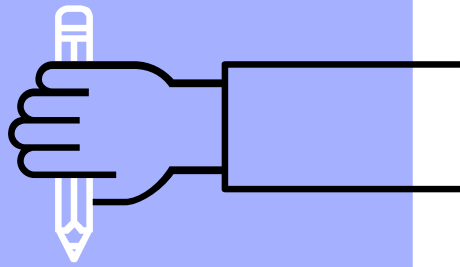
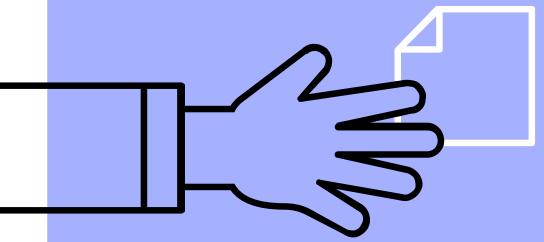

FOUND
7 Lost Principles of Continuous Delivery

It's not about "can we build it?", it's about "should we build it?"



1.
Create a repeatable and reliable process

For releasing software



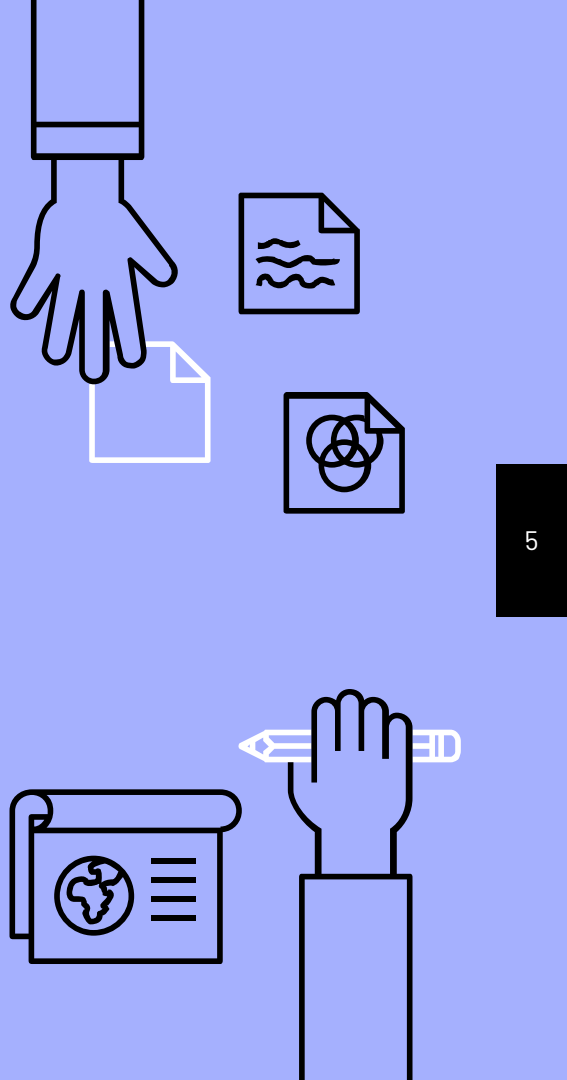
BIG Question

"How long would it take your organization to deploy a change that involves just one single line of code?"

