Module 10

Agile Teams

Topics:

- Scrum model
- How to manage branches effectively
- Pull request (PR) and continuous integration (CI)
- Agile development lifecycle
- How to report bugs
- P&D’s perspective on managing teams

Scrum Model - Roles

- Scrum Model
  - A popular way to organize / lead an Agile team

- “Two-pizza” team
  - 4-9 people, which is a typical team size for SaaS projects

- Scrum Master
  - A member who “owns” this iteration (also called Sprint)
  - This member is expected to keep the team focused on the task at hand, enforce team rules, remove impediments that prevent the team from making progress, etc.
    - Ex) Enforce coding standards, drive daily scrums, etc.

- Product Owner
  - A member (not Scrum Master) who represents the voice of the customer and, therefore, prioritizes user stories.

- Scrum relies on self-organization, and team members often rotate through different roles

Scrum Model - Activities

- Daily Scrum
  - 15-minute daily standup meeting in which each team member answers these questions:
    - What have you done since yesterday?
    - What are you planning to do today?
    - Are there any impediments or stumbling blocks?

- Pair programming
  - Reduce development time (absolute time, not person-hour) and improve software quality
  - May result in higher programming costs due to two people working on the same task
  - Driver: works to complete the task
  - Observer/Navigator: thinks strategically about future challenges and makes suggestions to the driver.

Scrum Model - Activities

- Iteration Planning Meeting (IPM)
  - Happen at the beginning of each iteration
  - Discuss stories and estimate their points; Planning Poker
  - Decide which features to do first based on the customer’s needs (or the Product Owner on behalf of customer)

- Retrospective Meeting (Retro)
  - Happen at the end of each iteration
  - Plus/Minus/Interesting (PMI): Each member writes down what they thought went well, went poorly, and was unusual or noteworthy (neither good nor bad) during the iteration

Q1

No one is talking at the beginning of the Iteration Planning Meeting (IPM). Who should break the silence?

A. Product Owner
B. Customer
C. Scrum Master
D. Senior developers

Q2

If a software project is behind schedule, it may not be a good idea to add people in order to catch up because

A. programmers are expensive
B. it takes time to bring new programmers up to speed
C. communication overhead increases
D. SaaS cannot be built with large teams

Q3

Select all that are true: Agile developers should value

A. Individuals & interactions over processes & tools
B. Working software over comprehensive documentation
C. Contract negotiation over customer collaboration
D. Following a plan over responding to change

Q4

True or False: The basis of agile development is that each phase of design and implementation should be fully completed before moving on to the next phase.

A. False
B. True

Version Control & Branching

- **<u>Branches</u>** allow variation in a codebase
  - **Feature branches:** support development of new features without destabilizing working code
  - **Release branches:** allow fixing bugs in previous releases whose code has diverged from main line of development

- **Merging changes** from one branch into another may result in conflict merges for certain files so we must always commit before you merge and before switching to a different branch to work on.

- With Agile + SaaS, feature branches are usually **short lived** and **release branches are uncommon.**

Branch Management - Feature Branch

- You should think branch as a DAG (directed acyclic graph) of commits, rather than a linear sequence.

- The master branch always contains a stable working version of the code

- Each branch is a feature in development

- Feature branch will be merged into the master branch after it is completed. Before the merge happens, code review via Pull Request is required. The developers may also need to resolve some merge conflict.

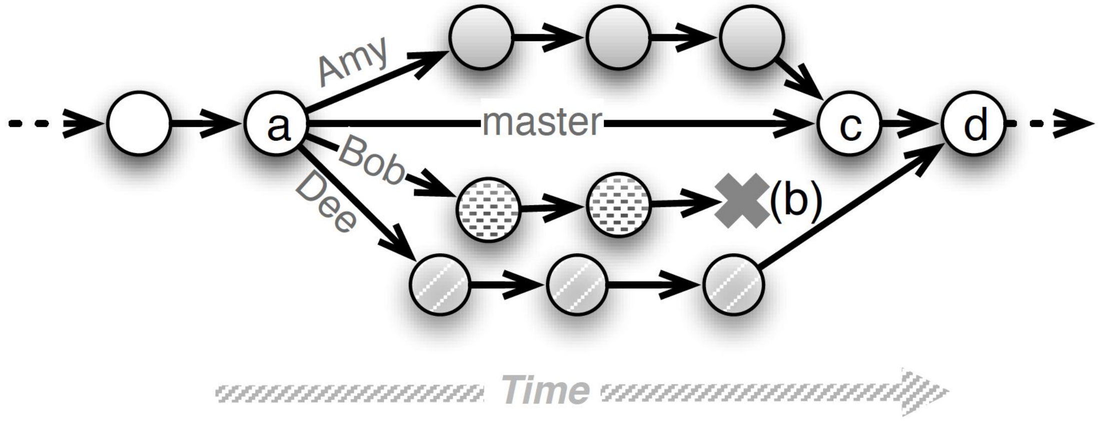


Figure 10.2: Each circle represents a commit. Amy, Bob and Dee each start branches based on the same commit (a) to work on different RottenPotatoes features. After several commits, Bob decides his feature won't work out, so he deletes his branch (b); meanwhile, Amy completes her work and merges her feature branch back into the master branch, creating the merge-commit (c). Finally, Dee completes her feature, but since the master branch has changed due to Amy's merge-commit (c), Dee has to do some manual conflict resolution to complete her merge-commit (d).

