over a long period. Figure 5 illustrates a merge conflict involving significant overlaps in code changes.

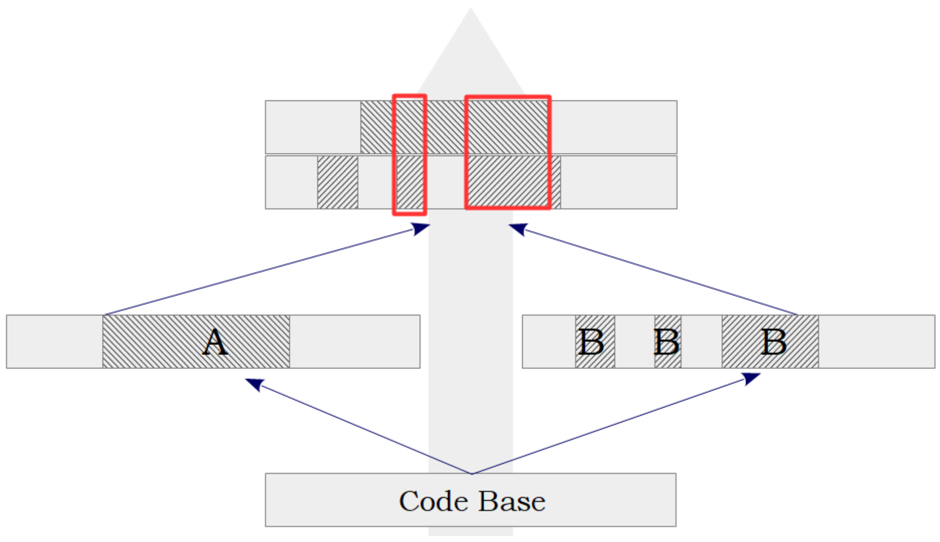


Figure 5. The larger the chunks are, the harder it is to integrate them

It is clearly much easier to integrate smaller changes, since there is less chance of large overlaps, and where there is an overlap, the common area remains small and manageable (see Figure 6).

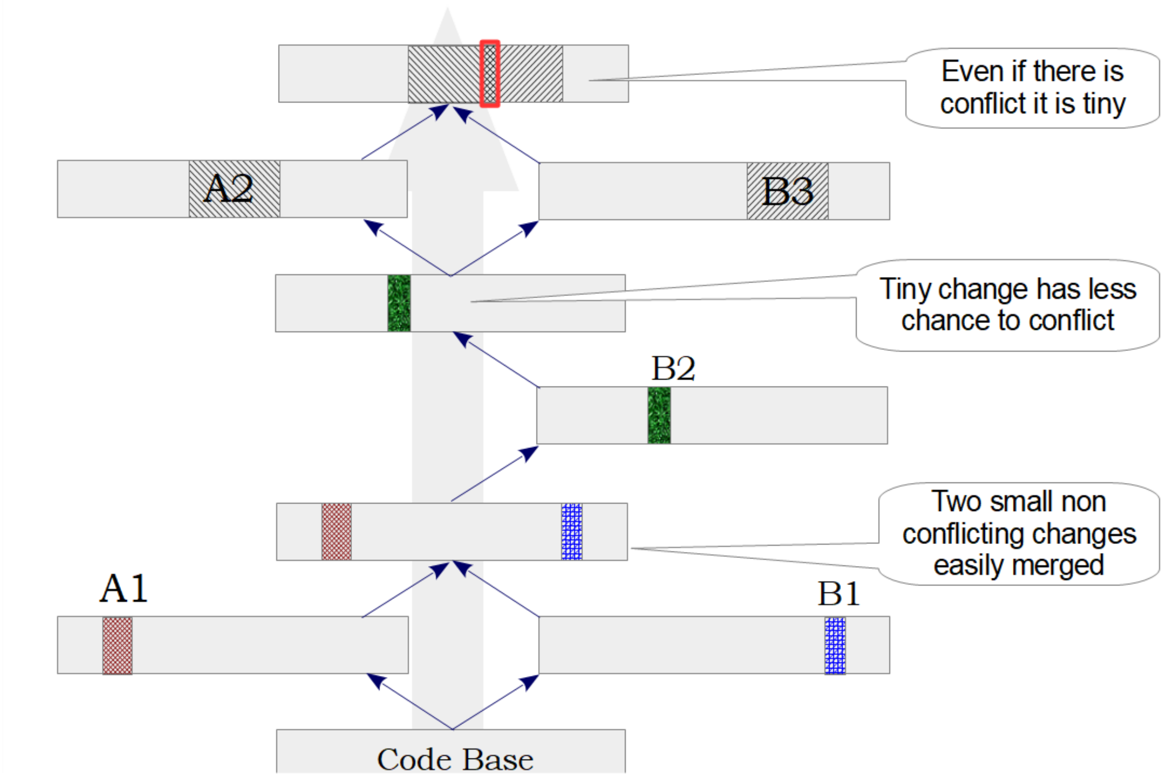

Even if there is conflict it is tiny
A2
B3
Tiny change has less chance to conflict
B2
Two small non conflicting changes easily merged
A1
B1
Code Base

Figure 6. Smaller changes are easier to integrate

During the pre-release merge window, a lot of changes are waiting to be merged into the release branch, creating a bottleneck in the code promotion process. A solution would be a strategic shift from larger releases towards continuous delivery (CD), with smaller value increments happening more often. Since the release branch would be created as a copy of the current main development branch, it would not require any new integration, since all integration happens when the pull requests to the main development branch are accepted.

In QPR Software, we have struggled with non-informative results from automated tests (TA) run on the main development branch (trunk) that rarely passed and often led to many failures, often with different causes. One negative effect of the constant failures was that developers were not motivated to monitor the TA status, which created a vicious cycle. Another consequence was that one faulty change used to break the environment that other developers were also using. An obvious improvement is to have a review process and tests carried out before the change is accepted into the main branch, which would help maintain TA stability. An automatically enforced requirement to run tests on any small change offered for integration into the main development branch would force us to maintain the TA every day, thus improving TA stability.

Previously, having only one main branch with an automatically enabled build and test execution for all developers has led to bottlenecks, where developers have to wait for queued changes to be tested in the trunk before seeing the results of their change. One might think that delays would be intermittent, but in practice, changes used to come without interruption and so it made seeing the effect of one’s own changes nearly impossible. A lot of time was wasted in attempts to identify and fix faults, for example: changes A, B and C may contribute to issues in the trunk, and it was hard to identify causes. Compiling non-finished features (works in progress) together in one common development branch (the trunk) also used to create unwanted dependencies. A fix for feature A could easily become dependent on some code change in feature B. This was good for early integration testing and for sharing useful common components, but it created merge conflicts and prevented the independent promotion of individual features. Figure 7 shows the trunk, where the oldest revision is at the bottom and the newest is at the top. By following the history of changes from the bottom up, it is possible to see that A causes both a build break and TA failure. Repairing the broken build took one whole day, but investigating the TA failure was further complicated by changes made previously to the environment and changes B and C, neither of which were tested separately, so it took a week to identify the cause of the TA failure and to fix it.

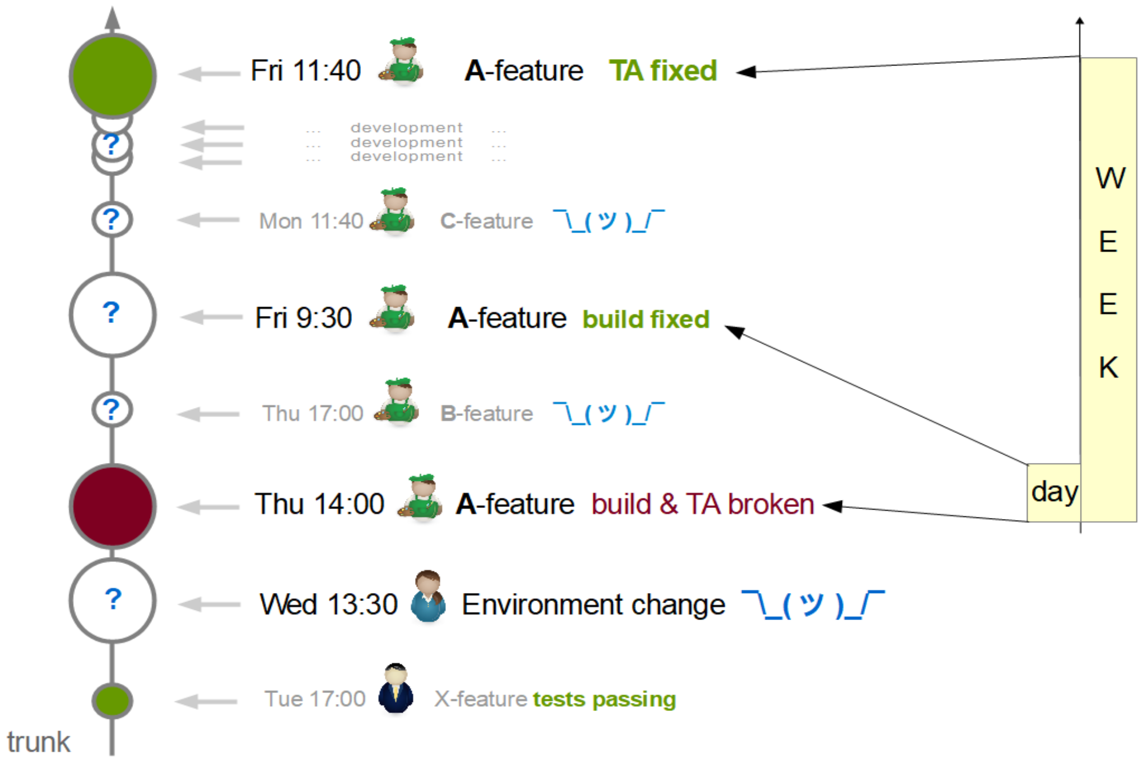


Figure 7. A case where faults introduced by feature A were masked by other changes

A majority of these problems can be solved by switching from SVN to Git and by implementing a simple and effective branching strategy. The major difference for code promotions to the release branch would be that instead of cherry-picking features from the trunk for promotion to the release branch, we will simply branch off every new release branch from the HEAD of "develop" in the Git workflow (equivalent of the trunk in SVN), so if there are features under development and not ready to release, they would have to be hidden using specific techniques. "Since many organizations new to Git have no conventions for how to work with it, their repositories can quickly become messy. The biggest problem is that many long-running branches emerge that all contain part of the changes. People have a hard time figuring out which branch has the latest code, or which branch to deploy to production." (Introduction to GitLab Flow | GitLab, n.d.)

2.2.5.2 Coding Workflow: What to change and how?

To **avoid merge conflicts** and to **provide new code for everyone as soon as possible** in DevApps, we will aim to develop in small increments with frequent integration. In the ideal situation, each increment consists of tiny independent self-contained changes accompanied by tests that establish there is no apparent regression and that the introduced functionality is working as stated in the description of the change.

Updating the work-in-progress (WIP) feature branch with the latest changes from "the main development branch" (to be named ‘develop’) also often helps to avoid merge conflicts. In addition, we would need to build and run TA for every feature branch. It makes sense also to encourage **early** Pull Requests with the aim of getting preliminary **feedback**.

To have a stable working code base for everyone to build new functionality on, we need to avoid broken code getting into the main development branch (typically called ‘master’ in Git). We need to eliminate cherry-picking single commits or revisions from the development branch to add to the release branch, as was done with SVN. Rather, a feature needs to be rebased on top of the main development branch. The responsibility for pulling the change into the main development branch is passed from the author of the changes to a group of authorities that consists of the team members reviewing and (manually) testing the change, the CI/CD pipeline doing the static code analysis, the automated testing, and the style and consistency checks. A direct push to the main development branch (as was possible with the trunk) can be blocked by setting up an automatic branch policy. As a result of using Pull Requests, base commits & fixes are promoted as one logical bundle, which hopefully should lead to a neat commit history and make promotion (merging) easier.

I would expect and recommend our pull request practice to emphasize **early testing efforts** and involve developers more in QA and to make them feel responsible (and proud) for the quality created.

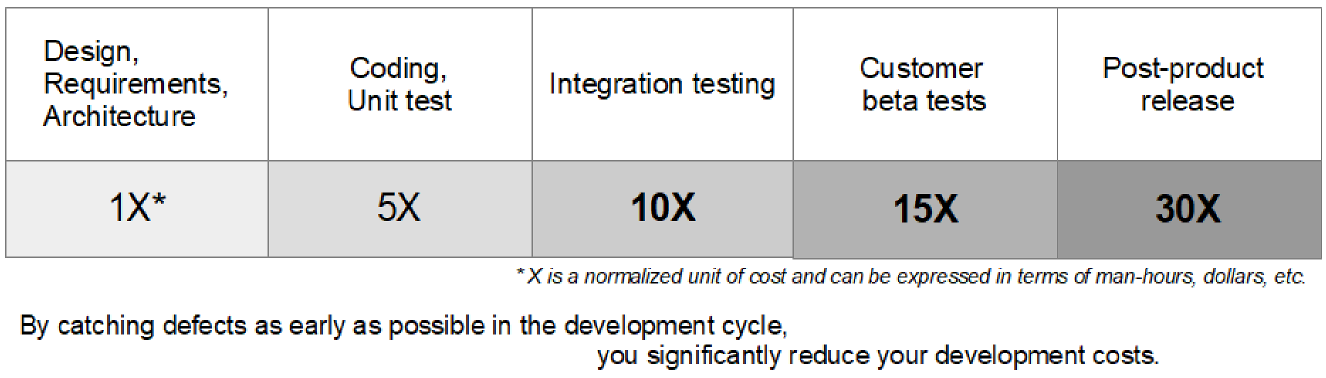
Code documentation can be handled variously in the code itself, in the descriptions of user stories, in architecture documents, and also in the **history of commits and by means of clear commit messages.** This can be handled as part of the new Git workflow.

2.2.5.3 QA: What to change and how?

Theory: Testing - Shift Left

Why build early, why do static analysis early, why test early, why review early? The reasoning is that if verification during the early development stages is not automated, some actions may be forgotten or skipped, leading to lost time and higher costs in fixing faults. For example, if change B depends on a previously completed change A, and verification is planned to be done after integration of both features, and change A has inbuilt design defects, it will be found only after verification is done (after integration), so not only change A but also change B would need to be rewritten - the effort to fix issues is effectively doubled due to verification being carried out later rather than earlier.

Numbers may vary, but all researchers agree that it is cheaper to find and fix a bug earlier in the development process. Data taken from (Tassey, 2002) is summarized in Figure 8.



Design, Requirements, Architecture	Coding, Unit test	Integration testing	Customer beta tests	Post-product release
1X*	5X	**10X**	**15X**	**30X**

**X is a normalized unit of cost and can be expressed in terms of man-hours, dollars, etc.*

By catching defects as early as possible in the development cycle, you significantly reduce your development costs.

Figure 8. Development phases and the cost of bugs shown from earliest to latest phase

A quick search of the software engineering literature yields many charts illustrating the idea that defects become exponentially more expensive to fix in later design phases. “Often they show the same power of ten per development phase multiplier we found in [the] hardware development literature.” (The cost of bugs – System Semantics, 2014)

Figure 9 shows one of the most demonstrative and self-explanatory visualizations, taken from the book “Software Testing” by Ron Patton:

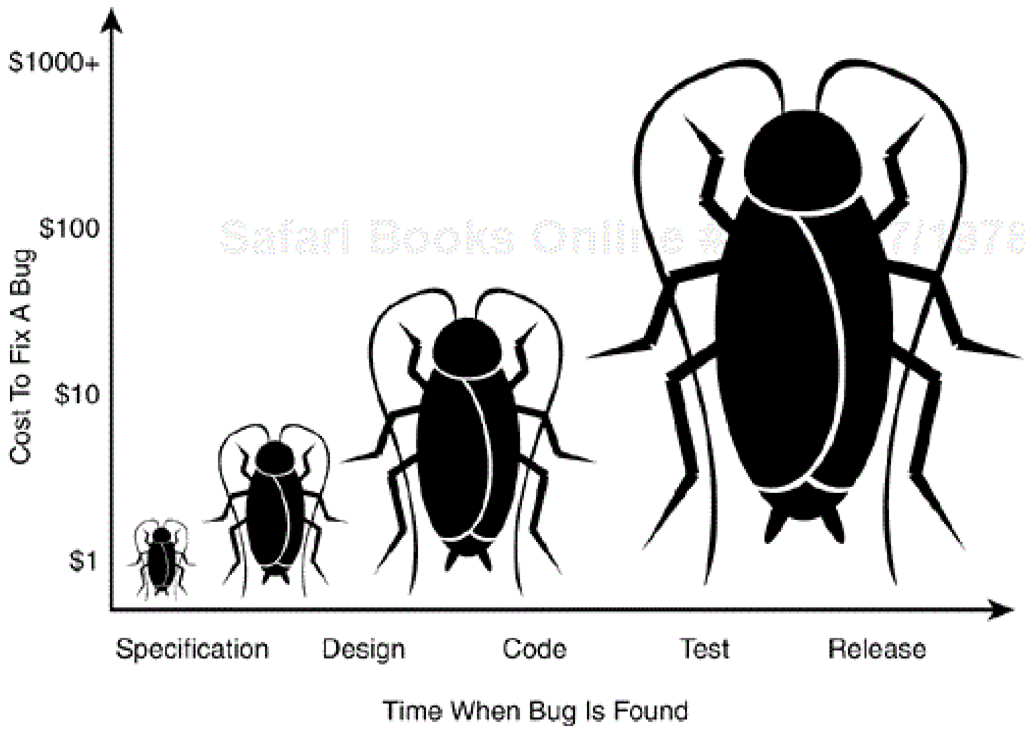


Figure 9. Software Cost-to-fix increasing 10-fold per project phase. (Patton, 2006).

Empirical: QA aspects for the new team to focus on

Author observations, discussions and questionnaire responses all made clear that the two most critical areas in QA to improve are the trustworthiness of TA and the focus on testing efforts. Previously we struggled with unstable and unreliable TA results, so developers lost trust in it - it ceased to be useful. A second problem was that testing efforts were not focused sufficiently on user-specific use cases and data, so we needed to add more exploratory and experimental testing that was done independently of the development team. Tests need to be prioritized such that most of the resources are spent checking the most crucial functionality and requirements (**efficient test coverage**). We likewise envisaged running the most critical tests often and quickly, while setting up less critical or time-consuming testing to execute on-demand **(more efficient use of time and resources for testing**). These two changes are quite extensive changes in themselves, so they were left out of the scope of the current study.

While changing the development workflow and process we sought to emphasize early testing efforts – the so-called “**Shift left”.** This shift left includes creating Pull Request from the feature branch to the main development branch and doing most of the testing on the feature branch before the Pull Request is approved.

Also considered was a renewal of review practices. It was suggested that the Pull Request annotation (cover letter, review record) could be used to emphasize early testing efforts and to involve developers more in QA, helping to make them feel responsible and proud of the quality achieved. The initial idea is that developers would give a summary for pull requests (PRs):

- summary of the change,

- engineering testing log,

- regression tests = “test to fail”,

- impact (regression analysis), what needs special attention,

- "technical debt" analysis like "Watch out for this thing I made, since I cut corners as discussed by the rest of the team to make a minimum iteration. But in future we should ..."

This can be encouraged by setting up a Pull Request template in Azure DevOps.

Since the beginning of the new QPR UI 2.0 development project, special effort has been made within DevApps to ensure **the CI&TA cycle is fast and short**, so developers do not have to wait too long to get their change tested and to see if any fixes are needed.

Scrum: What to change and how?

Encourage **planning granularity**: the idea is to have a hierarchy of work items [Epic – Feature – User Story - Tasks] and to divide the User Story into small tasks. This will likely improve estimation, since one of the main features of a great team is predictability. Smaller tasks help to split work and to deliver value increments (to use the Scrum jargon) more often. **Smaller user stories** and tasks also help avoid merge conflicts, since smaller user stories inherently involve smaller code changes. Small tasks are easier to complete and build confidence and pride. It brings satisfaction when small things are being completed every day. This creates positive reinforcement for learning. Well-defined tasks **increase transparency** in the daily work, as well as decreasing the "bus factor": it should be possible for different team members to work on the same task or to pick a task and continue the work on it.

The definition of done (**DoD**) needs to be defined, as well as what it means when the User Story is "ready for sprint" (i.e. what is the 'definition of ready'?). The definition then needs to be consistently applied and upheld, without violations. **Don't pick items for sprint User Stories that are not ready according to the definition of ready**.

Commit to less (Better to promise and deliver rather than to over-promise and fail). Define the minimum of "must have" items for the sprint.

Ensure that every item has an **owner** who takes responsibility to complete it.

Have more **iterations with customers**, define customer persona, know and understand your customer.

2.2.5.4 Contribyte "QPR 360 assessment" report Aug-Sept 2019

In 2019, an external consultant was invited to assess the QPR product development process, and their findings are listed here in brief (taken from their presentation). Main improvement areas (as relevant to this study) are as follows:

Gaining customer insight. This is aimed at increasing your awareness of your customers. Customer insight is generally poor at QPR and understood by a small number of people. Customer understanding within R&D is especially limited. Create user-personas for your products.

Tune the R&D engine. Take into use tools to support modern development – that is set up a 'toolset configuration', which improves transparency. Start recording and updating task progress in the appropriate tools, and ensure that progress is updated at least once a day. Provide access to your progress information. Refactor both the products and the testing environments. Your current toolset configuration doesn't support transparency and progress sharing, and the tools do not optimally support the team.

Scrum upgrade. Focus on improving team-level processes by increasing teamwork efficiency and creating customer value. Guide your progress with data – start using your data and making decisions based on it. Use daily cafes instead of daily stand-ups. “You are not getting the full value from Scrum. You don’t have goals for your releases and Sprints, estimates are subjective and not tracked. Stories are not appropriately defined (no acceptance criteria, no definition of ready, often too big) and are not verified against Definition of Done. The focus of Sprint planning is off target, and you don’t keep your backlogs in shape.” Include sprint pre-planning and backlog grooming into your process.

Revising testing strategies. Quality and testing needs to focus on revising the testing strategies, reconsidering the testing methods and addressing the technical debt in both your products and your testing platform. Experimental testing needs to be increased, with a focus on effective and affordable defect identification.

Knowledge sharing. Aim to harness the collective knowledge and wisdom of QPR and to address the challenge of personal knowledge. Form professional guilds and dissolve personal silos – harness your collective wisdom and share it.

2.3 Initial Scrum practices: August - September 2019 Iteration

QPR Software uses the Scrum framework to manage the software development process, but as with many process elements, Scrum is not applied that strictly. As a part of organising the new team’s development process, the author set out to refresh everyone’s knowledge about Scrum and to adjust our working practices to extract more benefits from the Scrum framework.

2.3.1 Theory: Tuckman's stages of group development

Tuckman's stages of group development are briefly described in Wikipedia: “these phases are all necessary and inevitable in order for the team to grow, face up to challenges, tackle problems, find solutions, plan work, and deliver results.” (Tuckman’s stages of group development, 2019) Figure 10 shows all the stages from the formation of the group up to the break-up of the team.