The underlying purpose

Bamboo

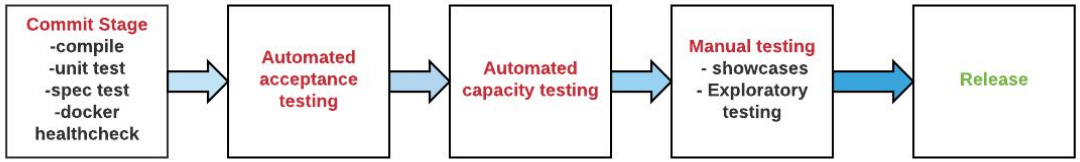
AWS CodePipeline

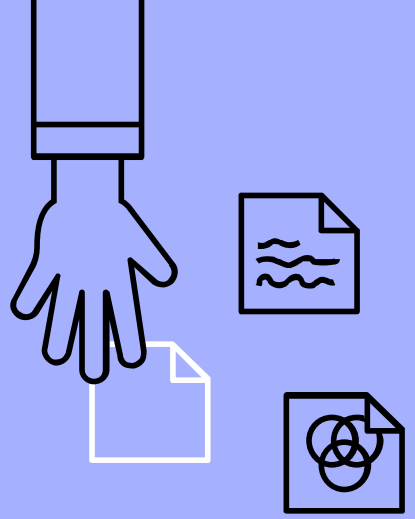


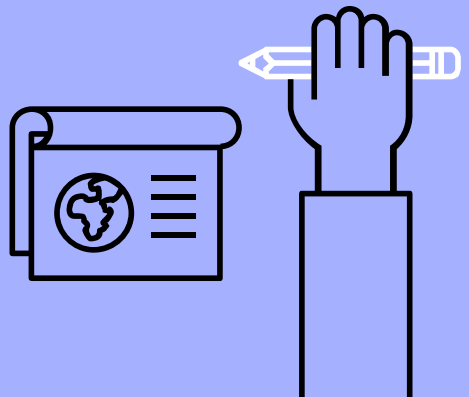
Every commit -> Source Control

Change

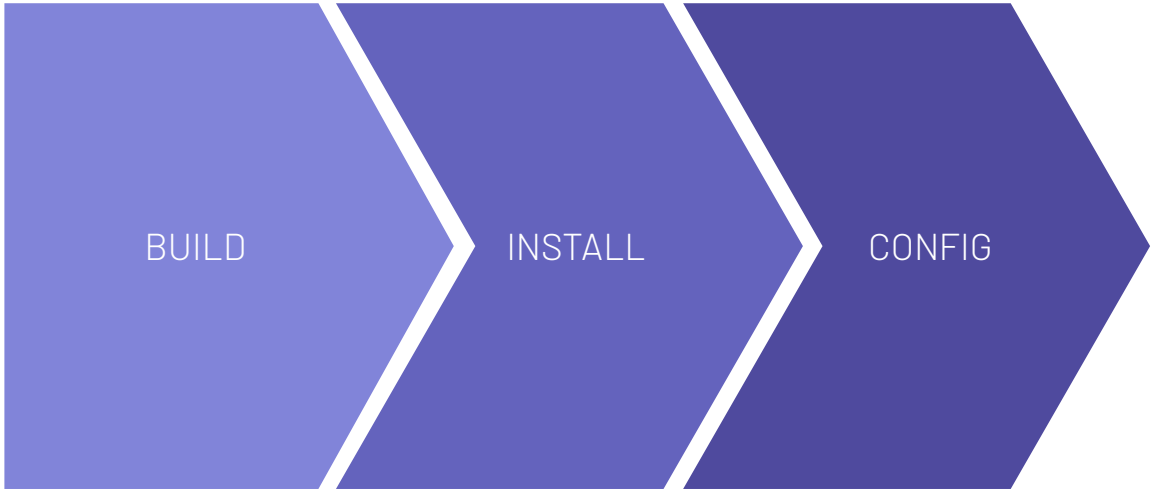
Release Candidate



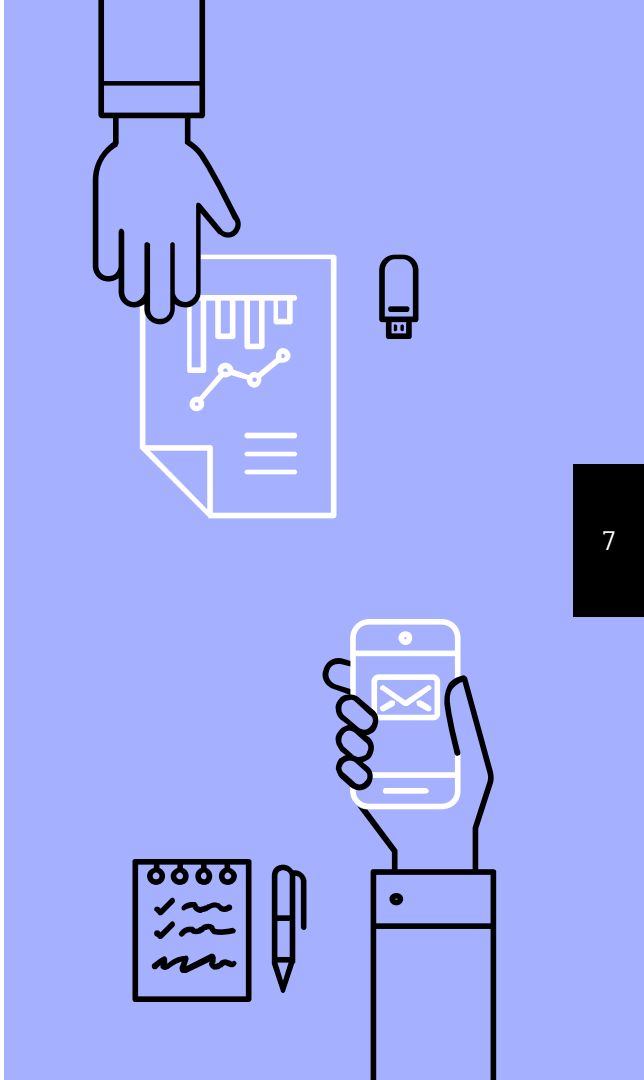




Deploying through the pipeline



3 Main Phases



Repeatable Process

Building the infrastructure