Branch Management - Release Branch

- The master branch still always contains a stable working version of the code
- The release branch has multiple versions of the code
- Common in non-SaaS software, such as libraries

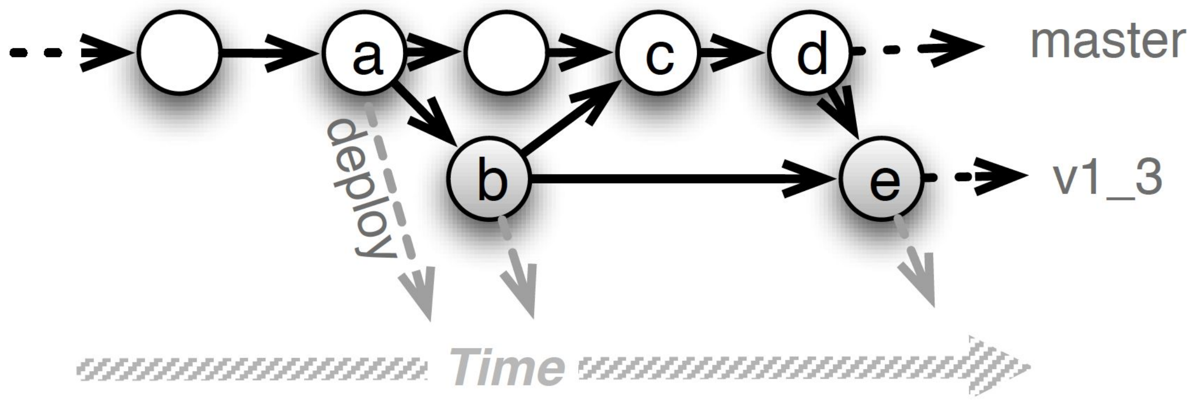


Figure 10.3: (a) A new release branch is created to "snapshot" version 1.3 of RottenPotatoes. A bug is found in the release and the fix is committed in the release branch (b); the app is redeployed from the release branch. The commit(s) containing the fix are merged into the master branch (c), but the code in the master has evolved sufficiently from the code in the release that manual adjustments to the bug fix need to be made. Meanwhile, the dev team working on master finds a critical security flaw and fixes it with one commit (d). The specific commit containing the security fix can be merged into the release branch (e) using `git cherry-pick`, since we don't want to apply any other master branch changes to the release branch except for this fix.

Merge Management & Small Teams

- Small teams typically use **shared-repo model** in which pushes and pulls use a single authoritative copy of the repo, often stored in the cloud on Github or on company server.

- Before changing files, committing your own changes locally and then merging the changes made by others is needed. In git the easiest way to merge changes from the origin repo is by pulling (git pull)

- If changes can't be automatically merged, you must manually edit the conflicted file by looking at the conflict markers in the merged file, and then commit and push the fixed version. In git, conflicts are considered resolved when the conflicted file is re-committed.

```
Roses are red,
Violets are blue.
<<<<<<< HEAD:poem.txt
I love GitHub,
=======
ProjectLocker rocks,
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:poem.txt
and so do you.
```

Pull Request (PR)

- A request that asks to merge one branch to another branch
    - Ex) a feature branch to the master branch

- Steps to take after sending a PR:
    - Every team member can see and comment on the PR
    - The requester is responsible for:
        - Addressing the concerns from team members
        - Making additional commits to fix bugs
        - Fixing code style
        - Resolving merge conflicts
        - etc.
    - Once the PR looks good, a team member will give approval to the PR
    - The requester will merge the branch.

Continuous Integration (CI)