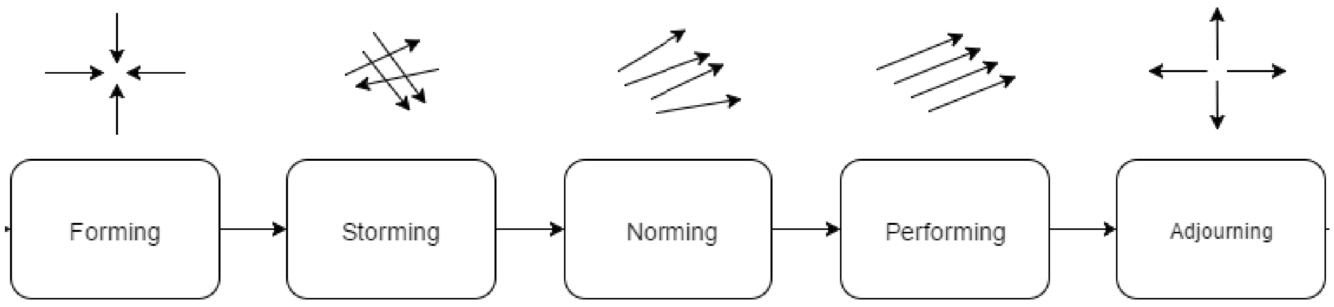


Figure 10. Tuckman's stages of group development. Source: wikimedia.org (CC BY-SA 4.0)

Forming phase: Why we are together? What is expected of us? What is expected of me? The team is focusing on its purpose and tasks.

Storming phase: sense of familiarity, frustration arises, active and passive conflicts. So, raise your voice asap. It takes about 3-4 sprints to arrive at the next stage.

Norming phase: solving conflicts, balancing mutual expectations, agreeing on norms, values and rules. It takes about 2 more sprints to arrive at the next stage.

Performing phase: safety in the team, no more fear of mistakes, and a shared sense of what is important and what is not, constructive work. Team members stand in for missing buddy.

Adjourning phase: say goodbye.

(Tuckman's stages of group development, 2019)

10 practical things to do to start (Verwijs, 2019) No rush through phases!

★ Reserve time for the Kickstart - foundation of the team. Setting up a Product Backlog.

★ Teach Scrum team to be on the same page.

★ Formulate a Team Vision - What it means to be a good team? Create a team manifest. Norms, values and principles on how people want to work together.

★ Getting to know each other. Break the ice and to get people out of their comfort zones. Use team-building games.

★ Create a Team Contract. Define roles and tasks: Who is responsible for what?

How do we register the team's success? What are the working rules?

★ Pick a Team Name. People more easily associate themselves with a group if you give them something small to identify with. Especially when there are other teams close by.

★ Set expectations. It will take time to normalize, we should expect the first few sprints to be challenging.

★ Retrospectives - as an opportunity to grow, learn and improve. Never skip it. Try different approaches to streamline routines.

★ Involve management to support the Kickstart. A Scrum Team will be dealing with a lot of complexity, both internally and externally. When kickstarting a new Scrum Team, it helps to ask management to come and show their support for the process.

★ "Bring it to the team" - The role of the Scrum Master is not to solve the problems for a team, but to help the team solve the problem themselves. Let the team come up with solutions themselves.

2.3.2 Team kick-off: Create a cultural manifesto for the DevApps scrum team

The name of the new experimental team to refresh the software development process and start the new product development was chosen by voting on alternatives suggested by everyone and “DevApps” received the most votes. It is an intended pun combining the idea of application development and the DevOps approach.

A workshop was organised, and the team created the manifesto.

What is a Team Manifesto? - A shared vision about teamwork and the required quality standards. Common norms, values and principles are agreed among the team members. Figure 11 below illustrates very well why a common understanding of goals and values is important.

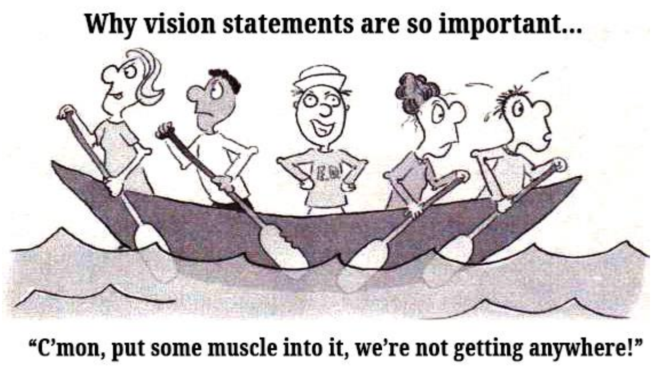
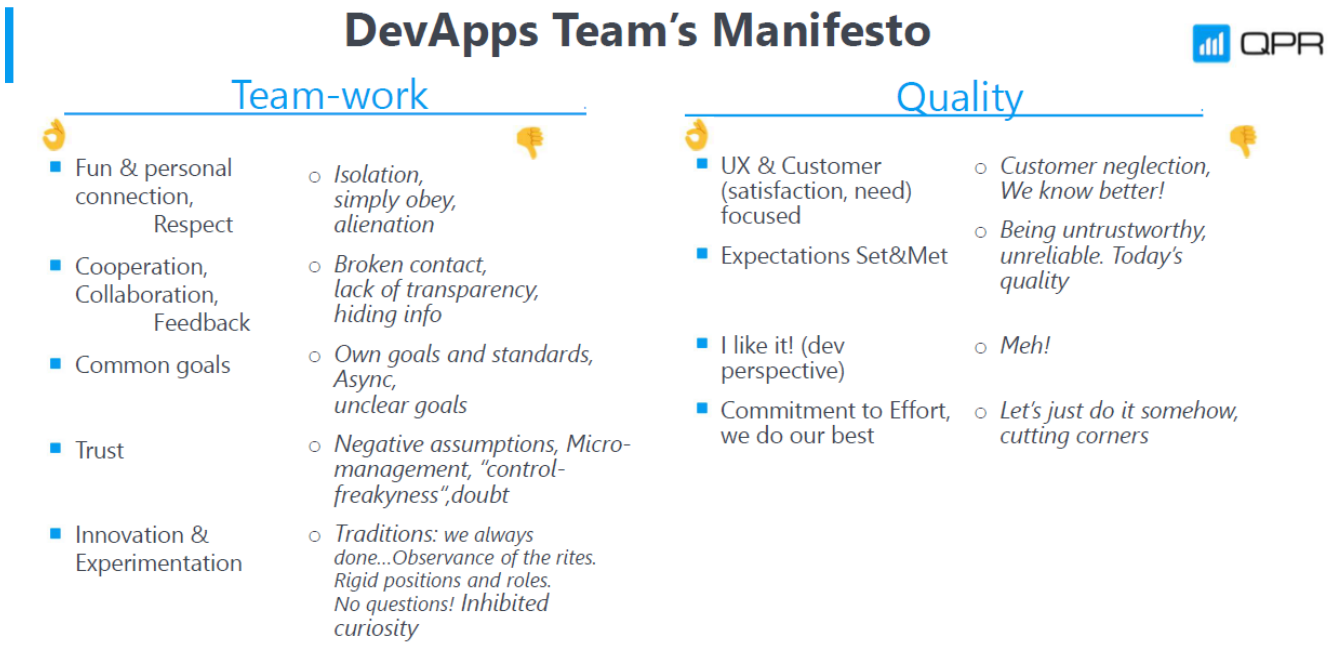

Why vision statements are so important...
"C'mon, put some muscle into it, we're not getting anywhere!"

Figure 11. Why it is so important to row in the same direction (How do leaders create inspiring visions?, 2015)

Why create a manifesto? - Values give direction to our work, our behaviour, and our actions. "Members of truly cohesive teams trust one another, engage in unfiltered conflict around ideas, commit to decisions and plans of actions, hold one another accountable for delivering against those plans and focus on the achievement of collective results" (Overeem, 2014).

A common understanding of what Quality means to us helps to tap into the team's sense of professional pride. It's self-enforcing. Since the team came up with it, individuals are more likely to behave responsibly and encourage others to do the same. Figure 12 shows the Manifesto created by the DevApps Team.



DevApps Team's Manifesto

QPR

Team-work

👌	👎
Fun & personal connection, Respect	*Isolation, simply obey, alienation*
Cooperation, Collaboration, Feedback	*Broken contact, lack of transparency, hiding info*
Common goals	*Own goals and standards, Async, unclear goals*
Trust	*Negative assumptions, Micro-management, "control-freakyness",doubt*
Innovation & Experimentation	*Traditions: we always done...Observance of the rites. Rigid positions and roles. No questions! Inhibited curiosity*

Quality

👌	👎
UX & Customer (satisfaction, need) focused	*Customer neglection, We know better!*
Expectations Set&Met	*Being untrustworthy, unreliable. Today's quality*
I like it! (dev perspective)	*Meh!*
Commitment to Effort, we do our best	*Let's just do it somehow, cutting corners*

Figure 12. DevApps's Manifesto (08.2019)

2.3.3 Team kick-off: Create working contract for the Scrum team

“The purpose of the working agreement is to ensure the Agile Team shares responsibility in defining expectations for how they will function together and to enhance their self-organization process. It creates an awareness of both positive (and negative) behaviours that can impact the Team and empowers the Scrum Master to keep them accountable” (Establishing an Agile Team Working Agreement - Tech at GSA, n.d.).

The team-working contract defines the “ground rules” for how the team will work together – it expresses the shared understanding of responsibilities and expectations. I decided to use The Role Expectations Matrix activity which is inspired on the Give and take matrix form the ‘Gamestorming’ book by Dave Gray, Sunni Brown and James Macanufo. (“Role Expectations Matrix | Fun Retrospectives,” n.d.)

This action was postponed for a later stage after which team members would have some practical experience of working together and would be able to see in practice what they expect from others and what others expect from them. It was planned to this after several sprints had passed.

2.3.4 Practical plan: Initial Scrum process setup

With the help and contributions of other team members, I made a summary of the main Scrum concepts and put those to the team’s wiki pages, so anyone could go and check information about the main Scrum concepts and events in short form. Here I present a condensed version of these wiki pages.

The initial setup for Scrum was based mostly on the Scrum Guide (Scrum Guide | Scrum Guides, 2018).

Scrum values

Every element of Scrum entails a goal. It is not recommended to change the core design of Scrum, to leave out elements, to not play the game by its base rules, since this has the effect of covering up problems and limits the benefits of using Scrum and any additions to Scrum, even to the point of rendering it completely useless. For this reason, it is important to repeatedly remind the team about the Scrum values:

1. Commitment - from previous experience I’ve noticed that commitment is often confused with “a promise to deliver”, though in fact commitment means dedication to quality and effort. And the focus is

on forecasting, on getting predictable outcomes. It was the useful therefore to shift the focus from promises to forecasting.

2. Focus - time-boxing activities and staying on-topic during meetings, focusing on the team's common goals and priorities, rather than on individual ones.

3. Openness (transparency)

4. Respect (diversity, consumer)

5. Courage (passion)

From an article by Gunther Verheyen, author of "Scrum - A pocket guide" (Verheyen, 2013)

Scrum events: Daily stand-up

I would argue that one of the biggest issues with daily stand-ups is routine. In order to fatigue with routines, I collected additional questions to ask occasionally during daily meetings:

- Are there any pull requests waiting for review?
- Any ad-hoc side tasks (not initially planned for the sprint)?
- Share gotchas and helpful findings, retrospective notes, fun facts.
- Do you need any help?
- How can I help the team's progress towards the sprint goal?
- Are we focused on reaching the sprint goal?
- Compliment your buddy, or give other kinds of feedback

Another important issue that needs follow up is transparency about daily progress, meaning that the team needs to have visibility about progress with tasks and it should be verified during the daily stand-up. If an update has been missed, they it should do it before the next daily.

Scrum events: Product backlog refinement

Product backlog refinement (also referred to as backlog grooming) is intended to prioritise user stories and break them down into smaller tasks. This happened mostly in meetings involving all team members and where the PO was "giving a task to the team". The aim is to shift towards more individual work before the backlog grooming meeting and encourage participants to take the initiative in discussing and challenging what is suggested initially. Also, it was recommended we move from backlog refinement that only takes place in scheduled meetings towards a more continuous process.

"This activity is all about interaction between the Product Owner, Development Team and stakeholders. If you were expecting a blueprint for a 'ready' item, you clearly need to do some homework on agility" (van Rooden, (1/3), 2016).

Scrum events: Sprint Planning

We aim to focus on the **Definition of Ready** for product backlog items and to pick for the sprint only those user stories that are groomed and defined well enough so that the team could start working on them.

The purpose of each Sprint is to deliver Increments of potentially releasable functionality that adhere to the Scrum Team's current "**Definition of Done**". As an outcome of the Sprint Planning, the team should be able to answer the following:

- What is this **Sprint's Goal**? What can we deliver in this upcoming sprint? - Forecast.
- How will the work that is needed to deliver the Increment be achieved? - Tasks.

("Scrum Guide | Scrum Guides" 2018)

Scrum events: Sprint Review

We used to skip the Sprint Review with the previous team where I worked. Now we will bring it back as a way to present achievements to customers, to celebrate the progress we have made, and to check what wasn't done and why, and how we can adjust the Product Backlog.

Result: revised Product Backlog ready for Sprint Planning.

("Scrum Guide | Scrum Guides" 2018)

Scrum events: Sprint Retrospective

Retrospective - this is an opportunity to grow, learn and improve. It should never be skipped. It is one of the most crucial parts of the Scrum iteration cycle, being an opportunity to reflect and discuss what was good and what was not so good. The main thing is to plan action points to improve the situation. It is the responsibility of the Scrum master to follow up and make sure actions are carried out and changes happen.

For the beginning I used the Mad Sad Glad game, which "frames discussion around the emotional journey of your team during the previous sprint" (Mad Sad Glad Retrospective," n.d.).

In future, we will also regularly review the working agreement (Team contract), once we have it in place, especially in the event of any changes or if a member consistently breaks one of the ground rules.

To minimize routine and facilitate discussion we will explore different approaches and new formats for the retrospective. This will be implemented in future springs, but in the initial stages, I will stick to Mad Sad Glad.

Estimation and Velocity

“Teams use t-shirt sizes, the Fibonacci sequence, or planning poker to make proper estimations” (Atlassian, User Stories, n.d.).

We used to estimate time for completing features measured in hours and it was too fine-grained and almost never matched the actual time spent on feature development, which might easily be double or more.

“The social and psychological aspects of estimation: using units such as story points and emphasizing relative difficulty over absolute duration relieves some of the tensions that often arise between developers and managers around [time] estimations: for instance, [when] asking developers for an estimate then holding them accountable as if it had been a firm commitment.” (Agile Alliance, What are Story Points?, 2015)

The intention with DevApps is to try to forget about real time and use story points - coarse-grained and relative measures for estimating the effort required. We will put the focus on learning the team’s velocity in abstract points, and monitor and based on that adjust our estimations regularly. We will add up the effort estimates associated with user stories that were completed during the previous iteration - this gives a value for the velocity of the team (What is Velocity in Agile? | Agile Alliance, n.d.). This metric will then be used as a maximum limit for the next sprint. With continuing iterations, velocity would be expected to stop oscillating and become more or less stable. The aim is to be able to give a forecast and state that pre-refined user story X will take Y story points, which can be approximately related to the sprint schedule. We also intend to learn how to limit our sprint plan, so we first would learn to focus on a sprint goal and complete and achieve what was planned. “Better [to] promise and deliver rather than over-promise and fail.” (Pilone and Miles, 2008)

2.4 Theory: Version control system and the development workflow

2.4.1 What are version control systems (VCS) and how do they compare?

“Configuration management” (CM) is also used as a synonym for version control. “CM refers to a process by which all artefacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified.” Those artefacts may include source code, database definitions, documentation, build scripts, tests etc. The decision to use one specific version control tool is the first step in developing a configuration management strategy (Humble and Farley, 2010).

“Version control systems (VCS), also known as source control, source code management systems, or revision control systems, are mechanism for keeping multiple versions of your files” (Humble and Farley, 2010). The source control is an essential tool for multi-developer projects. Source or version control systems **allow developers to collaborate on code and track changes made to the code base**.

Subversion and Git are among the most popular version control systems currently (Humble and Farley, 2010).

Git versus Subversion (SVN)

In the case of QPR Software, no other VCSs were in use to provide a comparison other than the currently used SVN, along with Git as the desired alternative, so the question boils down to Git vs SVN. I focused only on the differences most relevant to the QPR work environment.

First of all, it has to be mentioned that **SVN is a centralized** VCS, which has a main repository on a single server, while **Git is a distributed** VCS, where each user keeps a self-contained repository on their computer. “Every local repository is effectively a branch in its own right, and there is no mainline” (Humble and Farley, 2010). Nevertheless, it is conventional to agree on which branch to run code integration (CI). It should also be noted that since Git is distributed, not everything related to a project must be stored in the same location, so it needs some level of discipline to have all the important bits of code on one server.

The Atlassian documentation argues that using Git to power your development workflow presents a few advantages over SVN. First, it gives every developer their **own local copy of the entire project**. This isolated environment allows each developer to work independently of all other changes to a project (Atlassian, Git Workflow, n.d.). SVN also allows developers to have a local copy and to work independently, but those changes are **not checked in and not backed up** until they are sent to the central server. The difference centres on registering the change and sending it - in SVN when you make a commit, the revision immediately goes to the server, so you must also be online. However, in

Git you first choose which change you wish to add to be committed. You then make a local commit (possible even offline) and after that you can push it to server. The "staging area" or "index" allows you to quickly stage some of your files and commit them without committing all the other modified files in your working directory. Gone are the days of making two logically unrelated modifications to a file before you realized that you forgot to commit one of them. Now you can just stage the change you need for the current commit and stage the other change for the next commit." (Git - Book, n.d.).

The possibility to register code changes as commits to a local repository also enables developers to "**easily modify, reorder, or batch up" commits locally** (Humble and Farley, 2010).

A further point made by the Atlassian documentation was that Git "gives you access to [a] robust **branching and merging** model. Unlike SVN, Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories." Git also provides recovery tools for handling branch management failures. Git lets you create highly focused commits, even if you have made a lot of local changes (Atlassian, Git Workflow, n.d.).

Git supports collaboration on a branch: Since the system is distributed and all branches are equal, it is possible to pull changes from different branches and to collaborate without touching the main repository until it is needed (Humble and Farley, 2010).

Git helps to encourage engineers to frequently commit and rebase their work - Git makes **pull and rebase easy**. As detailed in the Pro Git book, written by Scott Chacon and Ben Straub: "The Git feature that really makes it stand apart from nearly every other SCM out there is its **branching** model." It also allows frictionless context switching, role-based codelines, feature-based workflow, disposable experimentation. Git is fast, providing multiple backups, and able to handle any workflow (About - Git n.d.).

Joel Spolsky, author of "Joel on Software" (Spolsky, 2004) and co-founder of Trello and Fog Creek Software, and CEO of Stack Overflow, advocates for Git in an article as follows. "Many Subversion users have told me the following story: 'We tried to branch our code, and that worked fine. But when it came time to merge back, it was a complete nightmare and we had to practically reapply every change by hand.' With distributed version control, merges are easy and work fine. If you are using Subversion, stop it. Just stop. Subversion = Leeches. Mercurial and Git = Antibiotics." (Spolsky, 2010).

Scott Chacon (the author of the Pro Git book by Apress) is another Git advocate: "I have done this for branches with one commit containing a one line change. The process is simple, straightforward, scalable and powerful. You can do it with feature branches with 50 commits on them that took 2 weeks, or 1 commit that took 10 minutes. It is such a simple and frictionless process that you are not annoyed that you have to do it even for 1 commit, which means people rarely try to skip or bypass the process unless the change is so small or insignificant that it just doesn't matter." (Chacon, 2011)

2.4.2 Choosing a version control system and provider, CI/CD system (tools pipeline).

We decided to try Git with the new DevApps team at QPR to be able to compare the two based not only on theory, but also on real-world usage experience.

Reviewed options were:

- GitLab
- GitHub
- BitBucket (Atlassian)
- Phacility/Phabricator
- Azure DevOps
- EficodeRoot
- Git server hosted internally at QPR
- SVN internal server (currently used)

Criteria for assessment were:

❏ Price to get the service

❏ Good value/money ratio: employee’s working hours vs price to pay for service

❏ IPR handling policy,

❏ ICT requirements (user management via AD)

❏ Finances, billing matters (add accounts under enterprise billing system)

❏ Possibilities to migrate to another service (vs lock-in)

❏ Tools integration

❏ Support for agile development (small changes, fast feature branch lifecycle)

❏ Developer’s experience

❑ Branching experience – easy branching and fast branch lifecycle

❑ Convenient review tool (visual, more collaborative - show a diff and put comments there, have discussion)

❑ Has Review (discussion) history

❑ version control + review + static code analysis + build + TA integration

❑ Prohibit direct commits to master - is it possible?

❑ Support for Review "before merge"

❑ Pull/Merge Request or patch-set annotation to make review easier

❑ Easily review any branches

❑ Support for static code analysis + code style check "before merge"

❑ Build every development branch/commit "before merge"

❑ Support for running TA "before merge"

❑ TA must be able to run on branch/commit

❑ TA must run super-parallel fast

❑ Packaging and deployment

❑ Support for tests prioritization / classification and grouping to run on different stages.

❑ Use different method (file system) to provide resulting build package to users (to avoid problem with reserved folder)

❑ Requirements – issue tracking system – code – tests traceability

❑ Hook: Automatically register commit in issue tracking system

Decision:

Based on the criteria listed above, we chose Azure, since it fulfilled the ICT requirements (can use AD access management), while the development tools pipeline seemed to have most of the required blocks already integrated, such that we would not need to create and maintain special scripts to make systems work together and enable information to flow through the systems.

It was also chosen due to its expansion capabilities (Basic + Test Plans) and relatively low cost (being cost-effective to run & maintain). It also enables us to take into use a more comprehensive set of tools related to work management, product backlog and sprint backlog management. The full ecosystem comes with integration with other popular services: work management, wiki, repository code, branches and pull requests, while TA&CI seem to be well integrated.

Some remaining questions centred on test management and requirements management. However, we can probably distribute the job of fulfilling those needs among several systems and tools. The Git database can be moved just as easily as from any other provider. Azure has Git history visualization - developers can see the commits in topological order as a graph. The review tool allows developers to see diff and comment inline, as well as the history, which covers our basic needs. It is possible to review any branches and make comments to commits. The possibility is also there to prohibit direct commits to the master branch.

Azure DevOps represents the evolution of Visual Studio Team Services (VSTS). It provides an integrated set of features that you can access through your web browser or IDE client

- **Azure Repos (Code)** provides Git repositories or Team Foundation Version Control (TFVC) for source control of your code
- **Azure Pipelines (Build & Release)** provides Continuous integration and continuous delivery (CI/CD) that works with any language, platform, and cloud.
- **Azure Boards (Work)** delivers a suite of Agile tools to support planning and tracking of work, code defects, and issues, using Kanban and Scrum, or a combination of methods.
- **Azure Test Plans (Test)** provides several tools to test your apps, including manual/exploratory testing and continuous testing
- **Azure Artifacts (Packages)** allows teams to share Maven, npm, and NuGet packages from public and private sources and integrate package sharing into your CI/CD pipelines
- **Collaboration tools** that include customizable team dashboards with configurable widgets to share information, progress and trends; built-in wikis for sharing information; configurable notifications and more.