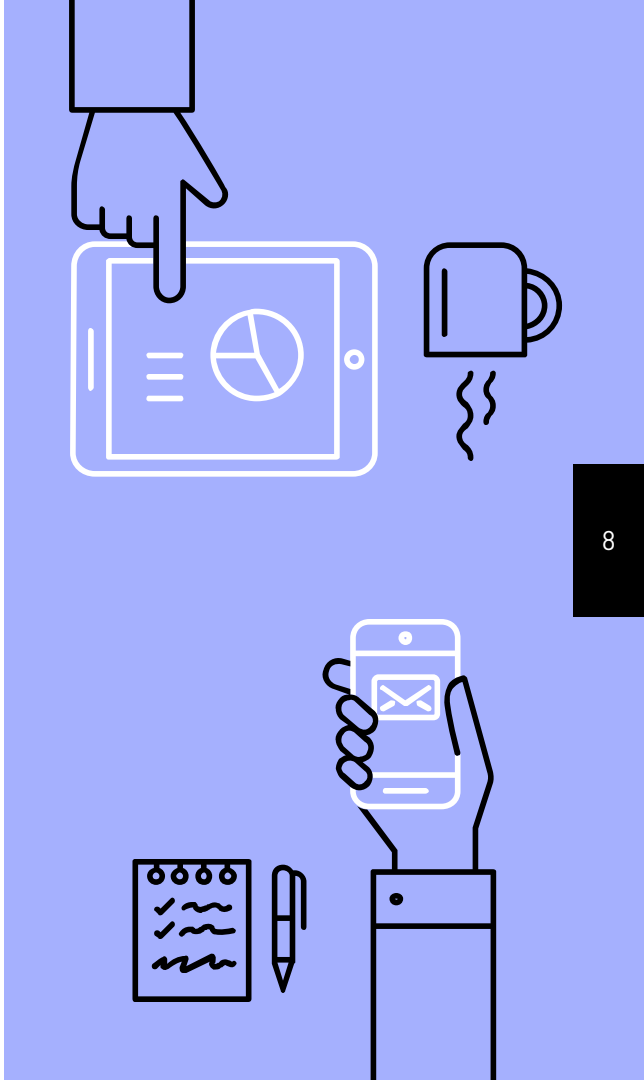
Provisioning and managing the environment in which your software will run

Installing the software and its dependencies

Installing the correct version of the software into the infrastructure

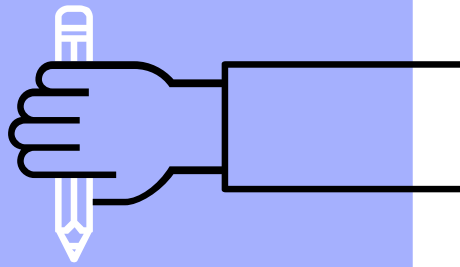
Setting up configuration for software

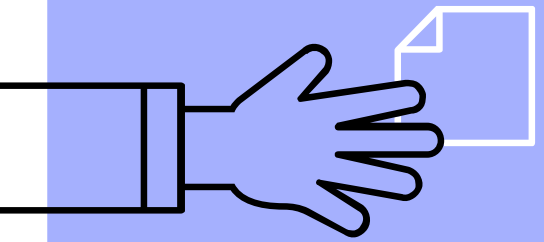
Configuring your software and database, including any data or state it requires