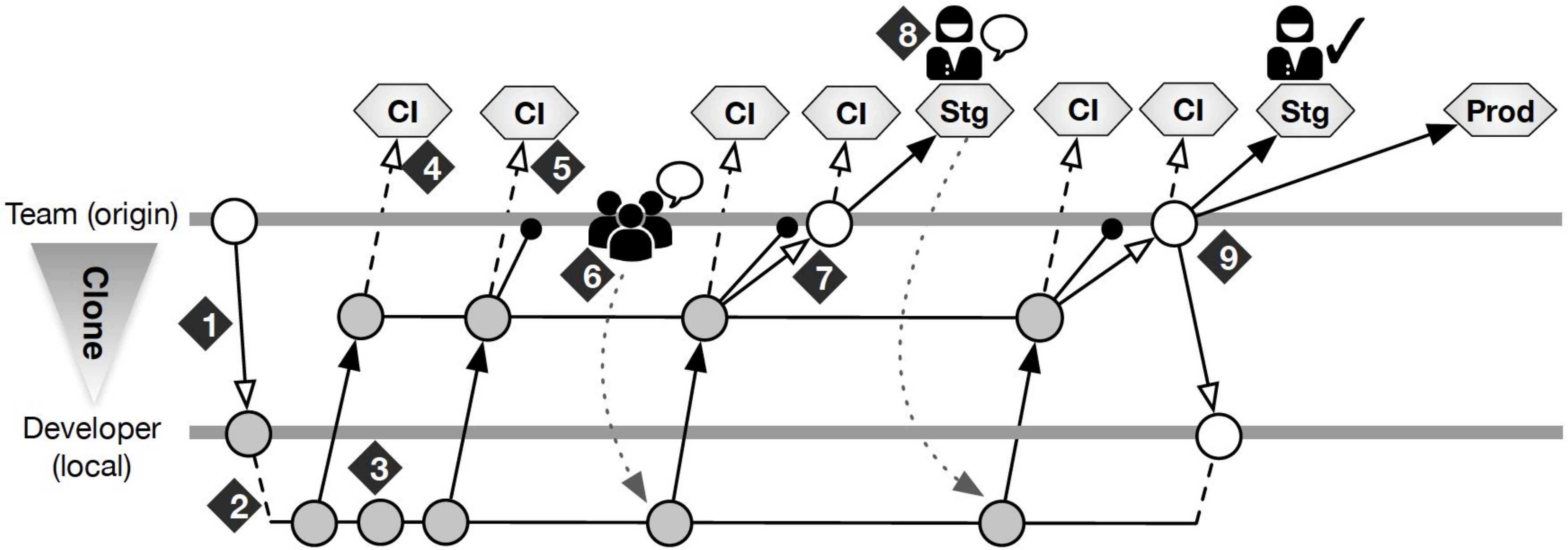
- The practice of automating the integration of code changes from multiple contributors into a single software project (master branch).

- Minimizes the time between when changes are made on a feature branch and when those changes are merged into the master branch and deployed for customer review.

- CI service
  - A service that automatically runs all test suites, usually in the cloud, each time some changes are made (e.g. when code is pushed).
  - Can also automatically detect code smells, run style checks, etc.
  - Ex) GitHub Actions, Travis

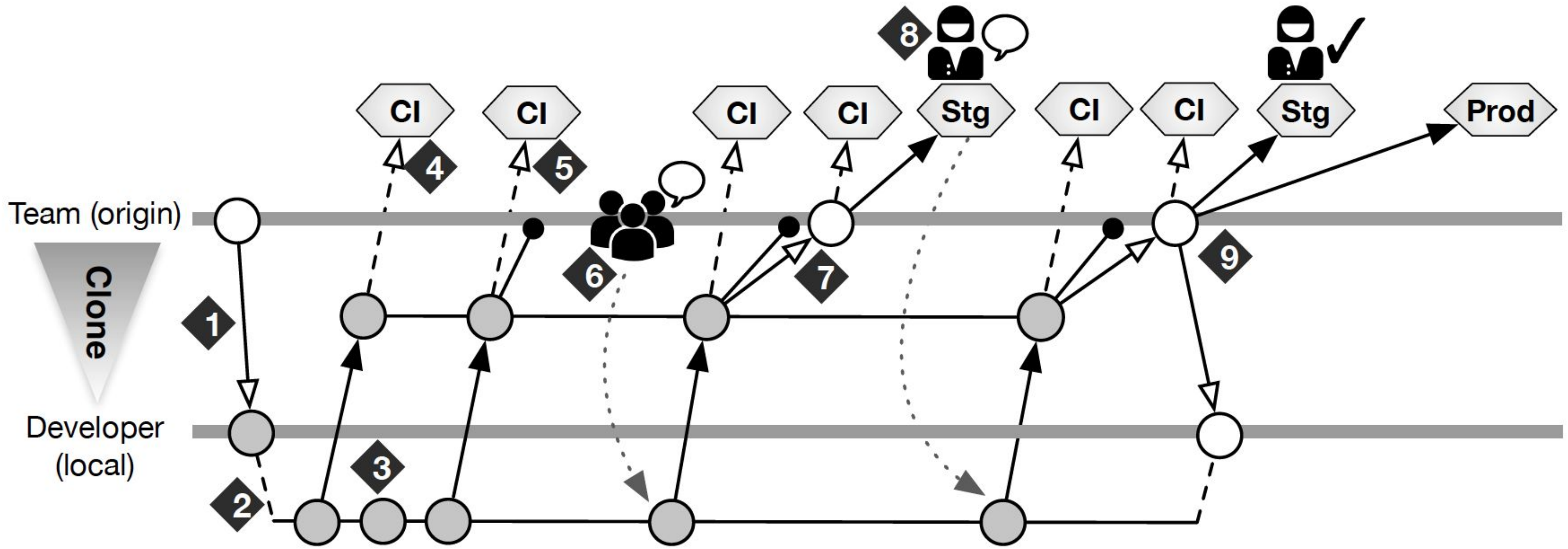
- Staging servers
  - Servers that resemble the production environment but in a much smaller scale
  - Provide a safe place to deploy new features for customer review before they are deployed in production.