The Azure DevOps online documentation can be accessed via: https://docs.microsoft.com/en-us/azure/devops/

3 Retrospective

As part of the Scrum process we held sprint retrospectives - meetings where the team could reflect on and discuss what went well and what needs to be improved. I used the "Mad-Sad-Glad" format to frame the discussion and collect feedback (Mad Sad Glad Retrospective, n.d.).

I analysed comments from the sprint retrospective meetings by classifying them into themes, taking into account how many times an issue was mentioned. Some bits of information or answers were either ignored or remained unidentified. I then identified classes within the data and listed examples of how informants mentioned each class. See meeting details in Appendix 5.

Some of the classes matched very well with the attributes of good teamwork as defined in the Team Manifesto (they could be considered the team values):

Fun & personal connection, Respect

Cooperation, Collaboration

Feedback

Common goals

Trust

Innovation & Experimentation

UX & Customer (satisfaction, need) focused

Expectations - set and met

I like it! (developer's perspective)

Commitment to Effort, we do our best

Additional classes (team values) were identified and added from the sprint retrospective discussions:

Agility of process

Feeling of accomplishment (achieving)

Usability of documentation and chats

Involvement, ownership

Freedom vs stability/certainty

Meeting efficiency

Tools satisfaction

3.1 Retrospective comments analysis 29.07-6.09.2019

In this section I combine the feedback gathered from the retrospective meetings held after each of the first two sprints: 29.07-16.08.2019 and 19.08-6.09.2019, as these two sprints included more planning and learning to prepare for the real development work.

The following themes were reflected on by the team as being positive in the first two sprints (Glad):

Innovation & Experimentation (mastery and autonomy) - team members were happy to learn new things, to investigate, to invent and come up with ideas. It was mentioned how good it feels to take part in the beginning of (green field) development.

Fun & personal connection, respect - many of us felt inspired, enthusiastic and commented on the good spirit. We took responsibility, had a good atmosphere and had fun.

Agility of process - the Scrum practices seemed to be already better than in previous teams; not getting bogged down with old manual processes felt good, as did the feeling of flexibility, informality, freedom, and a chance to self-organize.

Involvement and ownership - it was felt that team members had a lot more say in what to do and how, while feeling involved lead to caring more about the outcomes.

Feeling of accomplishment (achieving): things were progressing well. Decisions were made, the development environment was seen to be emerging, everybody felt we were moving forward, we had almost met the sprint goals, and things were also getting clearer, we understood our target better and implemented the “Hello World” App!

Common goals - planning the “critical path” (high-priority tasks) for the sprint was considered a very useful practice.

Tools satisfaction - the first trials using Azure DevOps gave an impression that it is an attractive set of integrated tools, and ideas about the team’s workflow started to evolve.

Some comments on what made team members glad after the first two sprints:

- The start-up phase is the best a developer can attend, the atmosphere is good, a promising start, enjoying it.

- Flexibility in day-to-day work (currently), informality, no strict rules about following tasks; feeling of freedom; time to invent, learn and have ideas.

- Being involved in setting the vision, brainstorming, hence caring more, increased ownership.

- Learning new skills (React, node, git etc).

- Possibility to create something totally new!

- Things are getting clearer and the common goal is more visible, and we get to know and understand what our target is.

- Things are progressing, people are delivering useful stuff, and we almost completed the sprint goal

- Good team spirit and atmosphere, work is almost fun, things are progressing

- We have a kind of Hello World!

Points for improvement, worries or concerns mentioned (Sad&Mad):

Usability of the documentation and chats - the Microsoft Teams chat seemed for some of us too messy and too noisy. Information was scattered everywhere across different systems and locations, so some team members had difficulties in finding documents. We also did not yet have agreed instructions for the development work.

Common goals - we did not define the sprint goal. In the beginning, we were not sure how and where to create the product backlog and so we started initial development without a backlog. The team was not yet taking enough ownership of user stories, not enough work was being done in reducing user stories, not everyone was able to participate in the common setup tasks (setting up practices, environments, TA&CI).

Freedom vs stability or certainty - it felt overwhelming, like we had too many things to do at once, too many changes happening at the same time, many decisions needed to be made and the team felt a bit slow in making decisions.

Feeling of accomplishment (achieving) - not all planned tasks were completed.

Cooperation, Collaboration - the risk of collective responsibility morphing into no one's responsibility was brought up, the desire to create a prototype sooner led to neglecting documentation and communication.

Meeting efficiency - it felt like we had too many meetings.

Agility of process - a worry about throwing away good established working practices was raised. Not documenting our decisions risked undermining the possibility to learn in the future and made it more difficult to serve as a case study and example for other development teams. The term "user story" was confusing for some team members.

<u>Some comments on what made team members sad or mad after the first two sprints:</u>

- "There are so many balls up in the air"
- "So many things to do that it's overwhelming, we need to focus and prioritize"
- "Feeling a bit slow with making decisions, too worried about risks."
- "Lots of things to be decided"
- "Not aggressive enough in reducing user stories"
- "Some tasks are not yet finished"
- "There were good established working practices (Entice and MEA) that were refined over many years, aren't we throwing them away?"
- "Co-creation (creating together, collaboration) is great, but sometimes shared responsibility leads to irresponsibility. Collective responsibility leads to no one's responsibility."
- "Transparency: If we want to see the total work remaining for reach a sprint goal, then we need to have an initial estimate and work done (marked every day, so we can track daily progress)."

3.2 Agreed adjustments

As a result of reflection on the first two sprints, the team agreed to:

Document our decisions.

Continue aiming to follow the Scrum framework better than before, but not to make too detailed time allocations for team members and allow flexibility instead.

Estimate and plan work in abstract units - story points.

Treat the sprint plan as a forecast, but not as a promise.

Aim to organize, refine and prioritise the backlog.

Aim at reducing or splitting stories.

Define the sprint goal and aim to complete sprint tasks as a first priority.

Assign all tasks, so every task would have a responsible owner.

Set up and use a calendar reminder for dotting i's and crossing t's - finalizing tasks neatly.

Make a wiki page to store links and accompany them with short summaries.

Keep valuable old practices such as specifying and planning work items (user story or SDTs) before starting to implement, adding a description to a user story or SDT, write up the requirements, create a more detailed specification in a work item if needed (in many cases it is indeed necessary). Everyone in the Team must have a good understanding of the task: what will be done, what not, what is the “guesstimate”, any known grey areas (should not be too large), regression and other risks, test plan, support materials.

Treat the user story as the main "working document", i.e. centralize all related information and links to other systems.

4 Iteration 2: September 2019

Continue following the initial plan, while considering previous observations and conclusions.

4.1 Organize issue tracking and work management (backlog, epics, stories, tasks)

First we tried to create a hierarchy of epics, user stories and tasks in Axosoft, though it does not adequately support this kind of approach. We therefore turned to Azure DevOps, which provides work tasks management that integrates well with the development process. Another significant shortcoming is that Axosoft is very slow in usage, with the overall subjective user experience appearing to be bad according to what I have read and heard. A comparison between Axosoft and alternatives can be found at slant.co ("Slant - 20 best alternatives to Axosoft as of 2019," n.d.)

4.1.1 Epic - Feature - Story/Topic - Task

Initiatives are collections of epics that drive toward a common goal. **Themes** are large focus areas that span the organization. And an **Epic** is a story that can be considered as a goal to be attained by the team. An epic describes end-user behaviour or drives change in end-user behaviour. Quite typically, an epic describes a task or job an end-user wants to accomplish, such as clarifying and reporting the state of a business unit in the corporation (Atlassian, Epics, Stories, Themes, and Initiatives,n.d.). An epic comprises a collection of related and interdependent user stories. How the user stories are related to each other becomes evident from a user story map that is defined as part of the epic. An epic may take several sprints to complete. An epic is never entered as a reference in a commit (Epic, Story, Task and Bugs - Developer Wiki - Confluence, n.d.).

User story (value added looking from the user's perspective)

The story is the smallest unit of work that brings value to the customer. It is explained in a few sentences in simple (not technical) language. The story helps to keep the focus on the user and on solving problems for the user, rather than being focused only on a list of tasks. Stories enable collaboration when several people work on the same goal doing different tasks.

User stories are often misunderstood as lightweight requirements given by the business stakeholders to the delivery team. To be clear, if a team passively receives documents in a hand-over, regardless of what they are called or whether they are written on paper, in a wiki or in a ticketing system, that's not really working with user stories. Organizations with such a process **will not achieve the full benefits of iterative delivery** (Atlassian, Epics, Stories, Themes, and Initiatives, n.d.).

To define a story we agreed that:

A story must be written as a user story (i.e. "As a <kind of user> I want <feature> so that <benefit>")

It is clear for whom – specify the user (if they are several different users then it may be another story)

Get feedback from the user (to check if you defined story correctly)

We agreed that we want requirements to be specified

Performance criteria and other non-functional requirements exist

Design sketches exist, where appropriate, and are understood according to the team's Definition of done

Tasks and sub-tasks

Acceptance criteria exist and are understood by team

Story has been estimated by the team and time to implement is less than one sprint

The team understands how to demo the feature

Anyone can pick a task and start working on it

New user stories and tasks can be identified and created by anyone in the team. Their prioritization is decided (mainly) by the product owner after discussing the business needs and technical feasibility, as well as other possible factors, with the relevant stakeholders and the team.

Tasks - small tasks that can be assigned to one person (like a UX or test or implementing some part of functionality). Task represents a technical activity, such as being asked to design a diagram, code a functionality, test a device, or prepare a dataset (Epic, Story, Task and Bugs - Developer Wiki - Confluence, n.d.).

4.2 Agree on the coding workflow and branching strategy

What is a branching strategy? It means all distributed branches of code are under control and to know how changes in code will be promoted from the first commit to public release. It is useful to ask what the rules will be for moving from one stage to the next. A branching strategy needs to be crafted. It covers which branches we are going to have, how code moves from one to another, and what are the rules (policy) for each branch.

Azure DevOps: Git branching guidance:
“Distributed version control systems like Git give individuals wide flexibility in how they use version control to share and manage code. Your team should find a balance between this flexibility and the need to collaborate and share code in a consistent manner” (Adopt a Git branching strategy, 2018).

“Since many organizations new to Git have no conventions for how to work with it, their repositories can quickly become messy. The biggest problem is that many long-running branches emerge that all contain part of the changes. People have a hard time figuring out which branch has the latest code, or which branch to deploy to production” (“Introduction to GitLab Flow | GitLab,” n.d.).

4.2.1 Questions and Conceptual decisions to make

- What would be the optimal and simplest coding workflow for this team and this product?

- How to make our tools developer-friendly?

- Which branches make sense to have for our needs?

- Branch naming convention:
Story branch, topic branch or feature branch? Git flow uses the “feature/” prefix, we probably won’t use it to simplify our flow in the beginning.

- Which kinds of environments do we need for testing - “staging”, “testing”, “nightly”, “development”, “release”, “pre-production” or “production”?

To Fork or not to fork?

Atlassian: forking-workflow is often used in public open source projects when each contributor has two Git repositories: a private local one and a public server-side one (which is a fork of the main upstream project repository). (Atlassian, Forking Workflow, n.d.)

Shared Repository Model (no forks) uses a single server-side repository acting as the “central” codebase.

For QPR UI development we want to pursue as light as possible a method for working with branches and repositories, so at this point there seems to be no need for forks from the main repository.

Pull Requests

Starting to use Pull Requests (PR) is one of the main, if not the most crucial change in the development process. It was actively discussed and agreed to be a good mechanism to enforce verification before a code change from a developer could be accepted into the main development branch (previously named “trunk”, now named “develop”)

PR is a great code review system, not just a way to promote your change to the master branch. You can use them to say “I need help on or a review of this” in addition to “Please merge this in” (Chacon, 2011).

“You can open a (WIP) Pull Request at any point during the development process: when you have little or no code but want to share some screenshots or general ideas, when you're stuck and need help or advice, or when you're ready for someone to review your work. You can also continue to push to your branch in light of discussion and feedback about your commits” (Understanding the GitHub flow · GitHub Guides, n.d.).

In the GitLab terminology the request to integrate a change into the master branch is called a “Merge Request”, rather than a “Pull Request”, which for our purposes is the same thing. GitLab.com about testing before merging: “In old workflows, the continuous integration (CI) server commonly ran tests on the master branch only. Developers had to ensure their code did not break the master branch. When using GitLab flow, developers create their branches from this master branch, so it is essential that the master never breaks. Therefore, each merge request must be tested before it is accepted....There is one drawback to testing merge requests: the CI server only tests the feature branch itself, not the merged result. Ideally, the server could also test the master branch after each change. However, retesting on every commit to [the] master is computationally expensive and means you are more frequently waiting for test results. Since feature branches should be short-lived, testing just the branch is an acceptable risk. If new commits in master cause merge conflicts with the feature branch, merge master back into the branch to make the CI server re-run the tests. As said before, if you often have feature branches that last for more than a few days, you should make your issues smaller.” (“Introduction to GitLab Flow | GitLab,” n.d.). This is relevant to the current QPR UI 2.0 project development in AzureDevOps: in order to minimize the chances of a conflict, a developer needs to rebase on the latest develop branch before making a PR, but while the PR is going through review and approval time goes by, there may be another PR merged into the develop branch, which creates a potential risk of merge conflicts or some failures as a result of integration. For now, while we have a fast build & test cycle and until we find a better way to overcome the risk of conflict and the integration problem, we will be retesting on every commit to the develop branch.

Feature branches vs Continuous Integration contradictions

Feature-driven development , feature branches and cherry-picking features for release may lead to merge conflicts if the branch stays isolated for too long and diverges too much from the mainline. Martin Fowler (author of a half-a-dozen books on software development) says: “Fear of big merges also acts as a deterrent to refactoring”. As a result, “teams using feature branches shy away from refactoring”. That’s why he considers Feature Branching to be a bad idea compared to CI (Fowler, 2009).

GitLab provided an answer to a discussion about the size of changes introduced by a feature branch and its life-time: “Most feature branches should take less than one day of work. If your feature branches often take more than a day of work, try to split your features into smaller units of work” (Introduction to GitLab Flow | GitLab, n.d.).

To my mind, the argument about branches versus CI boils down to talk about the divergence between main development branch and the feature branch. Every team probably needs to find their own optimal pace to integrate new changes into the master branch. Moreover, rapid integration is necessary for sharing and reusing blocks of code, but also for seeing how different development efforts (different features) depend on each other, which enables developers to agree on how to collaborate and how to make the code modular to lessen intersections. “By looking at both their features they can come up with a better design that affects both their work-streams. With the isolated feature branches our developers don't discover this till late, probably too late to do much about it” (Fowler, 2009).

Merge vs Rebase

What is best, to Merge or Rebase? The answer is not so straightforward.

In an Atlassian Git blog, Nicola Paolucci discusses merge vs rebase (Git team workflows: merge or rebase?, 28.10. 2013). She discusses the best way to incorporate implemented new functionality back into your main line of development. The topic is controversial and there is a lot of heated online debate between the two camps: always rebase vs. always merge.

Merge and Rebase are designed to integrate changes from one branch into another branch - they just do it in very different ways. Rebase gets the latest updates from one branch (usually the master) and then puts all changes made in another branch (the feature branch) as the new commits on top of what was there in master. In other words, rebasing works by abandoning some commits and creating new ones (Atlassian, Bitbucket: Git fast forwards and branch management, n.d.).

Rebasing is like saying, “I want to add my changes on top of what everyone else has already done.” (Paolucci, 2014). Figure 13 explains how rebasing your feature branch in master looks like.

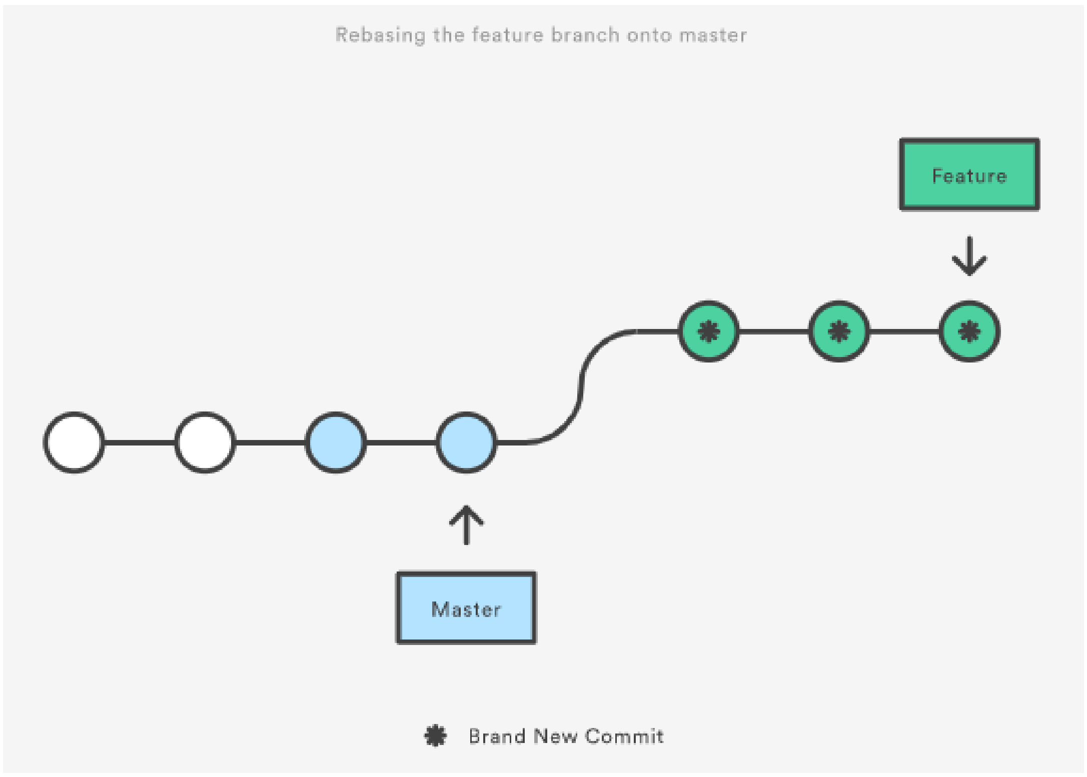


Figure 13. Rebasing the feature branch onto master (Atlassian, Merging vs. Rebasing, n.d.) (CC BY 2.5 AU).

Figure 14 explains how merging your feature branch with the master looks like.

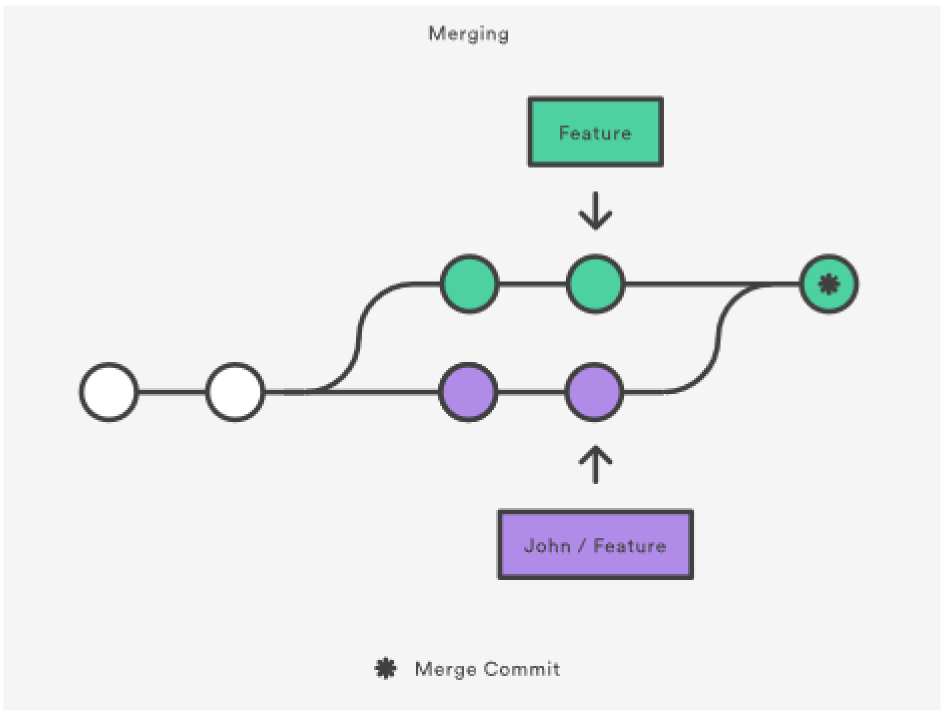


Figure 14. Merging two branches together (Atlassian, Merging vs. Rebasing, n.d.) (CC BY 2.5 AU)

Merge ties two branches together by creating a merge commit and preserving their history. Merge enables better traceability (at the expense of readability and clarity) compared to Rebase. But if development is very active, the merge commits can pollute the branch's history (Atlassian, Merging vs Rebasing, n.d.).

There is obviously an agreement that it is good to rebase when you are developing locally to keep the history tidy. It is good practice to rebase your feature branch on top of the main development branch when it has not yet published (pushed to the server). Rebasing will keep your history neat and linear. With an interactive rebase you can clean up your feature-in-progress development history by eliminating the insignificant commits (merge can't do that). Overall, Rebase seems to be beneficial when used with care (keeping in mind that rebasing may be destructive). The tradeoffs are safety and traceability.

Rebasing may involve the side effect that you have to resolve similar conflicts again and again (Introduction to GitLab Flow | GitLab, n.d.) - in this case, it is good to reuse recorded resolutions - rerere. Also, when a local branch is pushed to the server, it is made public and the Golden Rule says you should not rebase anymore, since other developers may now check out your branch and commit changes to it (Atlassian Git tutorials: Merging vs Rebasing). After the branch was published, any changes from other developers need to be incorporated with git merge instead of git rebase. However, if you can agree with colleagues on how to collaborate on your branch and pull new changes from each other, and if you know what you are doing and you coordinate your actions with other contributors to your branch, then it may work satisfactorily.

Force-Pushing: If you try to push the rebased X branch back to a remote repository, Git will prevent you from doing so because it conflicts with the remote X branch. But you can force the push using the "--force" flag. It overwrites the remote X branch to match the rebased one from your local repository. This is safe to do if you are the only one who contributes to this branch and others only review.

It therefore appears to be a good idea to avoid merge commits, but not to eliminate them. In the event of merging, the history represents what has actually happened in detail. But an explicit merge commit may be useful to mark the point when the feature "graduated" to release (Paolucci, 2013).

In our workflow we want the codebase to be clean and the history to represent what happened on a general level, so we will aim to rebase when possible during development in the branch. When development in a feature branch is complete, we intend to rebase/squash all the work down to the minimum number of meaningful commits and integrate the functionality that is ready into the main code base with fast-forward.

Fast-Forward vs Non-Fast-Forward

A non-fast-forward merge is a merge where the master branch has had intervening changes between the branch point and is merged back into the master. In this case, a user can simulate a fast-forward by rebasing rather than merging (Git fast forwards and branch management - Atlassian Documentation, n.d.). As stated in the Introduction to GitLab Flow on GitLab Docs, a merge strategy called "no fast-forward" in Git, using –no-ff flag – merge (pull) requests, always creates a merge commit, even when the branch could be merged with a fast-forward (FF) without creating a merge object.

An FF strategy implies that pull requests must be rebased on the latest develop branch before being accepted. In this way, only FF merges are accepted into the develop branch.