The key in such a transition to continuous delivery is to expect things to get worse before you'll be able to make them better." — Matthias Marscha

2. Automate, if possible, everything

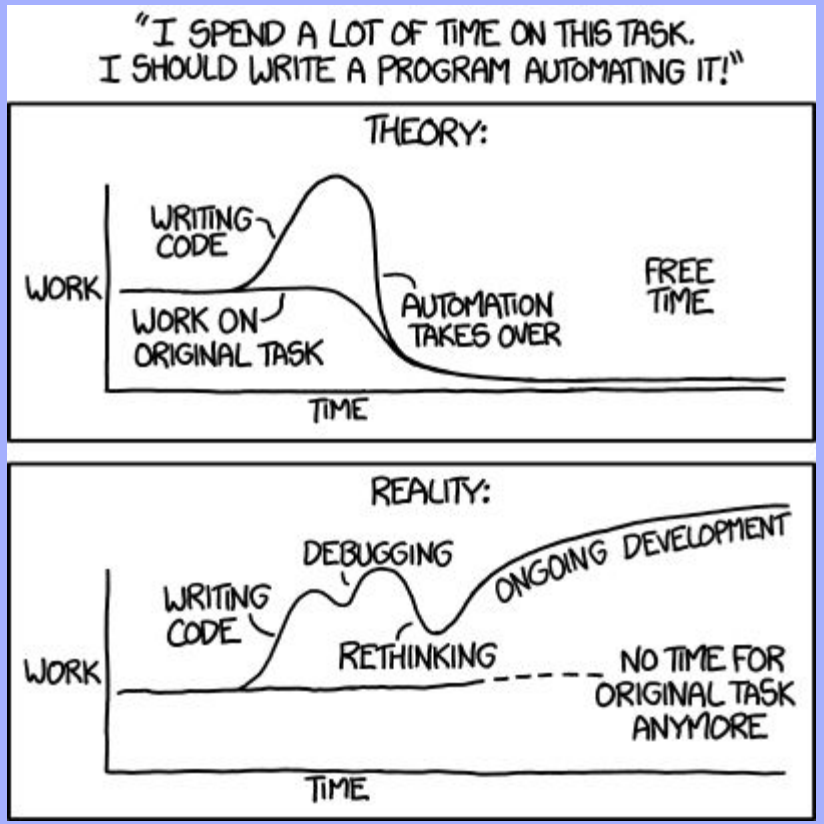




“The most powerful tool we have as developers is automation” - Scott Hanselman

- Not automating -creating a manual one
- Manual steps - requires communication
- Path gets wider - original set of instruction
  - Misinterpreted
  - Miscommunicated
  - Not maintained

- Computers - better at running repetitive task accurately
- Scripts to automate any manual process in our release process
- Commit automation script
- Others to collaborate
- Improve process


"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"
THEORY:
WORK
WRITING CODE
WORK ON ORIGINAL TASK
AUTOMATION TAKES OVER
FREE TIME
TIME
REALITY:
WORK
WRITING CODE
DEBUGGING
RETHINKING
ONGOING DEVELOPMENT
NO TIME FOR ORIGINAL TASK ANYMORE
TIME

What to automate?
When to stop?

- **Before we automate - ask yourself**
  - **Should go back to the drawing board**
  - **List steps - make process shorter & faster compared to manual process**

Otherwise ...