An Example Feature Development Workflow



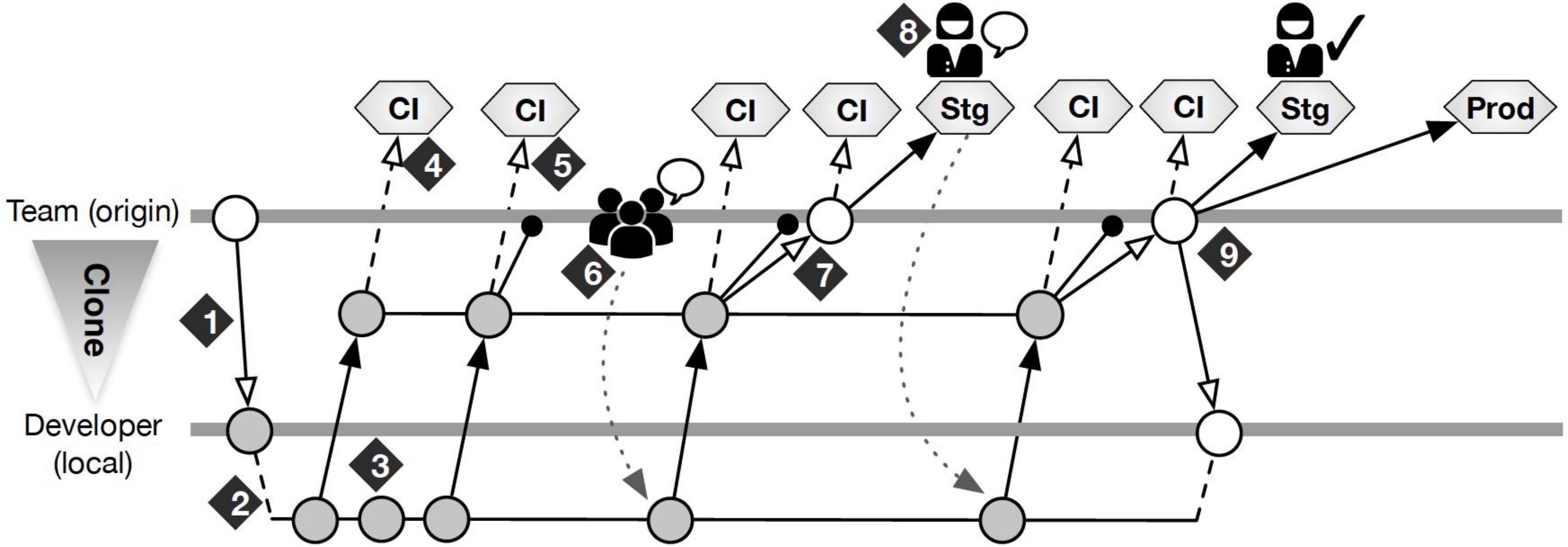
1. On `local/main`, you use `git pull origin main` to ensure you have the most up-to-date version of `origin/main`.

An Example Feature Development Workflow



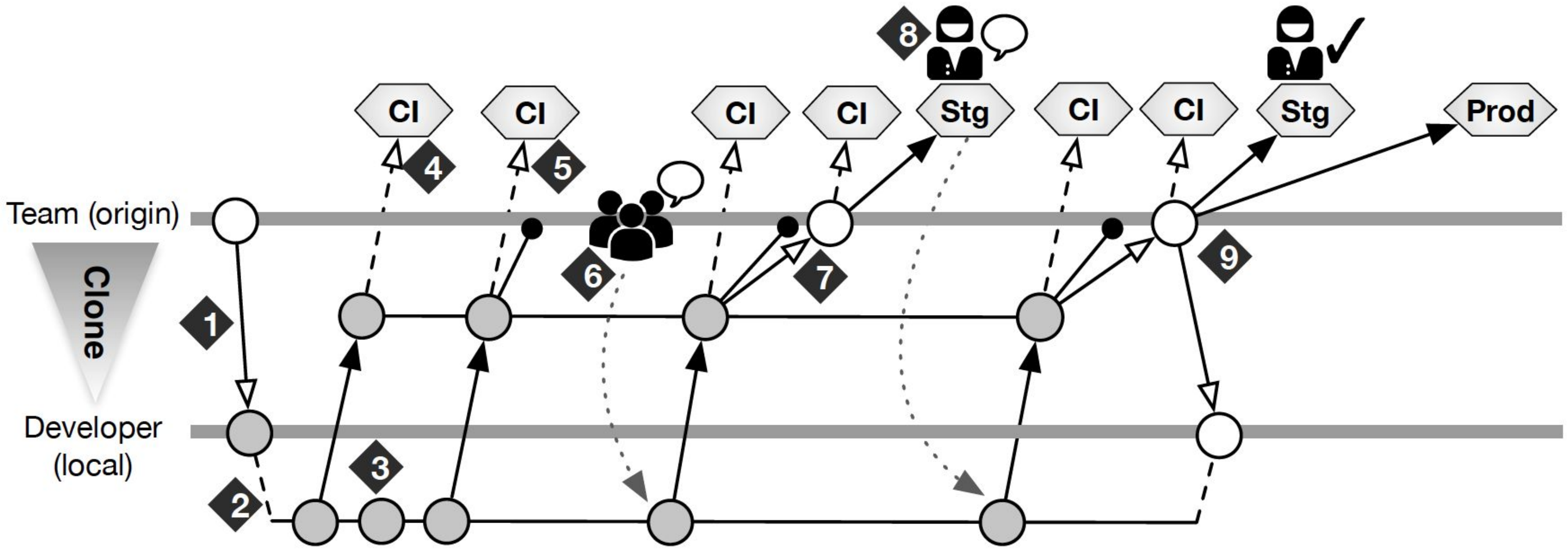
2. You create a new branch called `feature-branch` for the story. At this point, the story state changes from “Unstarted” to “Started” on Pivotal Tracker.