If a pull request (PR) with change A happens to be accepted and merged into the master between the moment when the branch with change B was rebased on the latest master and the moment when PR with B is merged, then the result of integrating A + B may break the master, since this code combination was not verified. It would therefore be reasonable to ask how to prevent making a pull request merge without rebasing to the latest master. I see two means available in Azure DevOps, one being a timeout for the build result - when the pull request is initially created, the build and test runs are done on an integration of the feature branch and master branch, though this result becomes invalidated after a given time. If it takes too long to approve a PR, then we will have to trigger another build and test run on a new integration result. Another solution is to run the build and tests on every commit in the master branch and to roll back the latest commit in the event of a failure. It remains to be seen how this will be set up.

4.2.2 Theory: Available Git workflow options

- Git flow - with the typical practices explained and if we eliminate the master and hotfix branches, it seems like it could meet our needs quite nicely. It may also be practical to rename our develop branch to master for more convenience with default Git tools settings.
- Github flow - maybe add one more quality gate and deploy first to pre-production.
- Gitlab flow - similar to Github flow, plus environment branches (deploy-on-merge).
- Atlassian basic Git flow for continuous delivery - similar to Github flow with rebase during development and explicit merge –no-ff.
- AzureDevOps Microsoft Git flow.
- Trunk-based development with Gerrit.

Considerations and criteria of assessment

There is no one-size-fits-all Git workflow. We want to begin with the simplest workflow to avoid unnecessary complexity and to minimize unnecessary cognitive overhead to the team.

The chosen strategy should provide a clear set of actions to guide the workflow so as to minimize mistakes. It needs to be easy to undo mistakes. We want the workflow to support and reinforce usage of short-lived branches that correspond to tasks in progress. We intend to proactively prevent merges that will have to be reverted by a build and test run (in an ideal situation) on a Pull Request.

Atlassian's article on "Comparing Git workflows" recommends choosing "a Git workflow that is a productivity enhancement for your team. In addition to team culture, a workflow should also complement the business culture. Git features like branches and tags should complement your business's release schedule" (Atlassian, Git Workflow, n.d.). Since we want to be CD-friendly from the very beginning, we need to choose a workflow that supports continuous delivery.

GitFlow

Atlassian: GitFlow workflow assigns very specific roles to different branches and defines how and when they should interact. It uses individual branches for preparing, maintaining, and recording releases. The master branch stores the official release history with tags (an abridged version of the project history), and the develop branch serves as an integration branch for features. Feature branches are pushed to the central repository for backup/collaboration. When a feature is complete, it gets merged back into the develop branch (Atlassian, Git Workflow, n.d.).

In 2010 Vincent Driessen claimed that GitFlow is a “workflow that helps developers keep track of features, hotfixes and releases in bigger software projects” (A successful Git branching model, 2010). Figure 15 gives an overview of the flow:

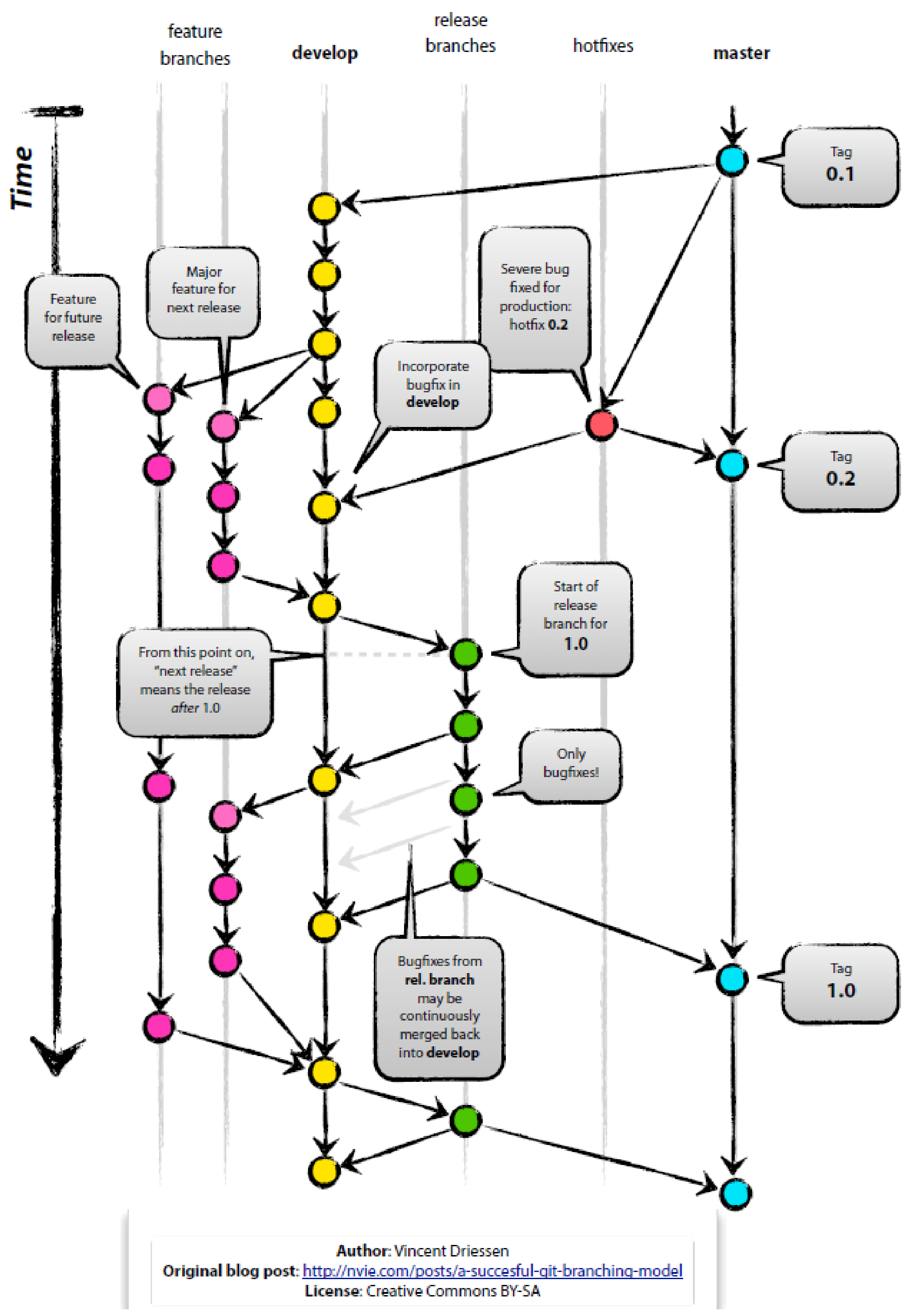

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Severe bug fixed for production: hotfix 0.2
Major feature for next release
Feature for future release
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/posts/a-succesful-git-branching-model
License: Creative Commons BY-SA

Figure 15. Git Flow, by Vincent Driessen. Driessen, 2010) (CC BY-SA 3.0)

Jeff Kreeftmeijer (2018) writes that “Git-flow makes it easy to work on multiple features at the same time by using feature branches” (Kreeftmeijer, 2010).

“It’s complicated enough that a big helper script was developed to help enforce the flow. Though this is cool, the issue is that it cannot be enforced in a Git GUI, only on the command line, so the only people who have to learn the complex workflow really well, because they have to do all the steps manually, are the same people who aren't comfortable with the system enough to use it from the command line. This can be a huge problem" (Chacon, 2011).

Known issues with Git Flow are well described in an article by Scott Chacon "GitHub Flow" in 2011: "If you're not doing versioned releases, Vincent's git workflow and the git-flow library might not be a right fit for you." The git-flow process is designed largely around the "release", but we (the DevApps team at QPR) are moving towards continuous deployment (CD) (Chacon, 2011).

In Git Flow the development happens on the develop branch, moves to a release branch, and is finally merged into the master branch. When the master is used by many tools as a default, the first problem is that developers must use the develop branch and not the master (master is for production), so developers have to switch to another branch. But this can be tweaked in Azure DevOps at least, since the develop branch can be set as the default for making pull requests to.

"The second problem is the complexity introduced by the hotfix and release branches. These branches can be a good idea for some organizations, but are overkill for the vast majority of them. Nowadays, most organizations practice continuous delivery, which means that your default branch can be deployed. Continuous delivery removes the need for hotfix and release branches, including all the ceremony they introduce." We could simplify Git Flow for our needs and simply not use the hotfix and master branches, but rather stick to only one branch per deliverable value increment ("release of a user story") and make bug fixes (hotfixes) directly to it, and set a tag on it if needed.

GitLab summarises the criticism of Git Flow: "Git flow is too complicated for most use cases" (Introduction to GitLab Flow | GitLab, n.d.).

Git Flow explained by Contribyte (in a workshop):

Contribyte recommends using:
1. "feature" or "user-story" branches (same as Git Flow)
2. "develop" or "master" branch (same as Git Flow)
3. "release" + tags + hotfixes all in one short-lived branch - one per "release of a user story". The master is used as a branch for tags and the releases history will not be used in our case. This is a simplified version of Git Flow. Figure 16 illustrates the branches and code changes flow.

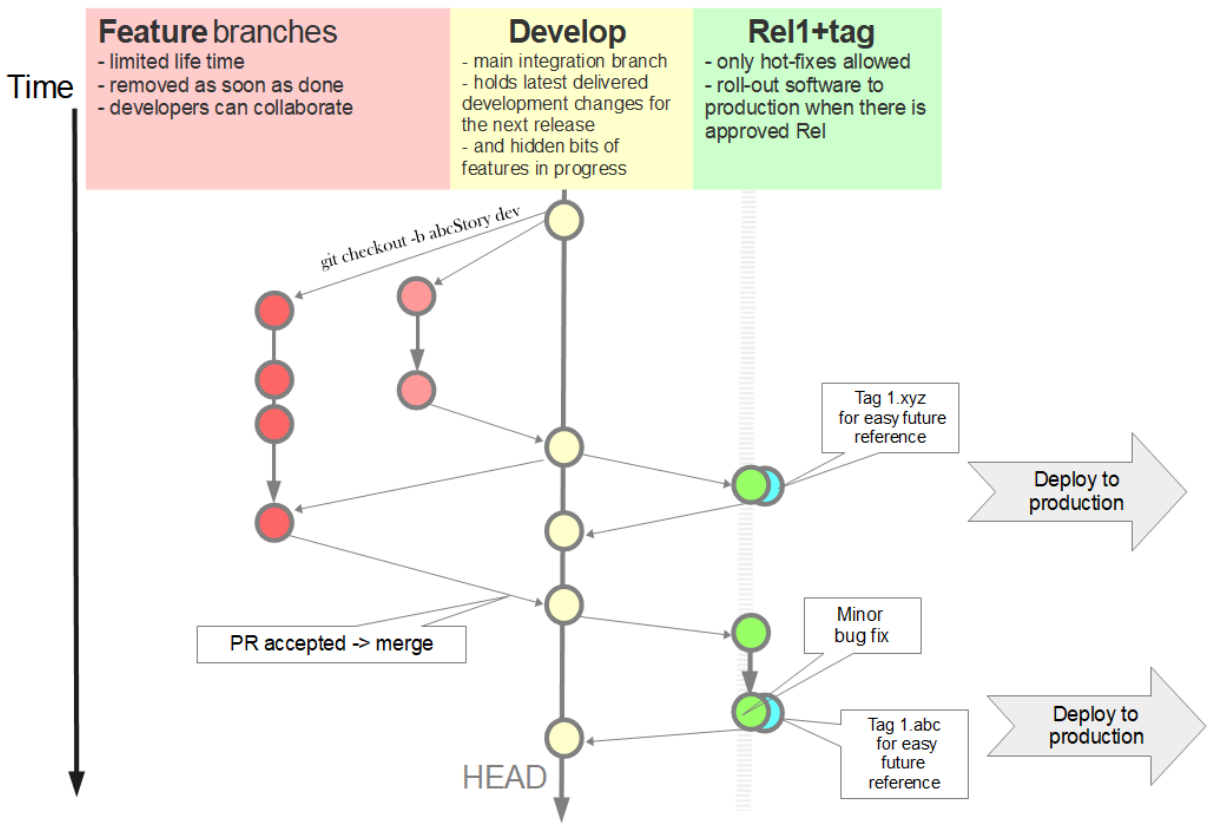

Time
Feature branches
- limited life time
- removed as soon as done
- developers can collaborate
Develop
- main integration branch
- holds latest delivered development changes for the next release
- and hidden bits of features in progress
Rel1+tag
- only hot-fixes allowed
- roll-out software to production when there is approved Rel
git checkout -b abcStory dev
Tag 1.xyz for easy future reference
Deploy to production
PR accepted -> merge
Minor bug fix
Tag 1.abc for easy future reference
Deploy to production
HEAD

Figure 16. Simplified Git flow drafted following the Contribyte workshop

GitHub Flow

“We can use the exact same process to address hotfixes (or any other changes) as we do to handle normal or even large feature development” (Chacon, 2011).

Simple GitHub flow rules:

- **The master branch is stable - anything in the master branch is deployable** at least within hours. If you push something to master that is not tested or breaks the build, you break the social contract of the development team.

- To work on something new, create a descriptively named branch off master.

- **Commit to that branch locally and regularly push your work to the same named branch on the server.** This is also a way to back up your work in case of laptop loss or hard drive failure and a way to have constant communication and transparency.

- When you need feedback or help long before you actually want the branch to be merged, or you think the branch is ready for merging, open a pull request - a kind of branch conversation view.

- After someone else has reviewed and signed off on the feature, you can **merge it into master**.

- Once it is merged and pushed to master, you can and *should* deploy immediately

Additionally, you could have a "deployed" branch that is updated only when you deploy. Figure 17 illustrates the branches and code changes flow.

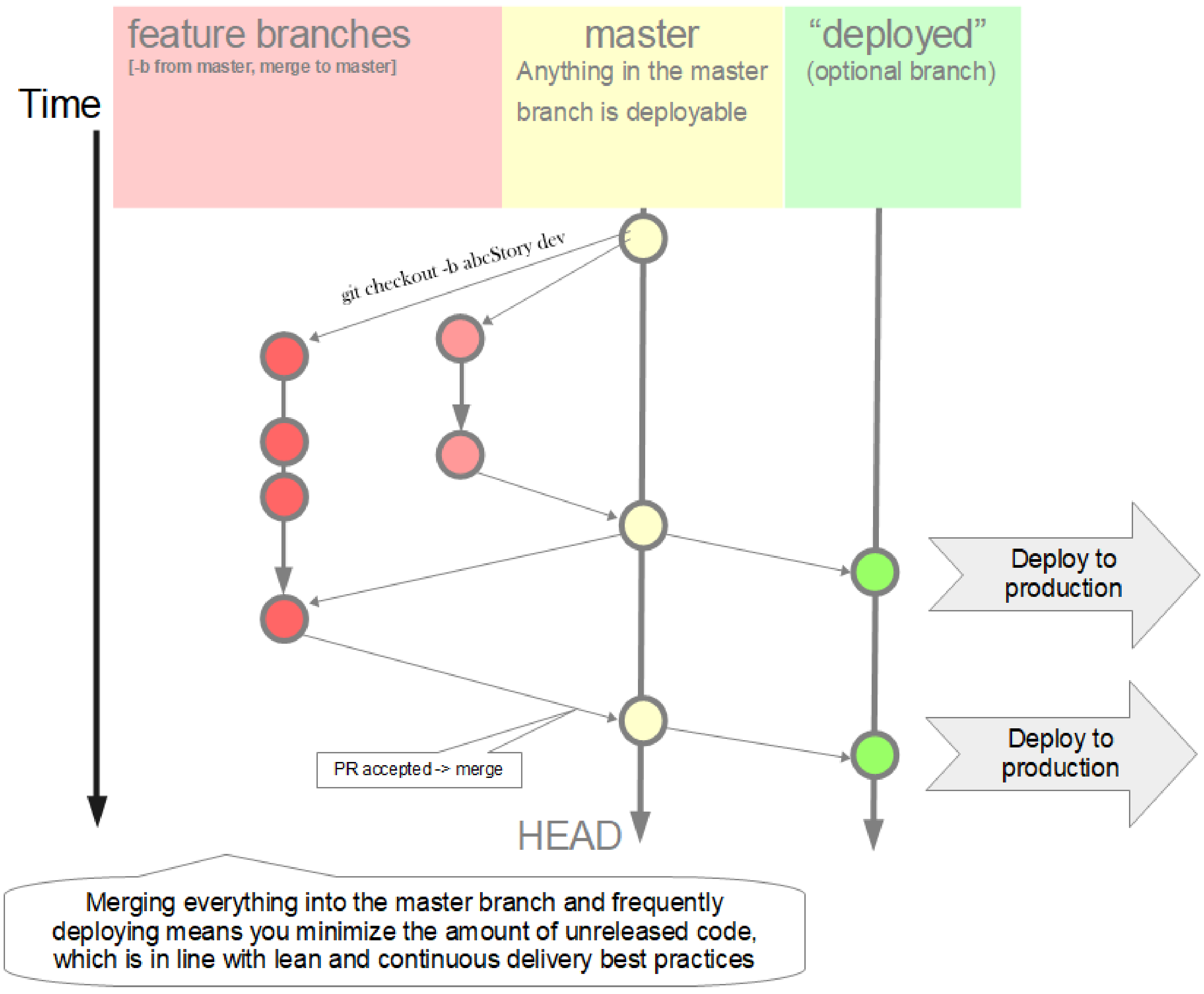

Time
feature branches
[-b from master, merge to master]
master
Anything in the master branch is deployable
"deployed"
(optional branch)
git checkout -b abcStory dev
Deploy to production
Deploy to production
PR accepted -> merge
HEAD
Merging everything into the master branch and frequently deploying means you minimize the amount of unreleased code, which is in line with lean and continuous delivery best practices

Figure 17. Git Hub Flow summary

"GitHub Flow has only feature branches and a master branch. Merging everything into the master branch and frequently deploying means you minimize the amount of unreleased code, which is in line with lean and continuous delivery best practices. However, this flow still leaves a lot of questions unanswered regarding deployments, environments, releases, and integrations with issues." ("Introduction to GitLab Flow | GitLab," n.d.)

“Git itself is fairly complex to understand, making the workflow that you use with it more complex than necessary is simply adding more mental overhead to everybody’s day. I would always advocate using the simplest possible system that will work for your team and doing so until it doesn’t work anymore and then adding complexity only as absolutely needed.” And “for teams that have set up a culture of shipping, who push to production every day, who are constantly testing and deploying, I would advocate picking something simpler like GitHub Flow.” (Chacon, 2011)

For us the goal to always have a deployable master (develop) branch may be too ambitious initially, but will remain the long-term target. Regarding the DevApps team’s flow, in the beginning we will try to do additional pre-production testing on a short-lived release branch, which may include a human check for visual design, some additional security and performance testing, and more independent beta-testing by users outside of the development team.

GitLab flow

GitLab flow combines feature-driven development and feature branches with issue tracking. The assumption is that every time you merge the Feature branch to “master”, you can then deploy. It is possible with SaaS applications, but not if you cannot control the timing of release (it depends on something else, e.g. Marketing department plans). GitHub flow also does not work when you have deployment windows (when deployment cannot happen on every merge to master). In these cases, you can make a production branch that reflects the deployed code. Then to deploy you need to merge the master into the production branch. If you need to cherry-pick a commit with a hotfix, it is common to develop it on a feature branch and merge it into the master with a merge request, and from there to release (if there is a release branch). This is an “upstream first” policy: merge a fix first into the master and then cherry-picking into the release (Introduction to GitLab Flow | GitLab, n.d.).

Compared to GitHub Flow, GitLab adds environments to deploy each of the branches: a merge of the feature branch triggers deployment from the master branch to the staging environment. A merge from the master branch to the production branch triggers deployment to the production environment. Optionally there may also be a pre-production branch and environment. Everything is tested in all environments.

Atlassian basic Git flow for continuous delivery

Nicola Paolucci from Atlassian recommends a strategy similar to that of GitHub Flow, with an explicit merge using the flag –no-ff. (Paolucci, 2014)

It has two guiding principles:

- The master branch is always production-like and deployable.
- Rebase during the feature development, but make an explicit (non fast-forward) merge when done.

When development is complete, record an explicit merge (merge using the flag --no-ff). The merge commit becomes just a marker that stores the contextual information for the feature branch. “This will preserve the context of the work and will make it easy to revert the whole feature if needed. [It is] also useful for future binary search to find the faulty commit (Git bisect)”, suggests Fredrik V. Mørken (Platform Engineer at Spacemaker) in his article “Why you should stop using Git rebase”. (Mørken, 2017).

Azure DevOps Microsoft

Azure DevOps: Git branching guidance recommends:

- Use feature branches for all new features and bug fixes.
- Merge the feature branches into the master branch using pull requests.
- Keep a high quality, up-to-date master branch. The code in your master branch should pass tests, build cleanly, and always be up to date, so that feature branches created by your team start from a known good version of code.

“Other branching workflows use Git tags to mark specific commits as [ready] for release. Tags are useful for marking points in your history as important, but tags introduce extra steps in your workflow that are not necessary if you are using branches for your releases” (Adopt a Git branching strategy, 2018).

Trunk-based development with Gerrit and verifications in Jenkins

A branching model that originates from SVN could be adapted for git. “Where developers collaborate on code in a single branch called trunk (master in the Git nomenclature), resist any pressure to create other long-lived development branches by employing documented techniques. They therefore avoid merge hell, do not break the build, and live happily ever after.”

Trunkbaseddevelopment.com (the site by Paul Hammant, 2017-2018) attempts to collect all the related facts, rationale and techniques for Trunk-Based Development together in one place: Small commits are done straight into the trunk, but before that the build is run (must pass). It may however be extended with short-lived feature branches (maximum of a couple of days) that can be integrated into the mainline via Pull Requests, which involves code review, style check, build and test runs and automated tests. People who practice the GitHub-flow branching model will feel that this is quite similar, but there is one small difference regarding where to release from (Hammant, Paul 2017-2018).

An experimental realization of the flow was set up by P.K. (one of the team members) and here is his description of the flow: “Gerrit is a Git-server plus front-end that supports a “review-before-merge” approach to feature development. Each commit is sent to “refs/for/master” instead of “master”, which will be intercepted by Gerrit and a code review is put in place. This means that the commit will be waiting for approval, while at the same point Jenkins can run code style checks and run also test automation. If something to repair is found, the original developer is informed about this and will be able to fix the issues, and then reintroduce the previous commit for the next review round. When all changes are OK the reviewer can “submit” the changes to the master branch. This also makes it possible to keep the feature “in flight” instead of merging it to the master for any reason, for example, to delay it until some other change is merged.”

The mentioned “review-before-merge” approach for feature development can be realized basically in two ways:
1. The Pull Request model (e.g. Github, Gitlab, Bitbucket, Azure DevOps), where any change is done on a specially created branch that is then requested to be accepted to master.
2. The Gerrit model, where changes are made directly to the local master branch and they have to go through a review system by creating a “Change request” on Gerrit, and be approved before they can land in the master branch on the VCS server.

“The flexibility of Gerrit comes [at] the cost of complexity” claims Marcus Carlsson in Beepsend Tech Blog (Carlsson, 2016)

We (DevApps team at QPR) rejected the Gerrit option mainly because we decided to go with Azure DevOps, which was a more ready-made solution where most of the tools are integrated and easy to set up, so we would not have to employ our internal workforce to set up and maintain Git, Gerrit and Jenkins.