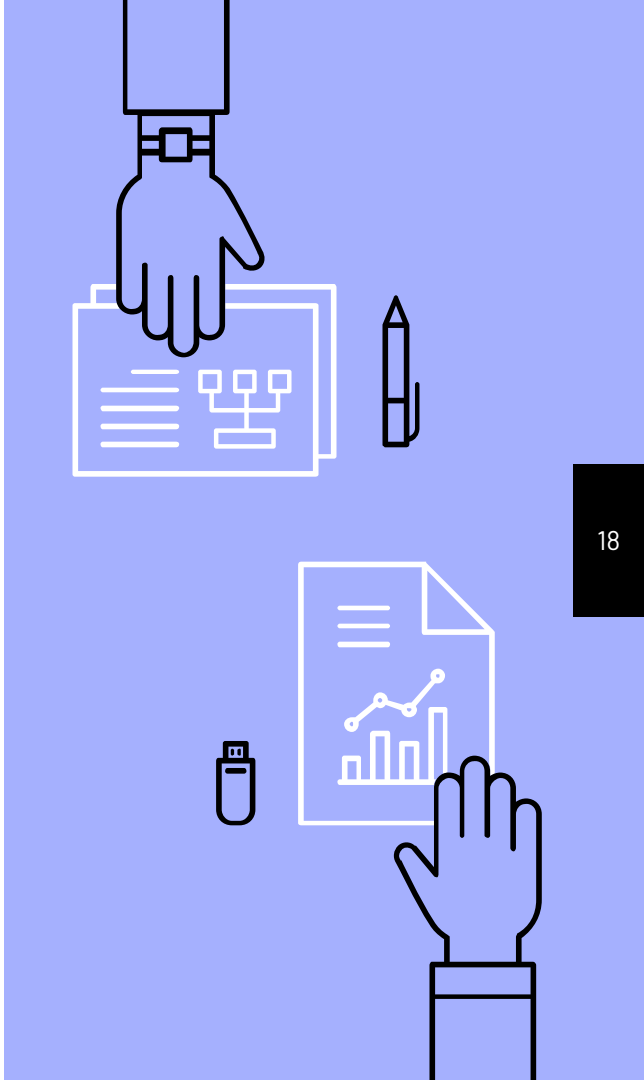
- **If you process are bad/flawed - making bad processes happen faster**

Automation will ultimately improve production velocity and quality.


3. Keep
everything
under version
control

Version Control

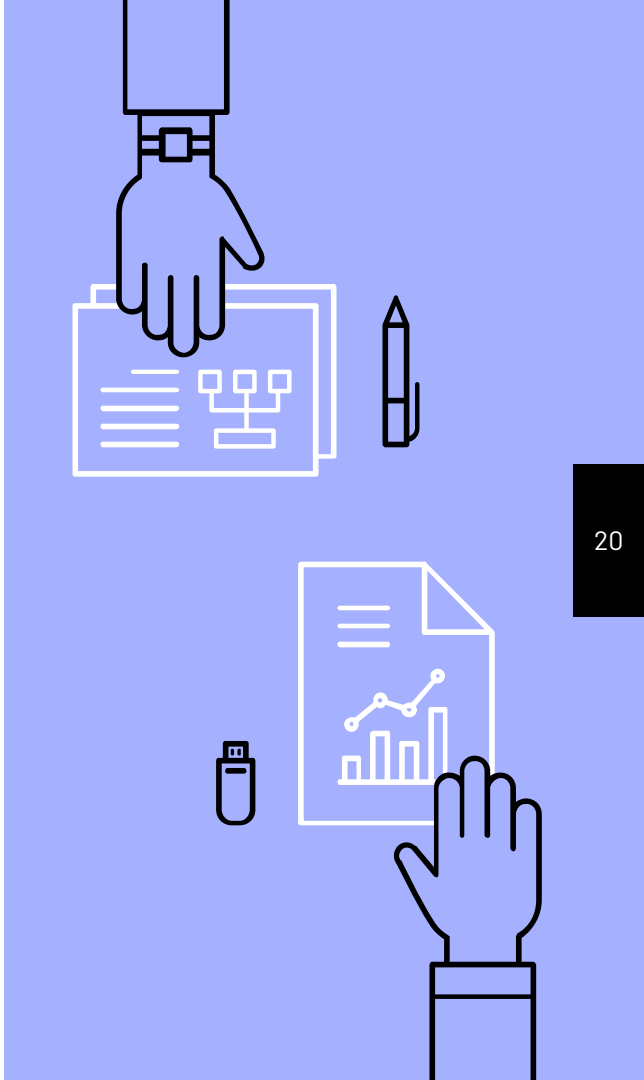
- Everything - not just software code
- Not about having backups (side effect)
- About documenting & annotating -process of creation



Code that is not annotated with its history and its motivations suffers to an extreme degree of entropy, becoming more and more difficult to understand.