An Example Feature Development Workflow



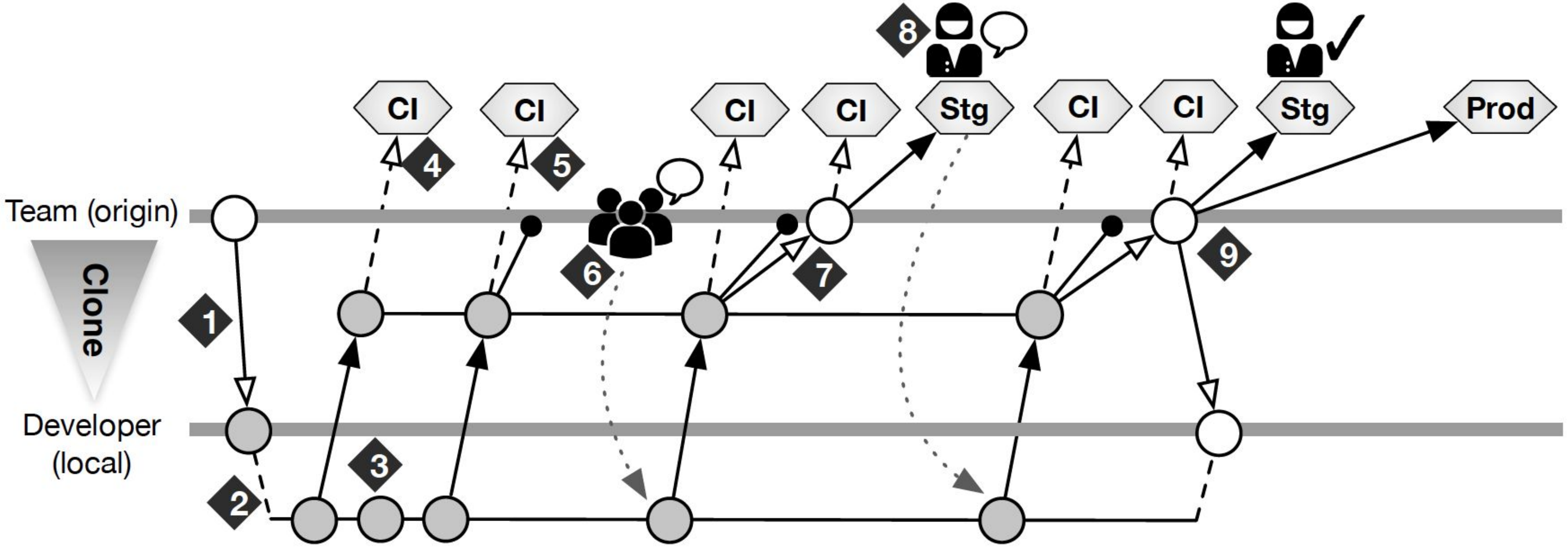
3. You write tests and code for the feature, committing frequently. You also periodically push the feature branch’s commits to `origin`. This keeps `origin/feature-branch` up-to-date with your `local/feature-branch` branch.