4.2.3 Practical outcome: Initial suggestion for git flow in the DevApps team

Constructing an optimal workflow for DevApps-team@QPR is not something I consider feasible within just a couple of months and then locking down. We came to a very general understanding about the appropriate branching model and workflow that we can begin with, but it still requires detailing and polishing. It still may have to adapt to some new needs that become apparent later.

A common concern among the team seems to be the importance of minimizing the change set (code implemented in a feature branch) that can be integrated into the master (named *develop* in the DevApps setup) and this master must be stable and tests must pass always. Minimal change also implies that it takes very little time to implement and the life span of its branch is short (couple of days). In most of the described workflows, this minimal code change arrives to the master via a Pull (or Merge) Request, which serves as the main gate to ensure the quality of the increment and the stability of code in the master after it is integrated. To have or not have dedicated release branches will be the decision of a specific team and company to decide on, depending on the pace of release and planning. We plan to create short-lived release branches per small value increment for delivery to customers.

The Azure DevOps Git branching guidance recommendations are comprehensive and work well for us. The Atlassian basic Git flow for continuous delivery is very close to our current needs, but probably the explicit merge (--no-ff) when integrating a completed feature into the main branch is not currently essential. Instead of merge (--no-ff), in most cases we would squash all commits from a feature branch and merge it into the develop branch, so we set up only one commit to the develop branch. Or we rebase the source branch commits into develop and fast-forward.

It seems reasonable to select the simplest workflow in the beginning, and I expect it to evolve with time and as needs are clarified or change. Here is the description of the main principles of the DevApps Git flow:

1) One **stable main development branch –** though named *develop* rather than the more conventional master, for the integration of completed features, which is continuously deployed and tested, the source code of HEAD always reflects a state with the latest delivered development changes for the next release.
GitLab docs: this is “in line with lean and continuous delivery best practices.”
2) Each developer does their work in their own (short-lived!) **feature branch** cloned from the develop branch and makes a Pull Request back to the develop branch. The feature branch is removed as soon as it is merged successfully into develop or may live until it is not needed anymore.
A general note about best practices from the GitLab docs: “Commit to feature (story or topic) branch locally and regularly push your work to the same named branch on the server. This is a way to

backup your work, a way to get feedback or help long before you actually want the branch to be merged. And use Pull Request as a great code review system, not just a way to promote your change to master. You can use PR to say “I need help or review on this” in addition to “Please merge this in”. If the branch has been open for too long and you feel it’s getting out of sync with the master branch, you can rebase on master and keep going.”

The GitLab Docs discuss the length of life for a feature branch: “Most feature branches should take less than one day of work. If your feature branches often take more than a day of work, try to split your features into smaller units of work” (“Introduction to GitLab Flow | GitLab,” n.d.). - we will keep that in mind and aim to have short-lived fast-merging branches. And “Commit often and push frequently: Every time you have a working set of tests and code, you should make a commit. Splitting up work into individual commits provides a context for developers looking at your code later. Smaller commits make it clear how a feature was developed, and they make it easy to roll back to a specific good point in time or to revert one code change without reverting several unrelated changes. Committing often also makes it easy to share your work, which is important so that everyone is aware of what you are working on. You should push your feature branch frequently, even when it is not yet ready for review. By sharing your work in a feature branch or a merge request, you prevent your team members from duplicating work. Sharing your work before it’s complete also allows for discussion and feedback about the changes, which can help improve the code before it gets to review” (Introduction to GitLab Flow | GitLab, n.d.).

3) The DevApps team are more likely to use one branch per released increment (user story). These **release branches** may be branched off from develop on a just-in-time basis and may receive bug fixes, but adding features there would be strictly prohibited. Theoretically, we could use a script to automatically build and roll-out our software to our production servers when the release criteria are met. But it remains to be seen what the release criteria will be and whether it will be possible to automate checks of these or will there still be some manual step or collective decision to “push the button and roll-out” an update.

“After announcing a release branch, only add serious bug fixes to the branch. If possible, first merge these bug fixes into master, and then cherry-pick them into the release branch. If you start by merging into the release branch, you might forget to cherry-pick them into master, and then you’d encounter the same bug in subsequent releases. Merging into master and then cherry-picking into release is called an ‘upstream first’ policy, which is also practiced by Google and Red Hat” (“Introduction to GitLab Flow | GitLab,” n.d.).

How to make fixes to the release branch is still an open question - first to merge fixes into develop and then merge to release X, or first to release X and then cherry-pick back to the develop branch? Scott

Chacon recommends using “the exact same process to address hotfixes (or any other changes) as done to handle normal or even large feature development” (Chacon, 2011). We have the possibility to add something to this process later if needed.

Issue tracking: “Any significant change to the code should start with an issue that describes the goal. Having a reason for every code change helps to inform the rest of the team and to keep the scope of a feature branch small. If there is no issue yet, create the issue, [just] as long as the change will take a significant amount of work, i.e., more than 1 hour” (“Introduction to GitLab Flow | GitLab,” n.d.).

We will connect code changes with work items in Azure DevOps via referencing the work item in the Pull Request, a policy that can be forced via the settings.

It is totally acceptable to create Pull Requests when work is still in progress for collaboration purposes, for discussion, and for early reviews; this kind of early PR can be created as a “draft” Pull Request in Azure DevOps, and later converted to a normal PR that can be approved and merged in the normal way.

To enforce coding standards, we intend to use linting in Visual Studio Code, branch policies in Azure DevOps, and hopefully other static code analysis tools later.

We also aim to block direct commits to the develop branch using the branch policies in Azure DevOps.

All that is summarized in the following Figure 18:

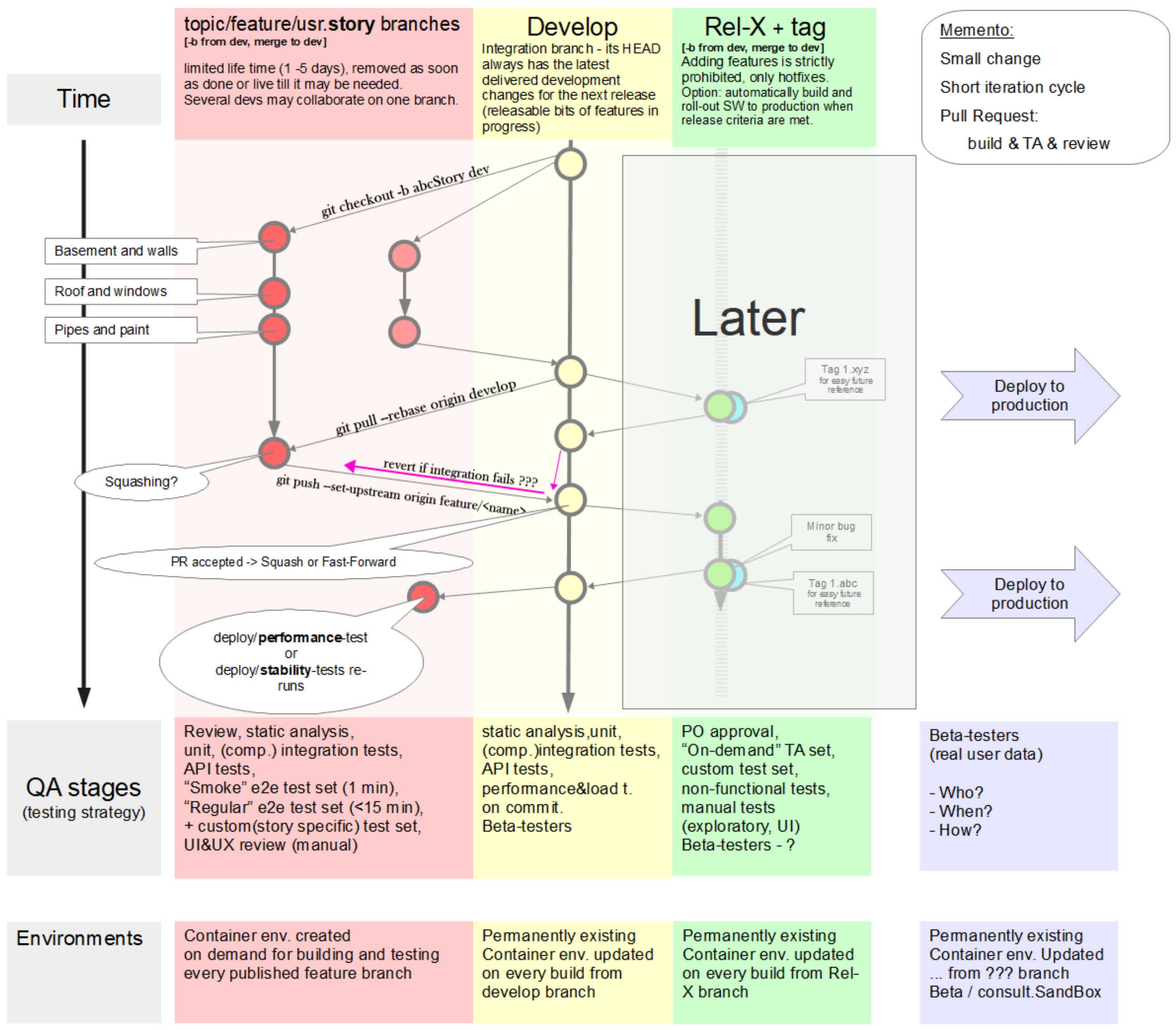
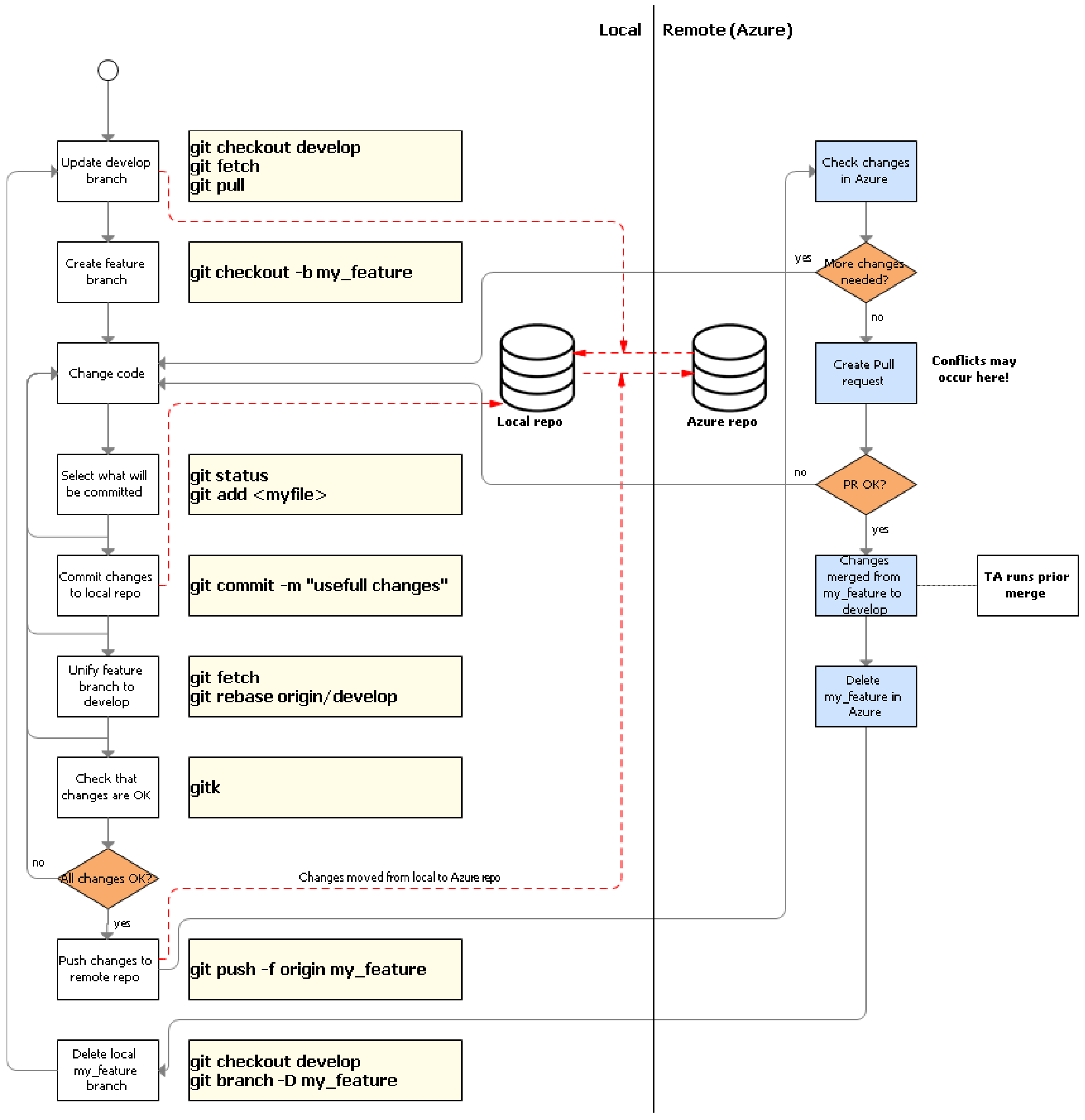

Time
topic/feature/usr.story branches
[-b from dev, merge to dev]
limited life time (1 -5 days), removed as soon as done or live till it may be needed. Several devs may collaborate on one branch.
Develop
Integration branch - its HEAD always has the latest delivered development changes for the next release (releasable bits of features in progress)
Rel-X + tag
[-b from dev, merge to dev]
Adding features is strictly prohibited, only hotfixes. Option: automatically build and roll-out SW to production when release criteria are met.
Memento:
Small change
Short iteration cycle
Pull Request:
build & TA & review
git checkout -b abcStory dev
Basement and walls
Roof and windows
Pipes and paint
Later
Tag 1.xyz for easy future reference
Deploy to production
git pull --rebase origin develop
Squashing?
revert if integration fails ???
git push --set-upstream origin feature/<name>
Minor bug fix
PR accepted -> Squash or Fast-Forward
Tag 1.abc for easy future reference
Deploy to production
deploy/performance-test or deploy/stability-tests re-runs
QA stages (testing strategy)
Review, static analysis, unit, (comp.) integration tests, API tests, "Smoke" e2e test set (1 min), "Regular" e2e test set (<15 min), + custom(story specific) test set, UI&UX review (manual)
static analysis,unit, (comp.)integration tests, API tests, performance&load t. on commit. Beta-testers
PO approval, "On-demand" TA set, custom test set, non-functional tests, manual tests (exploratory, UI) Beta-testers - ?
Beta-testers (real user data)
- Who?
- When?
- How?
Environments
Container env. created on demand for building and testing every published feature branch
Permanently existing Container env. updated on every build from develop branch
Permanently existing Container env. updated on every build from Rel-X branch
Permanently existing Container env. Updated ... from ??? branch Beta / consult.SandBox

Figure 18. Simplified Gitflow strategy proposal for DevApps Team@QPR

4.2.4 Step-by-step Git flow HowTo

A minimalist description and the steps of DevApps team Git workflow can be found in Appendix 7

Git workflow diagram made by A.M. (one of the team members) Figure 19:



Local
Remote (Azure)
Update develop branch
git checkout develop
git fetch
git pull
Create feature branch
git checkout -b my_feature
Change code
Select what will be committed
git status
git add <myfile>
Commit changes to local repo
git commit -m "usefull changes"
Unify feature branch to develop
git fetch
git rebase origin/develop
Check that changes are OK
gitk
no
All changes OK?
yes
Changes moved from local to Azure repo
Push changes to remote repo
git push -f origin my_feature
Delete local my_feature branch
git checkout develop
git branch -D my_feature
Local repo
Azure repo
Check changes in Azure
yes
More changes needed?
no
Create Pull request
Conflicts may occur here!
no
PR OK?
yes
Changes merged from my_feature to develop
TA runs prior merge
Delete my_feature in Azure

Figure 19. Git workflow diagram. © QPR Software Plc, used with permission

4.3 Retrospective

For more details see Appendix 5.

4.3.1 Retrospective comments analysis 9.09 - 27.09.2019

This retrospective covered the period from 9.09 to 27.09.2019 - 3rd sprint.

One more topic theme was added: Minimize the sizes of the value increment.

The following themes were reflected on by the team as positive in the 3rd sprint (Glad):

Agility of process - process is settling down, "Creative chaos", a bit chaotic, but things get done. There’s a more “free” feeling. Pull Requests get reviewed quite fast.

Cooperation, Collaboration - good teamwork, good communication about what needs to be done. Sprint review was inspiring, had nice conversations with all developers.

Feeling of accomplishment (achieving) - delivered first App pages, we achieved results as a team, we have first App running, I was able to contribute. Progress is good!

Usability of documentation and chats - Wiki is growing, Wiki is easy to edit

Some comments on what made team members glad after the 3rd sprint:

- “I was able to contribute to the team's progress.”

- “Even being distributed geographically, I had nice conversations with all the developers.”

- “Despite the sprint looking a bit chaotic, things get done and I feel there’s good communication about what needs to be done.”

- “Focus on critical tasks – minimal scope to deliver helps to keep the focus and stay on track.”

Points for improvement, worries or concerns mentioned (Sad&Mad):

Freedom versus stability or certainty: unclear who is going to do what, I don't know what to do, undefined user stories were picked for sprint, not enough initiative taken in sprint refinements.

Agility of process: A lot of planning, but not much to do. Not enough transparency, no descriptions in tasks. Documentation may be forgotten.

Cooperation & Collaboration: Related tasks are rather done in isolation. Branching strategy and workflow is not yet agreed.

Cooperation & Collaboration (expertise silo vs. sharing knowledge): Boring and annoying to configure. No TA and CI&CD environments, we rely too much on a single person, setting up TA&CD pipeline hasn’t progressed, TA is missing, changes coming to develop without tests, No environments to deploy, No CD yet,

Minimize value increment: The subjects for review may be complex, some PR have too many changes, one user story has grown too big, which led to unplanned work being done.

Mad:

unclear who will do what; tasks were too big.

Some comments on what made a team member sad or mad after the 3rd sprint:

- “I didn't have a clear development task for the sprint. A lot of planning, but not much to do. “
- “Not enough initiative in refinements.”
- “Homescreen user story has grown too big and ended up including other (unplanned) user stories.”

4.3.2 Agreed adjustments

User stories should be split into smaller user stories.

Organize pre-planning among developers, and groom sometimes without PO.

Work in pairs to share expertise on server, client part, tests, CI&CD infrastructure. Have walkthroughs before making a PR.

Establish small TA&CD group to share competence and get the pipeline implemented.

Add description in the task and some minimal necessary info about progress (daily), so it is possible to read and pick it up to continue the work if needed.

Try to communicate more and consider every small task as a part of the whole from the first stages of the work.

5 Iteration 3: October 2019

5.1 Agree on common terms to speak a common language (9.10.2019)

When a team is trying new tools and refining working practices, all team members need to be on the same page regarding terms and definitions if daily communications are to be understandable. To this end, I made a small questionnaire to elicit how people see the core terms in our work process and where is there might be a mismatch. I did not use any systematic or scientific approach to listing the terms: I first listed those that came to mind and invited team-mates to add if they wanted to. I asked people not to search for definitions via Google, since I wanted to identity what was in our own minds. Here are the opening remarks from the questionnaire:

"*Here are some words…*
What you think when you see those words?
Is it a familiar term?
How do you perceive these concepts?
Give a free definition in your own words.
Which terms are missing from my list that we need to define?"

There follow some brief highlights from the responses received. A full list of explanations and definitions that were given by the DevApps team can be found in Appendix 1.

Work items: **Epic, Feature, User story, Task** - There was some discussion in responses about the purpose served by each of the work items, where code changes belong (to features, user stories or tasks), and how to manage them. The division between a software development task (which includes product code change) and any other tasks (design, documentation, prototyping, testing etc) in my opinion seems harmful to a shared team responsibility in meeting the Definition of Done, so discussions on this issue will need to continue. A work item is needed to bring value to the user in the end, so it doesn't matter so much what is changed: code, UI design, or documentation etc.

Feature branch or User Story branch - the naming doesn't make any difference to us right now. Currently we use the "feature/" prefix for the git branch that contributes to a User Story implementation, "fix/" for a bug fix, and "spike/" for experiments. Feature branches are always cloned from the latest develop branch, though it is important that any branch can be accepted into develop if it has been approved by reviewers and passed verifications without failures. The user story may receive code changes via several branches and several pull requests, so the smaller and the shorter-lived the branch is, the better, since it allows new code to enter a common branch for others to reuse and also minimizes the possibility of merge conflicts.

We had an idea that we would implement a single user story in a single branch, so the user story would be completed via one Pull Request, but currently, user stories are not small enough, so this approach would take too much time.

Branching strategy - guidelines on how to do concurrent development, CI and deployment of the product by utilizing the branching capabilities of the version control system. Questions are centred on the workflow and which branches to use, how to create and when to delete them, and the reasons to use certain branches etc.

Commit - a coherent, atomic piece of implementation checked into the version control system, which has a clear, singular purpose. Naturally, for developers changing from SVN to Git, there was a mix up between the SVN commit which is always pushed to the server and the possibility to make a local commit using Git.

Main repository - our central Git repository in the Azure server keeps the public branch copies of develop, feature, and release.

Mainline - a branch or repository, where all of the approved changes are put. When creating a new branch, take the latest version from mainline. Or branch where development happens. This is where branches with finished features are not yet published. The branch on which features are branched from and merged to. Where changes are integrated for testing.

Codebase - the "sum" of all branches in the main repository. Or a set of all code related to some product (including tests etc). Codebase does not include third party frameworks and libraries.

Pull Request - a request to review and accept code into some branch. One comment points out that a “the call for a review before a change can be pushed to some codebase -> term "pull" here is confusing” - shows that we need to talk about the difference in approach when code is pushed and when it is pulled.

Continuous deployment - A software engineering model where product increments are deployed as soon as they meet the team's Definition of Done, or acceptance criteria, quality criteria etc. This may go through several staging phases to enable e.g. manual testing. The staging phase may take several days from the initial commit / pull request merge. Or is it a fully automated process to update the production environment? This needs to be checked.

These terms or concepts were not on my list, but we were discussed over the course of the autumn:
Increment,
Change-set,
Architecture,

Specification,
Plan,
Functional specification,
Technical specification,
Exploratory testing,
Test design,
Bug,
Non-Functional Requirements,
Code Reviews,
Definition of Done,
Definition of Ready.

The latter terms discussed over the autumn can be seen to be less about definitions and more about practices. Nevertheless it is good to understand and clarify what actions are meant when those terms are used.

5.2 Scrum

5.2.1 Daily cafe, improving daily stand-ups

With a view to reducing time wasted, breaking up routines and encouraging meaningful discussion in a distributed team, we started to use the Daily Cafe format, though it was not used daily, but rather a couple of times a week. It is described by the Lean Coffee website as “a structured, but agenda-less meeting. Participants gather, build an agenda, and begin talking. Conversations are directed and productive because the agenda for the meeting was democratically generated” (Lean Coffee | Start one in your city!, n.d.).

5.3 Define metrics for the SW development team

5.3.1 Theory: Goal - Question - Metric framework

“You get what you measure” - Human beings adjust behavior based on the metrics they’re held against, asserts Dan Ariely in his column “You Are What You Measure” (Ariely, 2010). Keeping that in mind, we will have to evaluate very carefully what to measure.