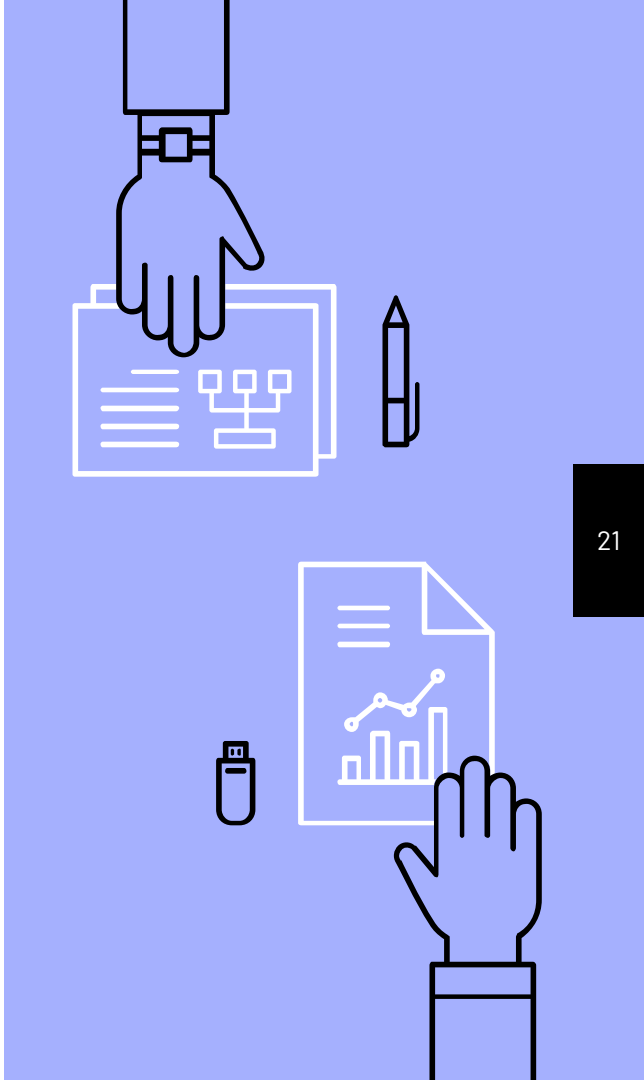
Configuration

- Lifecycle completely different than of code
- Sensitive information don’t checkin
- Config information
  - Modular
  - Encapsulated
  - No knock-on effect - unrelated configuration



Configuration

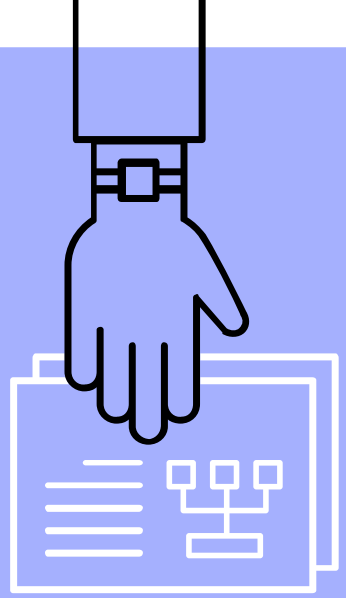
- Be minimalistic (simple & focused)
  - “Requirement” document
  - Deployment script
  - Db creation
  - And other similar examples
- Relevant version should be identifiable for any given build
  - Build number/ change set number – that references every piece



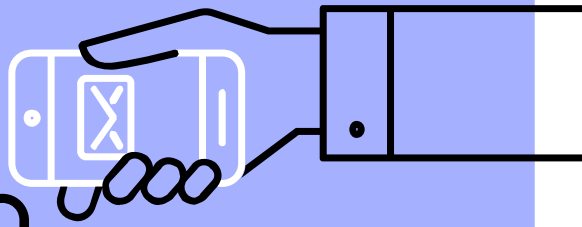

ASK YOURSELF

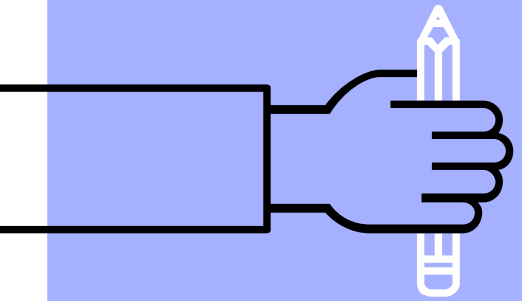
Could you completely re-create your live environment (like production), excluding production date, from scratch with the version-controlled assets that you store?