An Example Feature Development Workflow



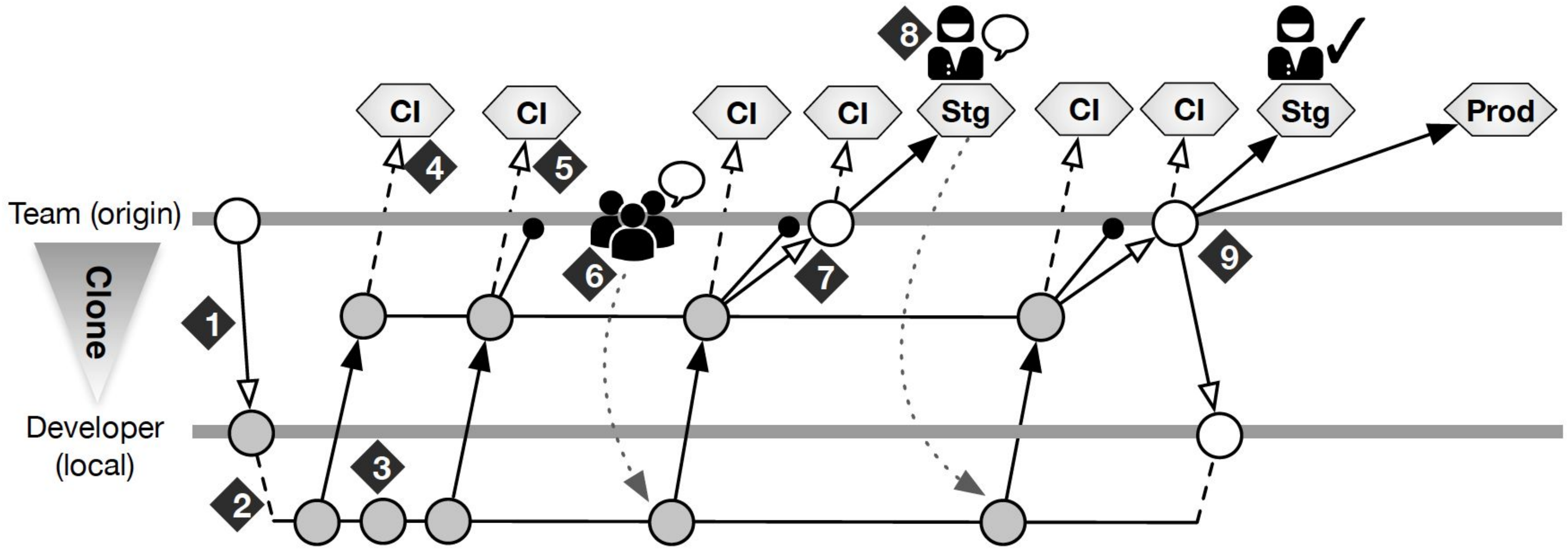
4. Your team has the workflow (CI service) configured so that any push to `origin` will automatically trigger an external CI run on whatever branch was pushed, such as `origin/feature-branch`.

An Example Feature Development Workflow



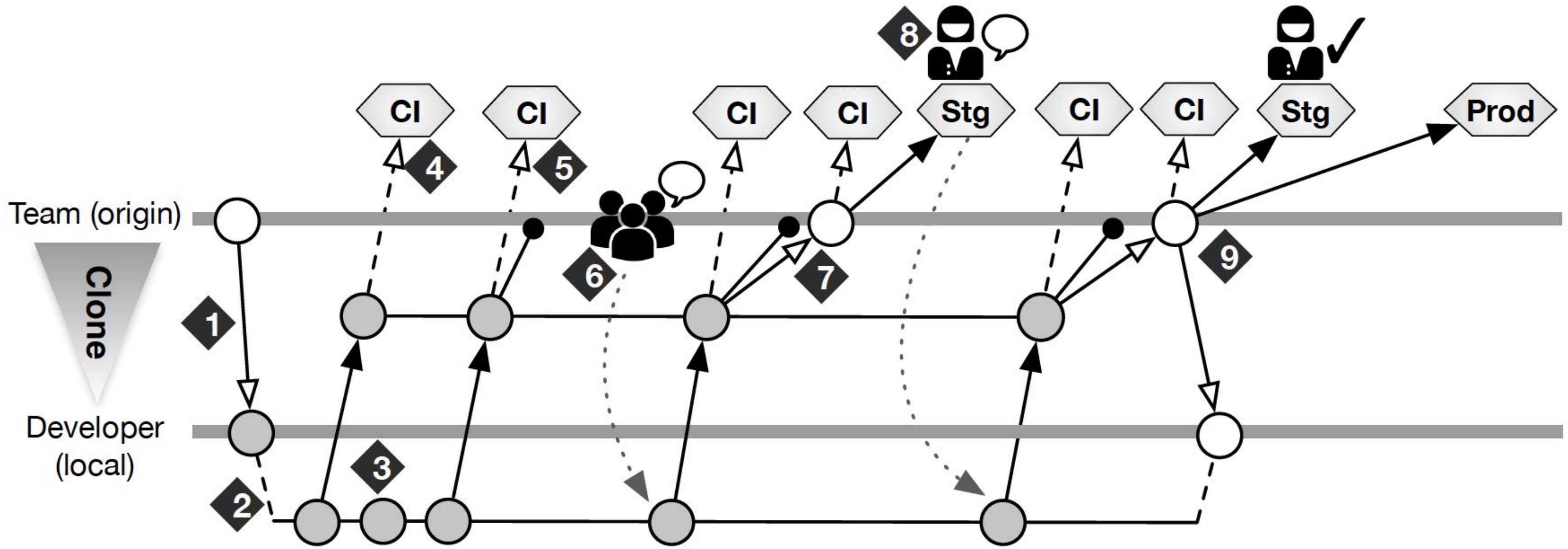
5. When the code and tests are ready, you mark the story “Finished”, and open a pull request. Note that the PR asks to merge `origin/feature-branch` to `origin/master`. This means you must push the most recent commits on your `local/feature-branch` to `origin/feature-branch` so that the PR will include your latest commits.