Another consideration is the limit beyond which there are too many metrics. “Companies rarely suffer from having too few measures. More commonly, they keep adding new measures whenever an employee or a consultant makes a worthwhile suggestion, but it is better to focus on the handful of measures that are most critical” (Kaplan and Norton, 1992).

It’s an old cliché: “What gets measured gets done.” And “If you can measure it, you can manage it.” meaning that measuring something gives you the information you need in order to make sure you achieve what you set out to do. For many people, the simple act of measurement increases motivation to perform because it is a way to determine whether you have won (Henderson, 2015). A measure is a quantitative number that counts for something. A metric compares the measure to some other baseline. The motivational metrics and measures create eustress and increase performance.

“Measure something that needs an action attached to it. The goals and metrics you are measuring have to align with goals” (“What Gets Measured Gets Done - iSixSigma, 2012). An example of a goal-oriented (top-down) software measurement is the Goal Question Metric (GQM) approach (Basili, 1993). GQM is a way to derive and select metrics for a particular task in a top-down and goal-oriented fashion. It was developed by Basili and Weiss during the 1980s and later extended by Rombach. Stating goals in advance leads to a selection of only those metrics that are relevant for achieving these goals. The GQM approach was originally used to improve software products and development processes (Eusgeld et al., 2008 p.39).

A goal has a purpose (e.g. to improve), implies an object under study (e.g. a software development process), a specific issue or quality attribute (e.g. developers’ engagement, initiative, satisfaction, quality of product, usability of tools) and a perspective (e. g. developers, business stakeholders). Questions refine the goal. By answering questions, it should be possible to conclude whether the goal has been reached. Measurements provide answers to quantitative questions, and questionnaires can be used to address qualitative questions. It is good to have a hypothesis and to verify whether the results match with the hypothesis and then to learn from this comparison. Several metrics can be defined for each question.

We will choose the metrics that are most critical at this point in the change process, with a view to modifying or adding later as necessary. It would seem to be useful to set up some motivational metrics to help achieve goals. Moreover, it is important to ensure our metrics do align with our goals.

5.3.2 Previously used metrics

These metrics were used, or some are still used with other product development teams (not the new experimental DevApps team).

Work distribution:

Split effort between new features development and defect fixing.
Work distribution by product (main focus on new product, minimal support to other)
Work distribution by work type.

Test automation:

CI cycle time (to run build and automated tests)
Failed case trend,
Status overview
Release failed tests trend

Scrum:

Sprint burndown (not monitored lately)
Sprint forecast accuracy (not monitored lately)
Sprint backlog status.

Defects:

Count of unidentified defects,
Defect severities,
Number of known defects

5.3.3 Practical application: DevApps team metrics - an initial proposal

To come up with some initial metrics I used the Goal/Question/Metric paradigm. Goals should be measurable, so measurement should be based on a goal. “Measurement is a mechanism for creating a **corporate memory** and an aid in **answering** a variety of **questions** associated with any software development” (Solingen and Berghout, 1999).

How it was done:

1. Develop a set of goals. Goal is defined by:

purpose (what object and why),
perspective (what aspect and who), and
environmental characteristics (where, context).

2. Generate questions that define those goals as completely as possible in a quantifiable way.

3. Specify measures to collect data that answer those questions.

4. Develop mechanisms for data collection.

5. Collect data, analyse data and provide feedback to the project on how to make future improvements and adjustments.

See Appendix 6 for the full Goal-Question-Metric matrix.

Goal: To state the success of the iteration (sprint) as a fact and for the development team to feel satisfaction.

Question: Do we meet sprint goals in the forecasted time?

Created Metric: Listed sprint goals that have been planned and accomplished in a table (in AzureDevOps/dashboard/QPRUI 2.0/ Sprint goals planned vs accomplished).

Question: Are we going to reach the MVP goal (release) by the end of this year (2019)?

Created Metric: Minimal Viable Product release Burndown chart (in AzureDevOps/dashboard/QPRUI 2.0 / MVP Burndown)

Goal: To improve estimates of Features and User Stories, allow the development team to measure its own velocity and be able to forecast the outcomes of iterations, then use this velocity to limit planned work and deliver it. We want to learn how to split features and user stories into smaller releasable value increments and deliver the increments more often.

Question: How many story points do we use on average in one sprint? We can count the average velocity retrospectively for autumn 2019, but to see the trend we need a more established development process, which is possible only after the “foundation stone” of the app and framework is laid down.

Created Metric: Number of story points completed in each sprint (=velocity). The team’s velocity is recorded in the AzureDevOps/dashboard/QPRUI 2.0 / DevApps. I expect an oscillation in the velocity from sprint to sprint, but at some point, we would expect to settle into a relatively steady number of points per sprint, which would constitute a predictable production output for the team. Another option would be to count the average velocity retrospectively for autumn 2019, though to see the trend we need a more established development process, which is possible only after the “foundation stone” of the app and framework is laid down.

Question: Do we remember to spend enough time on backlog refinement, so we would have enough user stories ready for implementation? Currently tasks under user stories are not defined in sufficient detail, planning sessions are not efficient, backlog refinement is done mostly in meetings, which is not the best use of time.

Created Metric: Measure time used for backlog refinement by the team per sprint (should be around 10% of the development Team's capacity).

Goal: Increase developers' subjective satisfaction with tools and development processes – focus on the developer's experience. We want an effortless and painless development workflow through the tools pipeline. Tools are needed that support and guide the developer (engineer, human) to the correct workflow.

Question: What is the “mood trend” in the team?

Created Metric: Take a subjective “happiness measure” from the sprint retrospective summaries. Count the comments in “mad-sad-glad" and see if there is any trend in the number of positive vs negative comments.

Goal: Improve the efficiency (maximum result with minimal expense) of the code development and promotions (from development to production). To be efficient in bringing value to users. To “boost development throughput”.

Question: How difficult and intimidating are the merge conflicts? How often do they take (subjectively) significant time? Previously at QPR, merge conflicts have been among the most critical problems delaying code promotions.

Metric: There may be several issues creating difficulty or meeting a target for merging a branch to develop per day/week, but currently it seems too much hassle to collect data on these issues, so we rely on a subjective impression coming from the sprint retrospective, as well as daily discussions.

5.4 Retrospective

For more details about meeting and questions see Appendix 5.

5.4.1 Retrospective comments analysis for the period 30.09-18.10.2019

This retrospective covered the period 30.09 to 18.10.2019 - 4th sprint.

The following themes were reflected on by the team as positive in the 4th sprint (Glad):

Feeling of accomplishment (achieving): getting closer to accomplishment, progress (3 comments), practical attitude, good progress, glad that we have some nice-looking views, current sprint's user story is very near to completion, (almost) all the tasks from the backlog have been done.

Cooperation & Collaboration: Good teamwork, spirit, co-working going well. Happy to get feedback.

Meeting efficiency: Retrospectives are better than in previous experience.

Agility of process: Pull Requests are fast to review and to complete. Review is thorough.

Tools satisfaction: Familiarization with new environment (Azure DevOps) and tools is going well.

Freedom vs stability/certainty: Free to do small improvements, refactoring. Flexibility.

Some comments on what made team members glad after the 4th sprint:

- "Progress in a creative mess."
- "Pull Requests are reviewed & completed fast."
- "I'm not the only one who is doing TA."
- "Collaboration: asking, criticizing, discussing and coming to agreement"
- "Now it is more clear which tasks to do than before (because of pre-planning)"
- "Development team takes responsibility and makes own decisions about what is best for the application and development progress."

Points for improvement, worries or concerns mentioned (Sad&Mad):

Common goals: We don't do planned tasks – no focus or is it bad planning? There were some distractions from outside the team. Some feature and task creep, so maybe we should focus a bit more on planned tasks.

Meeting efficiency: Planning sessions are not efficient; refinement is done mostly in meetings. Meetings could be improved. Probably too many meetings.

Freedom vs stability/certainty: Responsibilities are not clear and not fixed. I didn't have a programming task to do.

Agility of process: E2E test automation took more time than expected. It seems difficult to get the test design, test code and infra-stuff discussed and reviewed. There are some ready-made tasks that are there just to exist, "deploy to..." as a result, actual tasks get lost in the crowd. No estimations, hence no learning, the sprint board is not up to date, some tasks cannot be assigned – unclear purpose, what and how to do. Tasks under user stories are not sufficiently detailed.

Feeling of accomplishment (achieving): Incomplete things piling up. No CI pipeline, so DoD can't be fully met, no deployment environments to test a PR. Testing is lagging. Some core questions are still open.

Cooperation & Collaboration: Communication within the team may be improved: team members are working disconnected from each other.

Tools satisfaction: Pull Requests for merging conflicts and tests are failing.

Cooperation & Collaboration: Different programming styles - need to agree on a common one. We have different opinions and can't agree on technologies.

Unclassified: Difficult to choose work items for the Pull Request.

<u>Some comments on what made a team member sad or mad after the 4th sprint:</u>

- “No estimations for user stories and DoD, hence no learning “

- “Processes: tasks under user stories not defined in sufficient detail (better groom and pre-plan). Planning sessions are not efficient, grooming is done mostly in meetings, which is not the best use of time”

- “No single User Story DoD is fully met yet.”

- “Incomplete things are piling up”

- “Too much to do at once and everything is needed "yesterday" - load and pressure.”

- “Unprioritized/unfocused TA – spending hours discovering why TA fails on login/logout, TA is unstable, where a manual test is very simple. Login is done daily by everyone anyway. Why do we do that?”

Agreed:

Make a developer’s experience questionnaire before the next Retro.

Remind to update the board daily, so tasks would reflect actual status and progress.

Focus on improving the backlog refinement practices to get the backlog in order.

Discuss and review test design, test code, and infra-stuff.

Make meetings more efficient by creating an agenda, including individual preparation work, time-boxing, and ensuring there is focus.

Pay attention to communication, make sure that the two geographical sites do not become disconnected - keep working in sync.

5.4.2 Retrospective comments analysis 21.10-8.11.2019

Prior to this retrospective, I crafted a questionnaire based on the themes echoed in the previous retrospectives to collect feedback beforehand and to save time in the face-to-face retrospective meeting, so we could jump right to the discussion. I asked my teammates to classify their responses as "mad", "sad" or "glad" to preserve the previously used framework. The list of topics to consider was sent, with the instruction that they are free to skip any topic and treat the open questions more as a seeding for open reflection. I also acknowledged that any topic may give rise to positive and negative comments.

Another positive outcome from creating the questionnaire before the retrospective meeting was that people had some time to think about the topics and when they came to the meeting they had those thoughts fresh in their minds and our discussion was very fruitful, active, and focused on the most important things, according to my own impressions. Doing appropriate “homework” before the meeting made it more efficient.

Previously identified themes:

From the Team Manifesto	From the retrospectives
Fun & personal connection, Respect Cooperation, Collaboration (transparency) Feedback Common goals Trust (not mentioned in retrospective comments) Innovation & Experimentation UX & Customer (satisfaction, need) focused Expectations - set and met I like it! (developer’s perspective) Commitment to Effort, we do our best	Agility of process Feeling of accomplishment (achieving) Usability of documentation and chats Involvement, ownership Freedom vs stability/certainty Meeting efficiency Tools satisfaction Minimize value increment Added: Long-term planning

Many themes elicited contradictory comments and I see that as understandable, since the team and the new product they are developing are still embryonic and people will continue to learn, and we will continue to refine the working practices.

From the responses received, I noticed some fatigue with setting up the development infrastructure and process and I hope that the team could get up to speed with the product development soon. I'm sensing that some people focus on plain software development tasks, but not on DoD, which includes documentation, quality control tasks, and requirements management. That means we need to challenge ourselves more over DoD, discuss what is our "product" and why we have the DoD. Is our product just an application code? Do we want to have a mechanism to receive feedback about quality, which may help catch bugs earlier and cut the costs on fixing them? Are we building a framework that is easy for others to use? What makes it easy to use? In answering those questions, we can come to a common agreement about what is needed and whether we can drop some criteria in the DoD.

Achieved improvements:

Backlog refinement improved - as soon as we put more time and effort into refinement we were able to complete the sprint planning in 30 minutes, though 2 hours was reserved for that.

Pipeline to “build-test-deploy” now works only for the develop branch, so adding the possibility to do all verifications on any development branch before integrating it into develop is the next step. Daily cafes proved a beneficial practice; there were many positive comments in the responses, for example: “Daily cafe seems to work nice, allowing us to discuss something important right there and then.” E2E tests, communication and transparency of tasks-in-progress got better according to the team.

One of the comments was: “Inspiration and enthusiasm [on the] positive side. We just need to get to the point where our actions are more visible to business, easily reachable and start to discuss continuously (Business with Products&Technology) how to improve, what are the priorities, use cases in practice ... also developers need to get into this cycle”

Things requiring further attention:

Completing user stories according to the DoD and taking care of leftovers. There was the “Mad” comment: “DoD not met in any of the user stories. I use a lot of energy to make mine self-organized as a process work.” The first user story was completed just before the retrospective, another the week after, with three stories still in progress.

Development workflow:

Get the development workflow into better shape – start with planning tasks, storybook components, testing. Unit testing should go hand-in-hand with the task that implements the views and view models.

We need more debate and disagreements - different opinions and points of view are especially beneficial.

Fun & personal connection (atmosphere, team spirit) - all good, but there is concern about “an Oulu/Helsinki split developing. Nothing too concerning though... spirits still seem high.”

Inspiration and enthusiasm – mostly positive. Only one comment “I’m feeling quite inspired and enthused still.” – that “still” makes me think there is a decline coming, but this would be natural until some limit is reached, since people cannot exist in a constant state of stress, even if it is a positive excitement.

Innovation & Experimentation (Learning possibilities and curiosity) – mostly positive comments.

Cooperation & Collaboration (Communication and Transparency):

“Sad that sometimes it's hard to get answers in public [i.e. group] chats, and I'm inclined towards having more public discussions to share information and towards involving more people in discussion.”

The general perception of transparency is good according to other comments. However, this one I wanted to highlight as particularly perceptive: “We (as a team and as individuals) are not the only stakeholders in our tasks. Our ways of working need to change to reflect this. Just because I/We (as individuals) don’t get tremendous value from an activity personally doesn’t mean that the activity isn’t worthwhile.”

Long-term planning:

Think and plan together with customers about the longer-term development roadmap, the possible customizations of the application, who will be the first real customers and what kind of solutions will they want etc.

Common goals in the operational development (OD):

Involvement of all team members in the operational development (OD) tasks requires more initiative from everyone. This will build competence and know-how and involve developers in the OD work. Working in pairs is useful to share knowledge and avoid silos.

One of the comments: “Guide to myself: consider all actions from this perspective:
How does what I am currently doing...

- add value to customer?
- make our product better, more attractive, more useful, ...?
- if this is beneficial, what is the cost-benefit ratio?”

Common goals in balancing doing things and documenting or communicating things:

“Too much overhead. “

“I should document more probably. For documentation we should clarify the expected value of the different documentation tasks... might help with motivation to complete them.”

These two contradictory comments make me think that for any documentation tasks, we need to put the focus on the purpose and to consider who will likely use that documentation. Think about other roles in the team and outside of the team, what they would be searching for in the documentation.

Agility of process (a balance between the defined process versus the need to self-organize and make many decisions daily (creative chaos&freedom versus stability&certainty):

Some comments gave the impression that some people would prefer to do more or could do more if they were assigned more tasks, or tasks would be distributed. Can it be solved by the practice “if in doubt, ask the Scrum master”? Or maybe a rule “pick the next task from the sprint backlog”?

Some selected comments:

“Already seeing benefits from changed ways of working. Better shared understanding.”

“Currently, I enjoy the "creative chaos" and I feel it helps me in my work.”

“Creative chaos is calming down a little bit at a time. It is becoming clearer all the time what should be done and how.”

“I wish we could preserve flexibility, and at the same time be more organized in some areas.”

“Too much overhead still, mostly in process, planning, grooming, defining Azure tasks...

“Rushing and fast and dirty, delivered fast !== agile”

Minimize value increment:

The splitting of user stories and trying to have the smallest possible stories still needs a lot of practice. “Splitting user stories feels cumbersome. Wouldn't want to do that in multiple iterations. There is much overhead to create a new user story and copying stuff and editing the original one.” - this comment shows that we need to talk more about why we need to split and how to do it in the most efficient way. If a story is split in the early identification stage, then it should not be such a burden.

Couple of other comments: “Refinement improved, but there's room for more improvement.” “Tasks are not defined well enough. Tasks are too big. Many things are done that are not related to any task. The software development process is slow, productivity is low, total value of the increments are tiny, quality is low.” Backlog refinement in pairs turned out to be a useful practice, so we will continue it and bring questions and the present user story into the refinement meeting.

Scrum practices:

“IMO we should drop time-boxing and instead focus on priorities if we are moving towards Continuous delivery and Kanban. The Kanban philosophy proposes making releases when there is a customer need, while Continuous Delivery suggest releasing per user story/value. Scrum has nice background philosophies, such as for backlog management and basic "controls".”

“Not all tasks were identified before the story was allocated to the sprint for development, some tasks continue to appear under that user story, which is natural and good, but better to have the majority of tasks defined earlier, during the refinement phase.”

Setting sprint goals and achieving them:

It was mentioned several times that it is good to set a clear “critical path” and that less "critical" things can be worked on with less focus. Even if we define some minimum goals, we cannot meet them 100% of the time, either the goal is too general and includes too much to be done, or the focus somehow goes away from DoD.

Feeling of accomplishment (achieving)

The perception of progress is contested:

“Too much overhead still.”

“Feels like it’s really starting to become apparent... progress is becoming more visible.”

Number of meetings and efficiency:

“The amount is fine. Efficiency could do with some improvements. Some more assertive pushing of the agenda could be useful... we’re mostly introverted and need the occasional push to participate.”

“Still quite a lot of meetings. Sometimes seems that the agenda does not get completed and we go a bit astray.”

“The number of meetings seems to be understandable for the early development phase when we need to agree on so many things.”

“Meetings have been more effective than earlier. More decisions have been made and action points have been given.”

A bonus question in my questionnaire was: “How to make the process clear, accessible, light and not reliant on self-discipline or to rely on it as little as possible?” My idea: automate what’s possible, define criteria, use checklists, set up calendar events and reminders, agree on principles (create Team Contract) and leave the rest for individuals to decide.

Answers from the team members:

“… instead of a precisely defined process, this kind of "gate" or "criteria" might be better.”

“The fewer documented processes that are needed the better. Maybe we could focus more on the intended outcome of our activities”

“I like the freedom to choose how to work, but the end result must be the same from each individual's process.”

“Some kind of general guidelines are needed, but of course it gives flexibility if a person can decide some things.”

5.4.3 Agreed adjustments

In the first iterations the team was not sure about what to do or how, as expressed in the retrospective; they raised concerns about the lack of assignment of tasks and it wasn’t clear what to do. “Ensure that every task has an **owner** who cares [enough] to complete it” - this wasn’t maybe the best idea. When we adjusted our refinement practices and discussed and defined user stories in more detail, the need to have tasks assigned was somewhat diminished, so we decided to put more focus on respecting priorities and DoD; whenever somebody finishes all his tasks, s/he would take the next task from sprint backlog and not start something s/he likes more or has a better idea about. By responding more to the sprint backlog priorities, there is less and less need to assign tasks beforehand.

Add compliments as part of retrospective - in my questionnaire I asked everyone to complement their teammates and we all enjoyed reading those comments in the retrospective meeting.

In the next retrospective, we will make a short report about what has happened since the last retro to address the issues identified.

Some games would be nice to help generate debate and disagreement, and also to learn how to give feedback. Some games could also help to facilitate collaboration and get to know each other better.

Idea: Agree on some focused workdays or time of the day, there would be no meetings on those days.

5.5 Further development

5.5.1 Scrumban? Kanban?

Sprint boundaries - do we need them or not? Take a peek at Kanban - consider how it may support a more investigative approach to the product discovery and development.

Some advice from Contribyte was that we should not switch to Kanban only because we failed in Scrum.

5.5.2 Coding workflow and branching strategy

Next it seems to be useful to consider the potential of the Git history, how to read it? Who needs to read the history and what kind of information can it provide? What are the “use cases”?

Gatekeeper or integration manager - a single person who commits to the 'blessed' repository. Do some research to find out if this role might be useful in our development process, how we decide about publishing new functionality to customers.

Check whether we want to take on an “upstream first” policy or make fixes in the release first. An “upstream first” policy means making a fix first in a feature branch (as one would do any other change), merging it into master and then cherry-picking it for release. The question (or topic) will be relevant when we start creating release branches and publishing updates for customers.

Avoiding and resolving conflicts requires further learning and adjustment of the process. We have already agreed that updating our own feature branch as often as possible with the latest changes from the develop branch (*git rebase*) is one method to help in this. *As often as possible* was defined as “every time before making any change in one’s own branch or every time when there a PR is merged into develop”. Keeping the change sets as small as possible is the other helpful practice to avoid conflicts, as is further modularization of the product code.

5.5.3 Metrics to consider down the line

Goal: To state the success of the iteration (sprint) as a fact and to enable the development team to feel satisfaction.

Question: Are customers and/or users happy?
Metric: Ask about satisfaction from QPR internal customers in free form or open questions. Regular (sprint) reviews are performed to check if development is going in the right direction and fulfils its requirements. And after the MVP is released, maybe create a questionnaire.

Question: Does our framework help in developing the customer solution?
Metric: Average time to develop a specific customer solution. This can be measured later when the development of those solutions that use our framework is started. We aim to minimize that time.

Question: How many story points do we use per average user story? How big or small is the average user story?
Metric: Average "story size" in points - we want to minimize the scope (and so time to implement) of user stories.
This will be possible to measure when the team has completed more user stories.

Goal: Improve efficiency (max result with min expense) of code development and promotions (from development to production) for developers in development teams. To be efficient in bringing value to users. To "boost development throughput".

Question: How agile is our code review process? There was an anticipation of risk that pull requests may accumulate and create a bottleneck in the process, so a metric to track this is needed. However, the subjective impression (from the sprint retrospectives) currently is that pull requests are being processed well and fast enough. Hence, we will not implement that kind of metric immediately.

Metric: Pull Requests per day waiting in the review queue.

Metric: Number of findings per review ratio (define finding - comments?).

Metric: Pull Request lifetime: time from initiation of PR to merge.

We will need to create a dashboard to increase the transparency for stakeholders outside of the development team, so that they could see our progress and forecast around product development.

We want our QA efforts to catch the most critical bugs earlier in the SDLC, so they are cheaper to fix. Therefore, metrics to monitor the QA efficiency would aim to provide information on the effectiveness of the Test Automation (bugs found by TA vs bugs found by a human). Has our testing sufficiently "shifted left" - how many faults are found in the early development stages vs faults found in the released versions, taking into account the severity? The intention is to measure the CI&TA cycle time, monitor deployment or build failures, test failures, test the stability. We want TA to be useful and to give fast feedback to the developers – “Did I break something or not with my change?” Moreover, it is important to monitor how rapidly we move to deployment, to creating a build and solving TA problems, as well as reacting if delays lead to time wasted.

It would be useful to assess the modularity of the product with the aim of minimizing overlapping code changes, which would lead to less merge conflicts from having several developers work on the same file.

See Appendix 6. for full Goal-Question-Metric matrix.