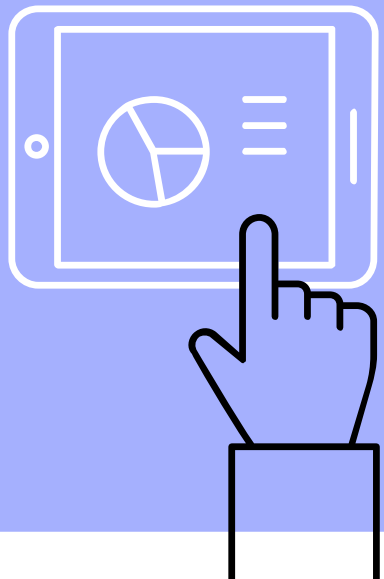
Can you regress to an earlier, known good state of your application?



4. If it hurts, do it more often — bring the pain forward



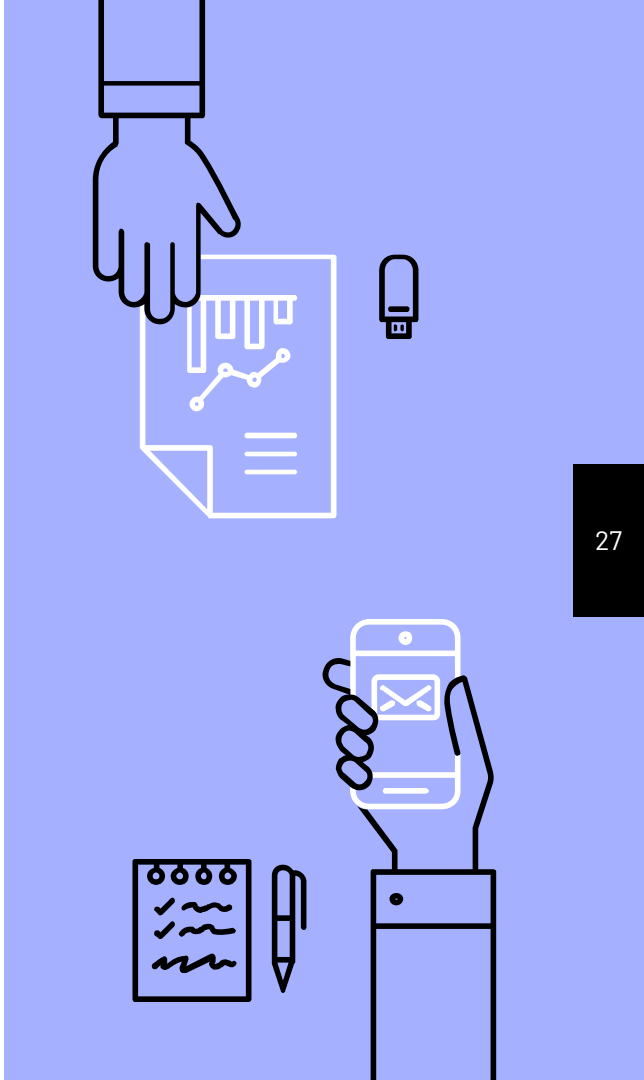


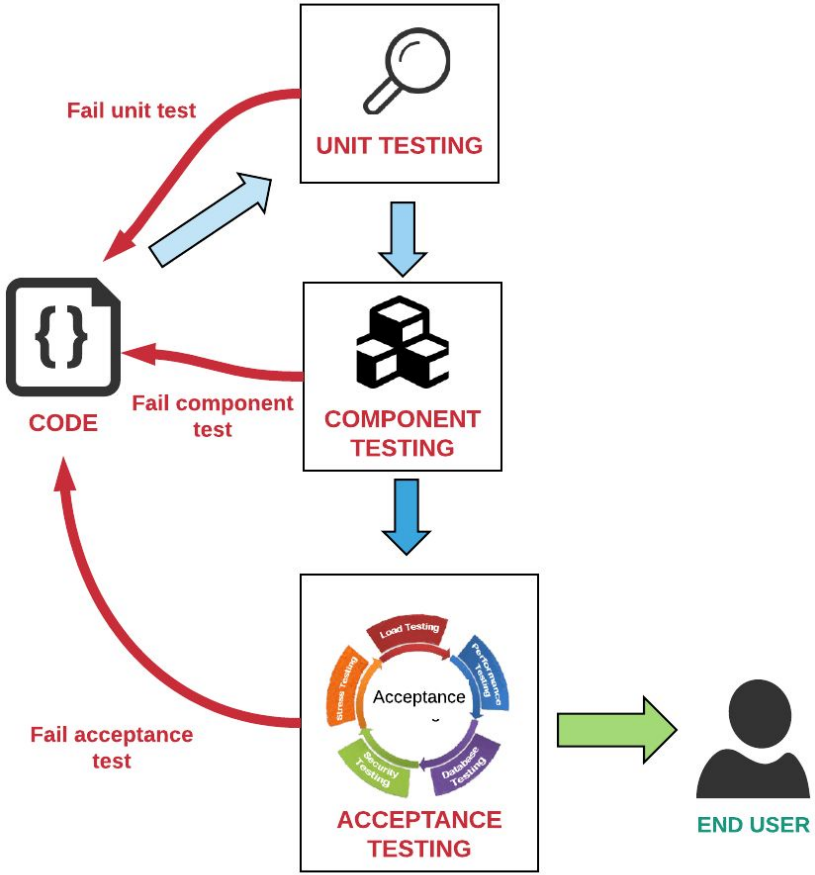


- Don’t do it at the end
- Do it continually from the beginning
- Application and build scripts need testing
- So do your configuration settings
- References to external - are good

At a minimum

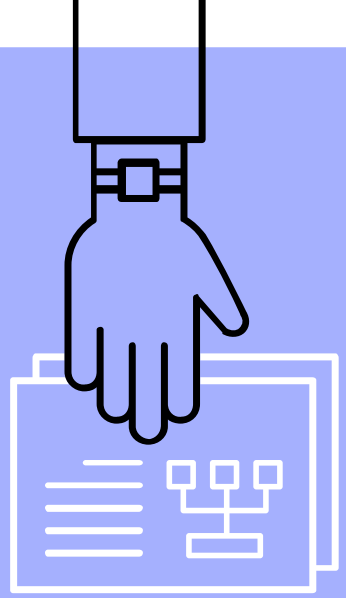
- Unit Test
  - Run periodically
  - Often after every change
  - Sooner - catch problem
- Component Test
  - specific module or program