An Example Feature Development Workflow



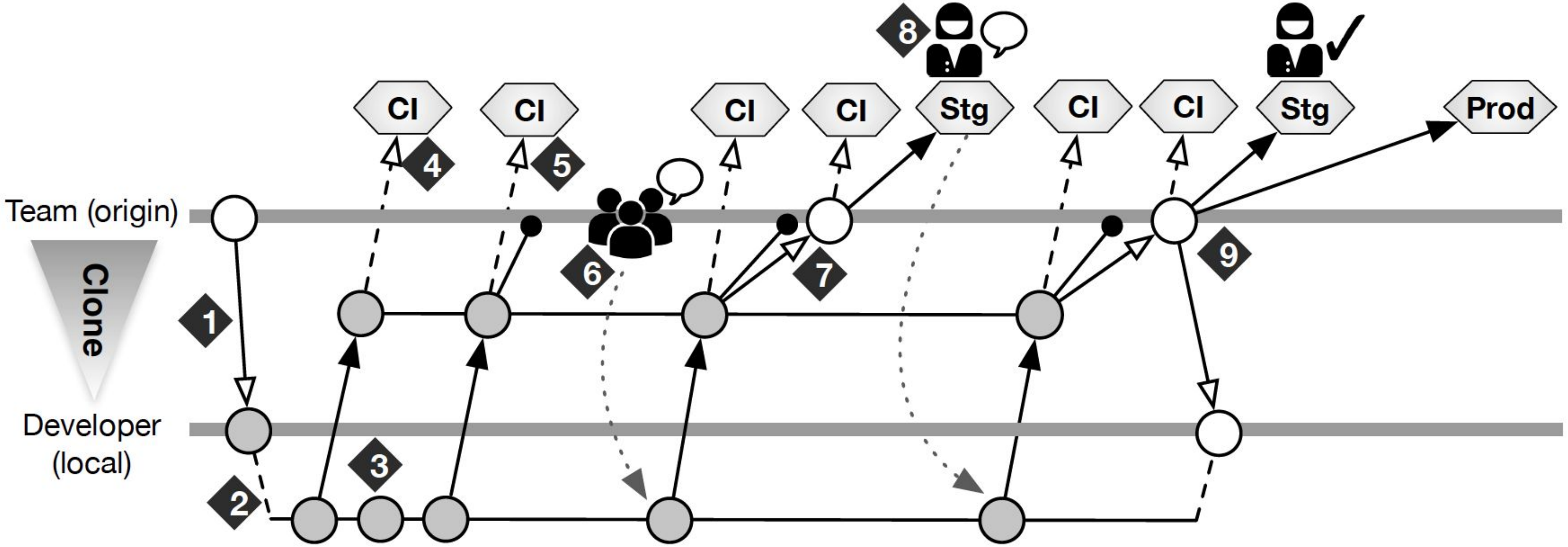
6. Other team members review and comment on the PR. In this case, they do not approve and require some changes. You then make the changes and reopen the PR (or can choose to open a new one).

An Example Feature Development Workflow



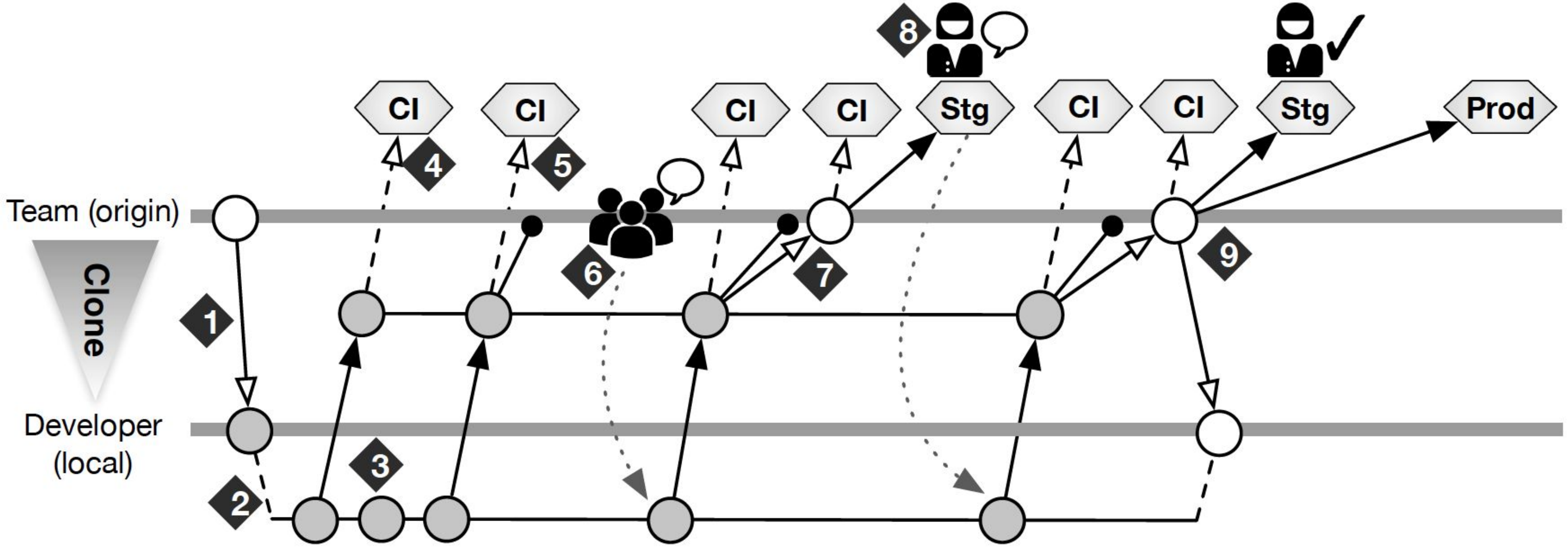
7. The revised PR is accepted and the changes are merged into `origin/master`, triggering another CI run. This is necessary because `origin/master` now includes not only this PR but possibly other developers’ PRs that have been merged since Step 1