6 Conclusion

Agile development is iterative by definition, and so this study set out to describe and analyse the initial phases of a new project and to seek further improvements from subsequent iterations, some of which are beyond the scope of the current study. This was the beginning phase of an experimental project that is aimed at trialling a different development process and different tools, to provide learning and an alternative positive example for other R&D teams at QPR Software. The DevApps team and new product development is still in the early stages, so this analysis can only arrive at preliminary conclusions.

Healthy development habits play a primary role in the software development process and the specific tools used are arguably secondary. It is much more crucial (and difficult) to learn good patterns, such as integrating code often, making small increments, building software continuously, reacting to failures rapidly, and doing verification during early development stages. Yet without trying various tools in practice, it is hard to realistically assess which tools serve the desired practices in the best way, so creating a "Guinea pig" team to run an experiment was a good idea. In making a final decision about the positive potential of changing the process tools throughout QPR, clearly more time and learning is needed.

We started by changing to Git from SVN. After 3.5 months it is still difficult to clearly see the effect of this change since many changes have happened at the same time and it is still difficult to attribute any single improvement to any single change factor. The behaviour change has begun, but it needs more time, while competence development and new capacity building also need to continue. Currently, the team has been able to identify the main questions and problematic areas based on their initial experiences. These will be addressed in part by focusing on further Git training. Git provides a convenient means to collaborate - on the feature branch or during a code review (web review tools typically come with some kind of Git solution).

What is obviously convenient is a new possibility to do code review via the web-based graphical interface of the Azure DevOps site and to make inline comments. Moreover, the Pull Request brings with it the possibility to set up several automatically enforced requirements, such as the need to resolve all code review comments, the need to get approval from a minimum of two reviewers, and the need to pass for the build and tests. As of now (mid November 2019), the CI pipeline is still evolving: builds and unit tests, API, integration, automatic E2E tests run on any commit that is added to the develop branch, meaning the CI automatically verifies changes after the pull request is merged into the develop branch. Automatic verification of every pull request from the feature branch, including for the build and automated tests, is being set up as I finalise this study.

Scrum improvement efforts were mainly focused on learning how to plan the minimal value increments (user stories) and to split the stories where possible. This is one of the biggest challenges to continue progressing with, since many of us have been used to working on bigger change sets (features) and this habit does not change fast. We have refined the Definition of Ready (for starting implementation) and Definition of Done for user stories and will continue strengthening the habit in respect to those criteria. We will continue focusing on grooming and defining the product development backlog together with business partners, to clarify and use customer needs as our core guidance.

The risk of creating competence silos seems to be significant and should be given further attention. The Daily Café format proved to be valuable, and we just started including “TED talks” into them on topics where team members are looking to exchange useful knowledge.

Preliminary observations were shared with the rest of R&D at QPR Software, but it will take some time to crystalize the lessons learnt and to make a plan on how other teams can utilize and apply those findings. The results of this study will be used for further improving processes in other development teams at the QPR Software Plc.

7 Discussion

A relaxed process created a feeling of freedom and boosted enthusiasm within the new DevApps team. The attempts to find new and improved ways of developing the product have proved valuable. Some questions raised by this process change remain to be clarified, which should happen as the new process becomes more established. A balance needs to be found between deciding what should be defined and fixed as rule/process/law and what should be up to the team, or even up to the individual to decide. The larger theoretical question constantly reappears - why do we need process? One answer is that process in the form of automatic tool restrictions and a set of criteria for any activity serve to ensure we are moving in the right direction and that we do it in the most efficient way.

Some agreed best practices cannot be enforced by tools or by criteria and so the challenge is to articulate those and seek ways to support those practices. One improvement goal for the process that was clearly emphasised in this study was seeking to lighten the process and make it convenient, such that people would naturally do it the right way. A concrete example of this would be frequently rebasing the private developer’s branch on to the develop branch (every time you are going to carry out a change, or every time you notice a new PR to be merged into the develop branch).

7.1.1 How to achieve a better process that is less reliant on self-discipline?

There may be many reasons for why people deviate from the established process. First people may **not know the process**, so the solution would be to make sure everyone is on the same page and understands why we want to do things in a specific way and not any other way. It is crucial therefore to mitigate the problem of getting everyone to **understand the rationale** behind the process. It can be addressed by involving people in the process creation and having as much discussion as possible about it. Arranging training may be not the best idea, since people may not pay the necessary attention. They will likely fare better in learning the process if they understand why we have it in the first place. It is easy to forget information that is not used, not questioned, and not discussed. To my mind, it seems more valuable to have a flexible moving process that allows us to be agile and get things done than to focus on describing the process in detail. Based on the experience gained through this study, I'd seek to define it in a minimalistic form, as criteria and checklists, and any room left in the definition would leave space for people to decide how to act on a case by case basis.

The tendency to look at the process only from the **perspective of one’s own role** needs addressing. In other words, understanding others may be critical when documenting something - one must think about a reader and what they would need to receive and understand. Role rotation, when possible, may be very useful in helping team members to understand other people's perspectives and points of view.

The **accessibility** of an established process is another issue. Being able to check what the process tells us to do requires access to specific resources and the ability to navigate models and to find the right one. In this, skills in using tools in the toolchain are necessary, where the process is maintained and all in all, several things would be expected to click into place. In daily use, the process should be easy to access. If it is more in the form of criteria or a checklist, then it would be right there where people need it, inside a user story, for example in a template for the user story. And the **possibility to change** it without any overhead or delay is also very important, since it keeps the process flexible and adaptable in the current situation.

The initial idea was to automate or **enforce** good practices, to make the right process for ensuring the most **convenient** way of doing things, such that there would be no urge to search for workarounds. We address this by defining core **guidance principles** and leaving as little as possible for self-discipline to follow what process dictates.

7.1.2 Smaller and better-defined user stories or features

This turned out to be one of the most difficult habits to change. The team had a lot to learn and this of course continues. In particular, the team is still learning what the user story actually is, how to define it, how to split it until we have a set of minimal value increments that can be implemented rapidly and independently of each other. The second aspect of the user story is the amount of details to provide in written form. It has been discussed that the user story and/or task should hold enough information so that any team member could pick it up and start working on it. This is important to mitigate the "bus-factor" – "the risk resulting from information and capabilities not being shared among team members, derived from the phrase "in case they get hit by a bus."" ("Bus factor," 2019)

7.1.3 Tuckman's stage

Though not a formal assessment, I think the DevApps team have arrived at the border between the storming and norming stages in the course of the first 3.5 months. It is easy to state 'No rush through phases!', but it is hard to apply this in practice when the business needs are pressing for the release of new product functionality. We found ourselves searching for the "sweet spot" between applying effort and time to learning and setting up tools and processes and the need to deliver a Minimal Viable Product by the end of 2019.

8 References

About - Git [WWW Document], n.d. URL https://git-scm.com/about/ (accessed 11.10.19).

Action research, 2019. . Wikipedia.

Ale Ebrahim, N., Ahmed, S., Taha, Z., 2009. Virtual Teams: A Literature Review (SSRN Scholarly Paper No. ID 1501443). Social Science Research Network, Rochester, NY.

Adopt a Git branching strategy, 2018. Git branching guidance - Azure Repos [WWW Document]. URL https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance (accessed 11.13.19).

Ariely, D., 2010. Column: You Are What You Measure. Harvard Business Review.

Atlassian, n.d. Epics, Stories, Themes, and Initiatives [WWW Document]. Atlassian. URL https://www.atlassian.com/agile/project-management/epics-stories-themes (accessed 11.10.19a).

Atlassian, n.d. Git Workflow | Atlassian Git Tutorial [WWW Document]. Atlassian. URL https://www.atlassian.com/git/tutorials/comparing-workflows (accessed 11.10.19b).

Atlassian, n.d. Forking Workflow | Atlassian Git Tutorial [WWW Document]. Atlassian. URL https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow (accessed 08.20.19c).

Atlassian, n.d. Merging vs Rebasing | Atlassian Git Tutorial [WWW Document]. Atlassian. URL https://www.atlassian.com/git/tutorials/merging-vs-rebasing (accessed 09.20.19d).

Atlassian, n.d. User Stories | Examples and Template [WWW Document]. Atlassian. URL https://www.atlassian.com/agile/project-management/user-stories (accessed 11.10.19e).

Basili, V.R., 1993. Applying the Goal/Question/Metric paradigm in the experience factory. Software Quality Assurance and Measurement: A Worldwide Perspective, 7(4), pp.21-44.Available at http://ssltest.cs.umd.edu/~basili/publications/proceedings/P62.pdf

Bus factor, 2019. . Wikipedia.

Carlsson, Marcus, Abandoning Gitflow and GitHub in favour of Gerrit, 2016. . Beepsend. URL https://www.beepsend.com/2016/04/05/abandoning-gitflow-github-favour-gerrit/ (accessed 11.13.19).

Cascio, W.F., 2000. Managing a virtual workplace. AMP 14, 81–90. https://doi.org/10.5465/ame.2000.4468068

Chacon, Scott, GitHub Flow [WWW Document], 2011 URL http://scottchacon.com/2011/08/31/github-flow.html (accessed 11.13.19).

Cutler, John, Good Process vs. Bad Process [WWW Document], 2018 URL https://cutle.fish/blog/good-process-vs-bad-process (accessed 11.3.19).

Driessen,Vincent, A successful Git branching model [WWW Document], 2010 . nvie.com. URL http://nvie.com/posts/a-successful-git-branching-model/ (accessed 11.13.19).

Epic, Story, Task and Bugs - Developer Wiki - Confluence [WWW Document], n.d. URL https://wiki.onap.org/display/DW/Epic%2C%2BStory%2C%2BTask%2Band%2BBugs (accessed 11.10.19).

Establishing an Agile Team Working Agreement - Tech at GSA [WWW Document], n.d. URL https://tech.gsa.gov/guides/agile_team_working_agreement/ (accessed 11.10.19).

Eusgeld, I., Freiling, F., Reussner, R.H., 2008. Dependability Metrics: GI-Dagstuhl Research Seminar, Dagstuhl Castle, Germany, October 5 - November 1, 2005, Advanced Lectures. Springer Science & Business Media.

Fowler, Martin, FeatureBranch [WWW Document], 2009. martinfowler.com. URL https://martinfowler.com/bliki/FeatureBranch.html (accessed 11.13.19).

Git - Book, Scott Chacon (Author), Ben Straub (Contributor), [WWW Document], n.d. URL https://git-scm.com/book/en/v2 (accessed 11.10.19).

Git fast forwards and branch management - Atlassian Documentation [WWW Document], n.d. URL https://confluence.atlassian.com/bitbucket/git-fast-forwards-and-branch-management-329977726.html (accessed 11.21.19).

Hammant, Paul 2017-2018 Trunk Based Development [WWW Document]. URL https://trunkbaseddevelopment.com/ (accessed 11.11.19).

Henderson, Ruth, What Gets Measured Gets Done. Or Does It? [WWW Document], Forbes, 2015 URL https://www.forbes.com/sites/ellevate/2015/06/08/what-gets-measured-gets-done-or-does-it/ (accessed 11.13.19).

Horton, M. and Freire, P., 1990. We make the road by walking: Conversations on education and social change. Temple University Press.

How do leaders create inspiring visions?, 2015. Leaders Lab. URL https://leaderslab.co.uk/how-do-leaders-create-inspiring-visions/ (accessed 11.08.19).

Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education.

Introduction to GitLab Flow | GitLab [WWW Document], n.d. URL https://docs.gitlab.com/ee/topics/gitlab_flow.html (accessed 11.3.19).

Jean McNiff, 2016. You and your action research project, 4th ed. Routledge.

Kaplan, R.S., Norton, D.P., 1992. The Balanced Scorecard—Measures that Drive Performance. Harvard Business Review.

Kotter, J.P., 1995. Leading change: Why transformation efforts fail.

Kreeftmeijer,Jeff, Using git-flow to automate your git branching workflow [WWW Document], 2010 URL https://jeffkreeftmeijer.com/git-flow/ (accessed 11.13.19).

Lean Coffee | Start one in your city!, n.d. URL http://leancoffee.org/ (accessed 11.13.19).

Lewin, K., 1958. Group decision and social change. New York Holt, Rinehart and Winston.

Mad Sad Glad Retrospective [WWW Document], n.d. . TeamRetro. URL https://www.teamretro.com/retrospectives/mad-sad-glad-retrospective/ (accessed 11.10.19a).

Mad Sad Glad Retrospective [WWW Document], n.d. . TeamRetro. URL https://www.teamretro.com/retrospectives/mad-sad-glad-retrospective/ (accessed 11.15.19b).

Mørken, F.V., 2017. Why you should stop using Git rebase [WWW Document]. Medium. URL https://medium.com/@fredrikmorken/why-you-should-stop-using-git-rebase-5552bee4fed1 (accessed 11.13.19).

Overeem, B., The Team Manifesto: the foundation every team needs – Barry Overeem – The Liberators, 2014 URL http://www.barryovereem.com/the-team-manifesto-the-foundation-every-team-needs/ (accessed 11.10.19).

Paolucci, Nicola, Simple Git workflow is simple [WWW Document], 2014. . Work Life by Atlassian. URL https://www.atlassian.com/blog/git/simple-git-workflow-is-simple (accessed 11.13.19).

Paolucci, Nicola, Git team workflows: merge or rebase? [WWW Document], 2013. . Work Life by Atlassian. URL https://www.atlassian.com/blog/git/git-team-workflows-merge-or-rebase (accessed 11.01.19).

Patton, R. (2006). Software Testing (2nd ed.). Available online: https://learning.oreilly.com/library/view/software-testing-second/0672327988/ch01.html

Pilone, D., Miles, R., 2008. Head First Software Development. O'Reilly Media, Inc.

QPR Software Plc, About QPR - Overview | QPR, 2019 [WWW Document]. URL https://www.qpr.com/company/overview (accessed 10.30.19).

QPR Software, 2019. . Wikipedia.

Ries, E., 2011. The lean startup : how today's entrepreneurs use continuous innovation to create radically successful businesses. Crown.

Role Expectations Matrix | Fun Retrospectives [WWW Document], n.d. URL http://www.funretrospectives.com/role-expectations-matrix/ (accessed 11.10.19).

Scrum Guide | Scrum Guides [WWW Document], 2018 URL https://www.scrumguides.org/scrum-guide.html (accessed 11.10.19).

Slant - 20 best alternatives to Axosoft as of 2019 [WWW Document], n.d. . Slant. URL https://www.slant.co/options/22030/alternatives/~axosoft-alternatives (accessed 11.10.19).

Solingen, R. van, Berghout, E., 1999. The goal/question/metric method: a practical guide for quality improvement of software development. The McGraw-Hill Companies, London.

Spolsky, Joel, Distributed Version Control is here to stay, baby [WWW Document], 2010. . Joel on Software. URL https://www.joelonsoftware.com/2010/03/17/distributed-version-control-is-here-to-stay-baby/ (accessed 11.10.19).

Spolsky, J., 2004. Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity, 2004. Corr. 3rd edition. ed. Apress, Berkeley, CA.

Tassey, G., 2002. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project, 7007(011), pp.429-489.

Teasdale, S., 2002. Culture eats strategy for breakfast!. Journal of Innovation in Health Informatics, 10(4), pp.195-196. ResearchGate URL https://www.researchgate.net/publication/287942641_Culture_eats_strategy_for_breakfast (accessed 11.3.19).

The cost of bugs – System Semantics, Ken Albin, 2014. URL http://systemsemantics.com/2014/08/the-cost-of-bugs/ (accessed 11.9.19).

Tuckman's stages of group development, 2019. . Wikipedia.

Understanding the GitHub flow · GitHub Guides [WWW Document], n.d. URL https://guides.github.com/introduction/flow/index.html (accessed 11.13.19).

van Rooden, Stephan, Product Backlog Refinement explained (1/3) [WWW Document], 2016 . Scrum.org. URL https://www.scrum.org/resources/blog/product-backlog-refinement-explained-13 (accessed 11.10.19).

Verheyen, Gunther , There’s value in the Scrum Values, 2013. URL https://guntherverheyen.com/2013/05/03/theres-value-in-the-scrum-values/ (accessed 11.10.19).

Verwijs, C., 2019. How To Kickstart A Great Scrum Team (10 practical things to do) [WWW Document]. Medium. URL https://medium.com/the-liberators/how-to-kickstart-a-great-scrum-team-10-practical-things-to-do-2143bdde1a8d (accessed 11.10.19).

We Make the Road by Walking: Conversations on Education and Social Change - Myles Horton, Paulo Freire - Google Books [WWW Document], n.d. URL https://books.google.fi/books?hl=en&lr=&id=zU8uFA4hlY0C&oi=fnd&pg=PR7&dq=horton+maked+the+road+by+walking&ots=q55lqDr-nc&sig=hGzFH1RKbt0mY26_B12MGnQc9uk&redir_esc=y#v=onepage&q=horton%20maked%20the%20road%20by%20walking&f=false (accessed 10.30.19).

What are Story Points?, 2015. . Agile Alliance. URL https://www.agilealliance.org/glossary/points-estimates-in/ (accessed 11.10.19).

What Gets Measured Gets Done - iSixSigma, 2012. URL https://www.isixsigma.com/community/blogs/what-gets-measured-gets-done/ (accessed 11.13.19).

What is ScrumBut? [WWW Document], n.d. . Scrum.org. URL https://www.scrum.org/resources/what-scrumbut (accessed 10.31.19).

What is Velocity in Agile? | Agile Alliance [WWW Document], n.d. URL https://www.agilealliance.org/glossary/velocity/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'velocity))~searchTerm~'~sort~false~sortDirection~'asc~page~1) (accessed 11.10.19).

9 Appendices

9.1 Appendix 1. Terms and Definitions given by DevApps team

QPR DevApps team specific:

Application template (in QPR UI 2.0 context) - a configurable skeleton or stub of an application, containing common functionalities and components, that can be used for quick and easy development of a QPR UI 2.0 solution for specific customer needs. We don’t use word “template” lately, so Application Framework includes all its meaning. **Application Framework** (in QPR UI 2.0 context) - comprises an application template, APIs, a set of reusable components (in a repository), a set of configurations (as in files), and documentation to enable building of framework based customer solutions.

Component (in QPR UI 2.0 context) - building block , React-module, UI components, such as buttons, graphs etc or bigger components that are collections of these small components. There are also data components that are invisible to the end user, but handle the data processing

Work item: **Epic** - a large set of related features that describes a business objective achievable by the application/system, takes months to implement.

Work item: **Feature** - a collection of related user stories in a product backlog, describes some logical functionality of an application, usually spans over several sprints to complete. Commits or Pull Requests don’t refer to features.

Work item: **User story** - sub-part of a feature, describes changes to be made in the product to bring minimal value increment to user. It can be implemented in a couple of days, but as maximum limit it should be possible to complete in one sprint. Can be realized by joint effort of several people. Commit or Pull Request may refer to it.

Work item: **Task** - contributes to user story, it can be about some functional change, or any other related work to complete Definition of Done for user story. Can be assigned and done by one team member. Commit or Pull Request may refer to it.

There was some discussion in responses about which purpose each of work items serves, where code changes belong (to features, user stories or tasks), and how to manage them. The division between software development task (which includes product code change) and any other tasks (design, documentation, prototyping, testing etc) in my opinion seems harmful for common team responsibility to meet the Definition of Done, so we will need to continue discussing that. A work item is needed to bring value to the user in the end, so it doesn’t matter much what is changed: code, UI design, or documentation etc.

Feature branch, User Story branch - don’t make any difference for us right now. Currently we use “feature/” prefix for git branch which contributes to User Story implementation, “fix/” for a bug fix, and “spike/” is used for experiments. Feature branches are always started from latest “develop” branch, important is that any branch can be accepted to “develop” only if it was approved by reviewers and passed verifications without failures. User Story may get code changes via several branches and several pull requests, so the smaller and the shorter-living branch is the better, so it allows to get new code to common branch for others to reuse and minimizes the possibility of merge conflicts.

We had an idea that we would implement a single user story in a single branch, so user story would be completed via one Pull Request, but currently User stories are not small enough, so this approach would take too much time.

Branching strategy - guidelines how to do concurrent development, CI and deployment of the product by utilizing the branching capabilities of the version control system. The workflow, which branches to use, how to create and when to delete them, the reasons to use certain branches etc.

Coding workflow - the "daily chores" of a developer, from fetching the latest version of source code from the "develop", making code changes and integrating them to making a deployable and ready for release product version. Also covers coordination of actions in the team when working together on the same codebase.

Software development workflow - may be a bit wider than just coding workflow, involving design, testing, deployment and documentation, but it’s hard to draw a line.

SW development working practices - the agreed on day-to-day ways we develop and test our software, including development environment set-up, collaboration, branching, agreed common workflow steps, rules and guidelines.

SW development process - a described process of how software is created, a formal description of a software development workflow.

SW development standards and principles - a set of best practices, principles, and guidelines the team commits to so that development and testing can be distributed more efficiently (there is no need for such guidelines in a one-man show kind of a development). Industry standards and proven operational models that makes life of SW developers easier. Might be related to actual coding and/or philosophical matters.

Code change annotation - PR comment, summary to help code reviewer explaining what was changed and why, maybe explaining some technical details of implementation. May also include regression analysis, mention what may be affected or need special attention.

Commit message - short, descriptive, unique comment to explain what was changed and why.

Commit - a coherent, atomic piece of implementation checked in to version control system, that has a clear, singular purpose.

Naturally for developers changing from SVN to Git, there was a mix up between SVN commit which is always pushed to the server and possibility to make local commit using Git.

Base commit vs **Fix commit** - the first one implements desired change and has meaning for Git history, the other fixes faults in initial implementation and can be squashed with the first one.

Merge vs **Rebase** - ways to integrate branches of code. Merge operation preserves history of commits in integrated branches. Rebase puts one branch on the tip of the other branch creating a set of new commits and creating a linear history.

Quality gates - a set of manual or automatic criteria at a certain point in development process that the code must pass before it can be promoted to the next stage (from feature branch to develop, from develop to release).

Staging - a testing environment for testing the code integrations, to let it mature. Also it can be an operation to move changes into staging area. It got different answers, but we will see if we age going to call one of our environments like that and if we need to clarify meaning later.

Production - a stage in the pipeline where the application is used by end-users in live environment. It was also explained as code or released version of software, as instance which is used by customers, as a production environment.

Main repository - our central Git repository in Azure server that keeps main public copy of the develop, feature, release branches of the software.

Master (branch) - is default branch name in git. A branch in Git which has the name “master”, it is often the default/assumed branch for certain tools and integrations.

Mainline (branch or repository?) - A branch or repository, where all of the approved changes are put. When creating a new branch, take the latest version from mainline. Or branch where development happens, there are all finished features that are not published yet. The branch on which features are branched from and merged to. Where changes are integrated for testing.

Codebase - the "sum" of all branches in the main repository. Or a set of all code related to some product (including tests etc). Codebase does not include third party frameworks and libraries.

Pull Request - a request to review and accept code into some branch. A comment “"call for a review before changes can be pushed to some codebase" -> term "pull" here is confusing” - shows that we need to talk about the difference in approach when code is pushed and when it is pulled.

Build - can be either a verb or a noun. Verb: to create an executable and/or running version of the software, to assemble a working application from source code. Noun: a package, an executable version of the software based on a specific revision of the "develop" branch.

Requirements management - elicitation of needed software functionality from customer research / feedback, writing down formal descriptions of said functionality prioritizing the functionality. Definition, prioritization, evaluation, cutting off duplicates, rejecting, and maintenance of a "requirements database" that can consist of several kinds of artefacts: documents, diagrams, mockups, designs etc. Especially, linking the artefacts to the product backlog items. Involves a system (tools) and activities.