An Example Feature Development Workflow



8. CI passes. `origin/master` is deployed to a staging server and the story is marked “Delivered”. The customer can now review and comment on the new feature. Unfortunately, the customer requests revisions, resulting in another round of changes, PR, and merging.

An Example Feature Development Workflow



9. After another round of changes, PR, and merging, the customer finally accepts. The feature will then go into production, and the story will be marked “Accepted”

Q5

Assuming you are using the "deploy from master" methodology in your team, which of the following is NOT a good practice for adding code for a new feature?

A. Fork, develop the feature on master of your own fork, and do a pull request
B. Fork, develop the feature on another branch of your own fork, and do a pull request
C. Create a branch on the main repo, develop on that branch, and do a pull request
D. Develop directly on master if the feature only affects one or two files

Q6

True or False: There are some situations when you should directly make changes to the master branch

A. False
B. True

Q7

In the "deploy from master" discipline, when is it acceptable to merge a pull request to the master branch that includes failing tests?

A. Never
B. nEver
C. neVer
D. nevEr
E. neveR

Q8

After a customer meeting, how does an effective team prioritize working on the user stories that were discussed, especially if the amount of work may exceed what the team can handle during that iteration?

A. The stories with the largest point values (most difficult to implement) are prioritized highest
B. The product owner assumes the role of representing the customer's view, and prioritizes stories accordingly
C. Each developer is responsible for claiming and prioritizing their own stories from among those discussed at the meeting, as long as they communicate with the other developers to avoid overlapping effort
D. The stories with the highest uncertainty of effort are prioritized highest

Q9

Which statement best reflects how Agile Scrum-based teams would prioritize and deliver User Stories?

A. The Product Owner sets the number of points each story is worth
B. The Project Manager sets the number of points each story is worth
C. Anyone can work on a story and mark it Finished, but only the Product Owner can officially Deliver it to the customer
D. Anyone can mark a story Delivered, but only the Product Owner can mark it as Accepted or Rejected

Five Rs: Reporting & Fixing Bugs

- Process on managing the **phases of the bug’s life-cycle:**
  - **R**eporting a bug
  - **R**eproducing the problem or **R**eclassifying it as “not a bug” or “won’t be fixed”
  - **R**egression testing that demonstrates the bug
  - **R**epairing the bug
  - **R**eleasing the repaired code

- Example of **R**eclassifying: Bugs that are enhancement requests or that occur only in the obsolete versions of the code or in unsupported environments may be reclassified to indicate they are not going to be fixed.

- **R**egression testing: Running tests to ensure that previously developed and tested software still performs after a change. In this case, it ensures that previously-fixed bugs do not reappear.

- No bug can be closed without an automated test demonstrating that we really understand the cause of the bug

Plan and Document Perspective

- **Project managers** are in charge and they write the contract, recruit the team and interface with the customer.

- The **project manager** documents the project plan and configuration plan along with the v**erification and validation plan** that ensures other plans are followed

- To limit the time spent in communicating, **groups are 3-10 people.** They can be composed into hierarchies to form larger teams reporting to the project manager with each group having their own leader.

Plan and Document Perspective

- Guidelines for managing people include giving them **clear goals** but **empowering them**, and starting with the **positive reviews but being honest about shortcomings** and how to overcome them.

- While **conflicts** need to be resolved they can be helpful in finding the best path forward for a project.

- **Inspections** like design reviews and code reviews let outsiders give feedback on the current design and future plans which lets the team benefit from the experiences of others, as well as making sure good practices are being followed.

- **Configuration management** includes version control of software components, system building of a coherent working program from such components and release management to ship the product to customers.

Q10

Comparing Plan-and-Document vs. Agile approaches to cost estimation, which statements are TRUE?

A. Both try to identify and estimate the impact of possible risks
B. Multiple iterations of the prototype are a risk-reduction tactic that is present in Agile but absent from P&D methodologies
C. In both methodologies, nontechnical risks are more likely to jeopardize the schedule than technical risks.
D. Both must deal with customers asking for changes after schedule is in place and project starts

Attendance

Attendance Link:

tinyurl.com/cs169a-dis-11