Requirement - a statement or other form of description for a functional or non-functional characteristic of the software. Definition of what the software or some functionality should do, not how, black box view: it should be most-ly from the user/customer perspective.

Toolchain or **Tools pipeline** - a selection of software engineering tools used typically together in a sequential fashion to support and serve software development process.

Continuous deployment - A software engineering model where product increments deployed as soon as they meet the team's Definition of Done, acceptance criteria and quality criteria. May go through several staging phases for enabling e.g. manual testing. The staging phase can take several days from the initial commit / pull request merge. Or is it fully automated process to update production environment?

Commit-release pipeline - an ordered set of activities and tools required to push a single commit all the way to production deployment. Includes a bunch of automation ideally.

Unit tests - tests developers do to their code before having the code reviewed, covering the functionality of a single module, such as a function, or a component. Automated Tests that exercise a single unit of code (typically inside a single file) with all other dependencies mocked. They are a development aid and should not be thought of as tests in the classical sense. 99% unit test coverage with no other testing and your application is essentially untested.

Integration tests - tests that cover interaction and cooperation between modules or make sure the software works with the environment and other software it was meant to work with.

E2E tests - cover end-user interactions in the application. They test end-to-end functioning of the application from UI actions to database or external APIs from users point of view.

API tests - tests that are written to test functionality of the API. Making requests to API and expecting data or errors as responses, depending on the request. API testing is a type of software testing that involves testing application programming interfaces (APIs) di-rectly and as part of integration testing to determine if they meet expectations for functionality, reliability, performance, and security. A kind of E2E tests focused on server API endpoints (Imagining the server as a complete application in it’s own right).

9.2 Appendix 2. Team Kick-Off meeting script (Manifesto)

Set expectations: show forming_storming_norming_performing

------1.------------------15min What is manifesto?

Manifesto - a shared vision about teamwork and the required quality standards, agreed way of working among the team members.

Norms, values and principles on how people want to work together.

Write down on post-it note "What does team mean to you?" or "What it means to be a good team?" or "Is – Is not – Does – Does not" - Sometimes, it’s easier to describe something by telling what this thing is not or does not.

You may imagine the worst possible team and the best possible team.

Gather all notes and ask the team members to read out loud and stick to the board.

Cluster the ideas by themes,

name cluster and prioritize the themes by voting:

X clusters / 3 = Z votes for one team member. All your votes can go to one theme.

Count votes and make a sorted list

Select the top 5 themes

--------------------------------- 15min "what does quality mean to you?"

repeat clustering exercise

--------------------------------- 10-20 min. Build Manifesto

Team Values (top 5) poster and the other for the Quality definition (retain order).

Give the team some time to brainstorm about the other extreme of the earlier defined themes.

For example: the extreme of ‘fun’ might be ‘boring’.

Write down all the extremes on the right side of the poster.

Invite everyone to sign below each of the posters.

Tip: If certain members are reluctant to sign the posters, find out why. It may be that the team needs to revisit certain points on the posters.

Make sure the team manifesto is well visible in the team area. Refer back to manifesto the regularly.

9.3 Appendix 3. Team Kick-Off meeting script (Team contract)

What is Team Contract? - “ground rules” and team disciplines. How we will work together – responsibilities and expectations.

1 part 30 min: Define roles and tasks - who is responsible for what? Fill in Roles-Expectations Matrix.

2 part 30 min: Formulate “team disciplines” = “ground rules” by asking for example:

Where do we save team meeting notes?

When are the Scrum Events? And which? Who is expected to be present there?

What happens when someone’s late to a Scrum Event, or unable to join altogether?

When are we — as a team — happy with a Sprint? Sprint success is - what?

We want to communicate gotchas and helpful/harmful patterns - how?

Have a closer connection to CustomerCare and users - how?

How do we address risks like missing domain knowledge, unfamiliar tools, unfamiliar people, previous habits to be changed.

If we find that we need to make changes to the sprint – how we do it in a controlled manner?

Note positive (and negative) behaviors that can impact the Team.

Examples of team disciplines we could agree on:

don’t be afraid to ask,

be on time for meetings,

think about “future generations to come after us”,

be open about any problems,

share all what may be useful,

any changes to the sprint should be agreed with X? and recorded to be discussed in retro

Definition of done for SDT (Story?)

Define and adhere to Version Control rules

Define and adhere to coding standards

Update Backlog before Standup daily

Respect your team member’s time

Review working agreement (Team contract) regularly, especially in case of any changes or if a member breaks one of the ground rules (esp. if consistently).

These are some references for understanding Agile Team Working Agreements:

- Agile Team Working Agreements How To Guide
- Creating a Team Working Agreement
- Forming, Storming, Norming, and Performing: Understanding the Stages of Team Formation
- How to Create Agile Team Working Agreements
- Team ground rules and working agreements
- What is a team ground rule or team working agreement
- Work Agreements for a Scrum Team

9.4 Appendix 4. New DevApps Team’s Manifesto

A shared vision about teamwork and the required quality standards. Common norms, values and principles agreed among the team members. Because they give direction to our work, to our behavior, and our actions.

Definition of the Team: "What it means to be a good team?“

Fun and personal connection & Respect:

- Team is a group of people with shared values
- Good spirit (= trust and friendliness)
- Pizza Perjantai 2 notes
- Personal connection (= understanding and fun)
- Understanding (also common terms) – 2 notes
- Supportive, easy-going atmosphere
- Familiar people
- Willing to chat about anything
- Helping each other
- Flexible and tolerant attitude (we all make mistakes)
- Power of vulnerability
- Respect and trust, Respect each other, Respect

Cooperation & Collaboration, Feedback:

- Solve real problems together

- Promote cooperation and learning
- Share knowledge – 2 notes
- Support each other - 3 notes
- Cooperation & collaboration - 3 notes
- Effectiveness = cooperation + good spirit
- Share resources
- Frequent feedback and interaction
- Respectful feedback
- Prize others, give compliments

Shared / Common Goals:

- Team is a group of people with the same goal (working together to meet common goals)
- Rowing in the same direction
- Common definition of done
- Shared vision

Trust inside the team and outside:

- Trust (knowing people in the team) 2 notes
- Everyone does their own part
- Trust that people know what they are doing, but verify
- Tools cover 90% (?) of process, and little left for self-discipline
- Delivering what is promised
- Team takes responsibility on delivering value
- Team is acknowledged and appreciated by business and customers
- Empowered to solve problems in best possible ways
- Authority, Mastery, Purpose (D. H. Pink)

Innovation & Experimenting:

- Promoting personal and professional growth
- Be curious and nurture the culture of innovation and experimentation
- Continuous learning
- Trying new different things
- Ability to identify and adjust, and change things that don’t work well

Diversity:

- Diversity is a good source of different ideas and suggestions

- Different perspectives (personality) compliment each other 1+1>2
- Different experience (skills) compliment each other 1+1>2

Responsibility:

- Team really owns and drives the product vision
- Takes pride in its work
- Drive for excellence (in everything we see important)
- Sharing passion for quality
- Self-organizing team

Rewarding:

- Shared glory (and shame)
- Win as a team
- Prize and glory

Openness & Transparency:

- Openness and transparency
- No destructive competitiveness
- Don't steal ideas
- … is not secretive
- Informative daily scrum messages
- Common language (terms)
- Common definition of done

Definition of Quality: "What does quality mean to you?"

UX & Customer needs&satisfaction focused:

- Safe, productive, achieving the best results possible with the continued focus to improve for the customer and company.
- It means doing the job completely and accurately the way the customer wants it.
- Happy paying user
- Quality thing fits to the purpose of it: ketchup resistant shirt when eating fries
- Look and Feel is "premium", feels up to date
- Product is easy to use:
- There are no interruptions to work/use of the product, be it from crashes, slowness, or some other hindrance.
- There's no need for cumbersome workarounds.

- You don’t need to stop and wonder why something works like it does.
- Quality is right things happening on right place on correct time, continuously
- When something breaks, it's easy to fix
- Mistakes can be undone easily
- Performance is smooth.
- UI is intuitive and easily discoverable
- Compliance to customer needs (just enough)

Expectations Set&Met:

- Predictability and reliability both in the product and process space. Whatever that means..
- Promise: product or service will meet the minimum requirements for certain purpose
- Standard
- Agreed quality levels
- what to expect from a product or service of chosen quality level
- Evaluation of product or service based on certain criteria
- Set of evaluation criteria
- Reality is never perfect, so Q. is defined by given time and resources
- Compliance to customer needs (just enough)
- Great software: what is needed, on time and on budget.
- Things just work
- Cost vs benefit ratio

“I like it!” (dev perspective):

- I’d want to use this product myself
- Safe, productive, achieving the best results possible with the continued focus to improve for the customer and company.
- Bugs are rare

Commitment to Effort, we do our best:

- Doing things at it very best and nothing less.
- As a member of product development, delivering good quality product feels good, like we’ve achieved something special.
- Taking the necessary steps and time to reach the level of standard we set for ourselves, not cutting corners for short term gains, as cutting corners leads to paying the price later.
- Continuity: Product or service needs to continuously maintain its quality level
- Take extra day now, but save time in the long run

9.5 Appendix 5. Sprint retrospectives meeting script and questions

In all 5 retrospective meetings I asked everyone to classify their responses to the following categories:

MAD – List the things that are driving you crazy, what made you feel frustrated or annoyed. What is stopping you from performing at your best?

SAD – What are some of the things that have disappointed you or that you wished could be improved?

GLAD – What makes you happy or proud when you think about this sprint? What are the elements that you enjoy the most?

<u>A questionnaire before the 5th retrospective meeting:</u>

To save time in face-to-face retrospective meeting, I'd like to collect your "mad"/"sad"/"glad" beforehand.
Here is the list of topics to consider. You are free to skip any topic if you want. These are just open questions for open reflection.
Naturally any topic may have something good to say and some negative things to mention.
Please classify each of your comment in the answer as mad/sad/glad, so I could connect results with previous retrospectives.
You can still edit after you submit you answers.

<u>Questions:</u>

How do you feel about balance between defined process vs need to self-organize while searching for a new better process? Balance between freedom vs stability&certainty. Is "creative chaos" (flexibility, no strict rules) working well right now? How much energy do you have to make many decisions daily?

Bonus question (no need for "mad"/"sad"/"glad"): How to make process clear, accessible, light and not rely on self-discipline (or rely as little as possible)? What about getting rid of process as text document? Some part can be automated, some regulated by criteria (like DoD), check-lists, set up calendar events and reminders, agree on principles and leave the rest for individual to decide - what do you think about this idea?

Utilizing Scrum framework and practices. What we already do well or better than before, and what we need to improve?

What has improved since last Retro? In any area.

What was so good that we need to make sure it persists?

Do you see any risks right now?

Do you enjoy a selection of tasks you have on your plate?

Compliment your buddy, or give any other feedback.

Were any side tasks added to the sprint (after sprint plan was agreed) which affected the schedule?

Highlights from Contribyte report – what would you like to bring up?

Involvement of all team members in OD tasks – what's the situation now, what's the trend?

Reducing or splitting user stories - any comments?

Agility of process: we agile doing ...what? We are slow, bureaucratic … doing what?

Meetings amount and efficiency – comments, any trend?

What is your subjective feeling about Teams (chats) noise vs signal ratio? Have you found ways to sort and filter information flows?

How is the experience with documentation? Can you find info that you need? How is editing experience?

Any trend or change in feelings of inspiration and enthusiasm?

What can you say about learning possibilities and curiosity at the moment?

Atmosphere, team spirit, personal connection at the moment?

What is your experience of having arguments, agreeing or disagreeing about anything?

Perception of progress – any trend, comments?

Are you happy with sprint goals? Do we define them well enough?

What can you say about us meeting our sprint goals?

What about amount of tasks and workload?

How do you feel about a balance between doing things and documenting or communicating things?

Do you get enough information from others? General transparency related comments.

Finalization of tasks properly and neatly: is calendar reminder for dotting i's and crossing t's useful?

Free comments about Life, Death and the Universe!

9.6 Appendix 6. Goal-Question-Metric Matrix

			Discussion and motivation	Mechanism for data collection - how?
Goal	Purpose Issue Object Viewpoint & Context	To state the fact of success of the iteration and the development team to feel satisfaction		
	Question Metric	Do we meet sprint goals in a forecasted time? Sprint goals planned vs accomplished		Gut feeling and count AzureDevOps/dashboard/QPRUI 2.0 / Sprint goals planned vs accomplished
	Question Metric	Are we going to reach MVP goal (release) by the end of this year (2019)? MVP Burndown		AzureDevOps/dashboard/QPRUI 2.0 / MVP Burndown
	Question Metric	Are users happy? Ask about satisfaction from PMSM (QPR internal customers) in free form or open questions.		Regular (sprint) reviews to check if development fulfils requirements. And after MVP is released maybe create a questionnaire.

	Question Metric	Does our framework help in customer solution development? Average time to develop a specific Customer solution.(minimize)		When the development of customer solutions on the basis of our framework is started.
Goal	Purpose Issue Object Viewpoint & Context	To improve estimates of Features & User stories and allow development team learn its velocity and be able to forecast outcomes of iterations - to limit planned work and deliver it.		
	Question Metric 1 Metric 2	How small are user stories which we manage to define and work on? Average size (time from creating a branch to moment when it’s merged to develop) of user story completed in the sprint (minimise). Average time worked on user story - hours worked on all of development tasks (from Axosoft) per number of user stories completed during August-November 2019 (managed in AzureDevOps) (minimize)	We want to learn how to split features and user stories into smaller releasable value increments, and deliver smaller increments but more often.	Manually. Can be taken and calculated manually, when there are some user stories completed.

	Question Metric	What is one story point in our experience? If we assume that one point is minimal story. TBD		
	Question Metric	How many story points we use per average user story? How big or small average user story is? Average “Story size” in points.		
	Question Metric	How many story points we use on average in one sprint? Average number of points completed in sprint. (=velocity)	We can count average velocity retrospectively for autumn 2019, but to see the trend we need more established development process running, which is possible after “foundation stone” of the app and framework is laid down.	AzureDevOps/dashboard/QPRUI 2.0 / velocity guidance
	Question Metric	Do we remember to spend enough time for backlog refinement, so we would have enough user stories ready for implementation? Measure time used for grooming by the team per each sprint.	Tasks under user stories not defined in necessary detail, planning sessions are not efficient (better groom and pre-plan), grooming is done mostly in meetings, which is not the best use of time.	

		(should be around 10% of development Team's capacity).		Can be taken from Axosoft.
Goal	Purpose Issue Object Viewpoint & Context	Increase subjective developers satisfaction with tools and development processes – “developer’s experience”	We want effortless and painless development workflow through tools pipeline. Tools to support and guide developer (engineer, human) to the correct workflow. "Automate" the best practices as far as possible by setting up tool pipeline so it has gates which won't let anyone to move to the next step until conditions are met.	
	Question Metric	What is the “mood trend” in the team? Happiness measure from Sprint Retrospective .Count comments “mad-sad-glad" and see if there is any trend in number of positive vs negative comments.		Manually

Goal	Purpose Issue Object Viewpoint & Context	Improve efficiency (max result with min expense) of code development and promotions (from development to production) for developers in development teams.	To be efficient in bringing value to users. To “boost development throughput”.	
	Question Metric	What is the time of full CI&TA cycle? Time from build start to the end of last automated test execution. (minimize)	We want short feedback loop for developers working on user story branches. Taking into account coverage(number of tests?) and resources to run executions.	
	Question Metric 1 Metric 2 Metric 3	How agile is our code review process? Pull Requests in a queue waiting for review per day. Number of findings per one reviews ratio (define finding - comments?). Pull Request lifetime: time from initiation of PR to merge.	We want to keep Pull Requests queue minimal, get review asap, but also be review thorough.	AzureDevOps/dashboard/QPRUI 2.0 / Pull Requests Manually. Manually.

	Question Metric	How often main branch (develop) build fails? Number of failed builds on main development branch per day or week. Number of days without a build failure.	We want stable builds. Consider different kinds of failure causes.	
	Question Metric	How fast problems with build are solved? Time from first failure to problem identified/fixed.		
	Question Metric	How often main branch (develop) **tests** are broken (not working)? Number of failed TA cycles on Dev per day	We want stable TA	
	Question Metric	How trustworthy is TA result? False failures to true TA failures ratio (on first run, no reruns)	We want TA to be useful and informative for developers – “Did I break something or not by my change?”	
	Question Metric	How fast problems with TA are solved? Time from first failure to problem identified/fixed.		Guessing this could be registered only manually, no way to automate?

	Question Metric	How often main branch (develop) **deployment** fails? Do we need some deployment to pre-production statistics? Number of failed deployments on Dev per day Or nr of successful deployments in a row.(maximize)	We want reliable deployment	Hopefully this can be registered automatically
	Question Metric	How fast problems with build are solved? Time from first failure to problem identified/fixed.		Guessing this could be registered only manually, no way to automate?
	Question Metric	How bad and scary **merge conflicts** are? How often do they take significant time (define significant)? It could be a number of issues with merging a branch to develop per day/week, but currently it seems too much hassle to collect these data, so we rely on sprint retrospective and daily discussions.	We want less merge conflicts. Good selling point for other teams to follow DevApps.	

	Question Metric	How modular is product code? Is it often easy to work on different parts of code and avoid conflicts? Is it possible to measure? Static code analysis maybe?	Modularity of product may minimize overlappings (conflicts when several ppl work on the same file).	
	Question Metric	How big is delivered increment? How often we deploy? Epic or Feature (or Story?) development time from backlog to release (deployment to X). (minimize) NB *actual working time vs calendar time (both may matter, but not always)*		
	Question Metric	How many user stories per sprint team can complete (DoD)? Team velocity for user stories. Trend of the velocity	How many “bananas” we can eat per sprint? We can count average velocity retrospectively for autumn 2019, but to see the trend we need more established process and “foundation stone” laid down to build on top of.	AzureDevOps/dashboard/QPRUI 2.0

Goal	Purpose Issue Object Viewpoint & Context	To improve quality of released product for the users (consultants and end customers)	We want our QA efforts to catch most critical bugs earlier in SDLC, so they are cheaper to fix.	
	Question Metric	How successful (effective) are our QA efforts? **Is TA effective?** **Is our testing shifted left enough**? Bugs found in development vs bugs found on released versions. (taking into account severity)	Test strategy section: Product quality and test process metrics	Formula is needed to calculate ~ dev.bug/rel.bug *severity
		Effectiveness of TA: bugs found by TA vs bugs found by humans (taking into account severity)		difficult to implement, needs everyone to be organized and motivated to collect data.
		findings /per each code review – ratio (see under development efficiency goal)		

Goal	Purpose Issue Object Viewpoint & Context	To maximise Transparency of development progress towards business as well as customers		
		Public radiator showing the roadmap / Kanban board at user story / epic level. Public radiator showing epic burndown chart. "Epic burndown velocity" if there is such a thing. Maybe as a number of stories (and/of story points) WIP and done for burndown?	Make sure that business knows what we are doing currently and what we will pursue next.	Azure DevOps Stakeholders view

9.7 Appendix 7. Git workflow instruction

All changes done to this project are committed through feature-branches - see Branching strategy.

Workflow in short

Start:

git checkout develop
git pull
git checkout -b feature/<name> develop
git status
git add filename

```
git commit -m "descriptive commit message"
git pull --rebase origin develop
git push --set-upstream origin feature/<name>
```

Create PR, do rework, commit, update PR:

```
git pull --rebase origin develop
git status
git add filename
git commit -m "descriptive commit message"
git push origin/my-branch-where-pull-request-was-made --force-with-lease
```

When finished remove the branch:

```
git checkout develop
git branch -D feature/<name>
git remote update origin --prune
```

Workflow in more details with explanations

Update your local develop using standard git commands

```
git checkout develop
git pull  <--- or fetch (pull ~ fetch&merge)
```

Creating feature using standard git commands

1. Start feature branch from develop and checkout to your branch by:

```
git checkout develop

git checkout -b feature/<name> develop <--- prefix all feature branches with "feature/"
```

There is no *origin/<name>* yet, [*git push --set-upstream origin feature/<name>*] will create remote branch.

2. Start making your changes and committing

```
git status

git add filename

git commit -m "descriptive commit message"
```

3. **Rebase** on develop branch as often as possible = every time you are going to do some change or every time when there is PR merged to develop.

```
git pull --rebase origin develop  <--- By default, the git pull performs a merge, but you can force it to integrate the remote branch using --rebase option.
```

Rebasing allows you to avoid merge commits in feature branches.

OR

```
git fetch

git rebase origin/develop
```

Now resolve conflicts if there are any.

git status

git add fixed_conflicted_filename

git rebase --continue

git add another_fixed_conflicted_filename

git commit -m "fixed conflict in filenames"

and optionally *git push*

4. When changes are committed push your feature branch into Azure for (preliminary) code review by:

git push --set-upstream origin feature/<name> <--- now your local feature/<name> will track remote branch origin/<name>.

General advice is to avoid using git rebase after creating the pull request, if you use pull requests as part of your code review process. Because now your branch is public and others can make changes on it. Re-writing its history will make it impossible for Git and your teammates to track any follow-up commits added to the feature. (Atlassian: merging-vs-rebasing)

But this is possible to workaround:

A. if you are the only one contributor to this branch you can continue rebasing and pushing with flag --force.
B. Or it's perfectly legal to rebase onto a remote branch if your local branch fall behind.

5. Run TA & build to pass on your branch. Do necessary fixes (reiterate through steps 2, 3 & 4)

Create Pull Request and get code review

When feature branch is pushed to remote go to Azure DevOps:

- ❑ Repos (in left side menu)
- ❑ Pull requests (under the Repos)
- ❑ Press button New Pull request from top right corner
- ❑ Create Pull request from your feature branch into develop

You may create PR as a draft and publish later.

You can update the pull request by committing and pushing your feature branch.

git push origin/my-branch-where-pull-request-was-made --force-with-lease

Check git push --help for `--force-with-lease` option.

It's perfectly legal to rebase onto a remote branch X instead of develop. This is needed when collaborating on the same feature with another developer and you need to incorporate their changes into your repository.

For Pull Request reviewer

git checkout develop

git pull

git branch -r <--- will give you a list of remote branches (-a shows all), branches which aren't pulled down are listed in red.

git fetch origin feature/<name>

git checkout remotes/origin/feature/<name> <--- and you checkout to the one you want.

and then you can build the application and test the feature and check how the UI looks.

Collaboration on a public feature branch

git fetch <--- to update your remote-tracking branches under refs/remotes/<remote>/

git branch -r <--- to see all remote branches and select the one you are interested in.

git checkout remotes/origin/feature/<feature-name>

git checkout -b feature/<feature-name> origin/feature/<feature-name> <--- creating local branch for remote public branch.

git rebase develop

Make changes and commit. Finally:

git push

Finish the feature with standard git commands

PR can be accepted to Dev when:
+1 Code builds
+1 TA passes
+2 from two other reviewers (or +2 from one reviewer).

We don't do automatic merging now, we want to have a human making final decision for the beginning.

After PR is approved:

1. In azure press Complete on top right corner on your Pull request

2. You may want to remove your local branch with:

    git checkout develop

    git branch -D feature/<name>

3. Update your list of remote branches by running:

    git remote update origin --prune