  - isolation from rest of the system depending on the life cycle model
  - like unit testing - uses real data instead of dummy data for testing
- Acceptance Test
  - criteria decided by the business
  - capacity, functionality, availability, scalability, and security

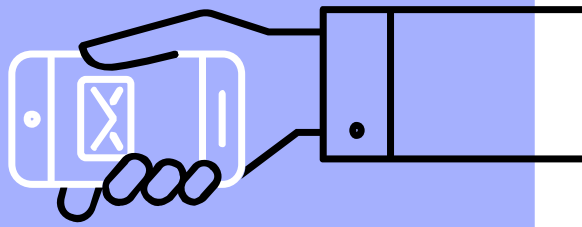



Fail unit test
UNIT TESTING
CODE
Fail component test
COMPONENT TESTING
Load Testing
Performance Testing
Stress Testing
Acceptance
Security Testing
Database Testing
Fail acceptance test
ACCEPTANCE TESTING
END USER

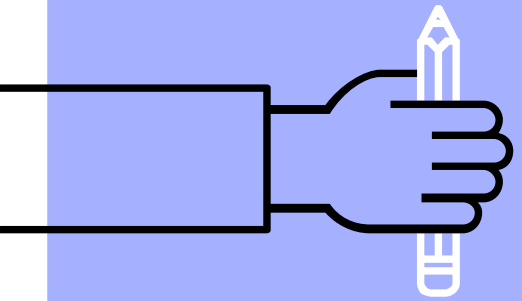
Don’t comment out failing test. Delete it if it’s not relevant or refactor the test.

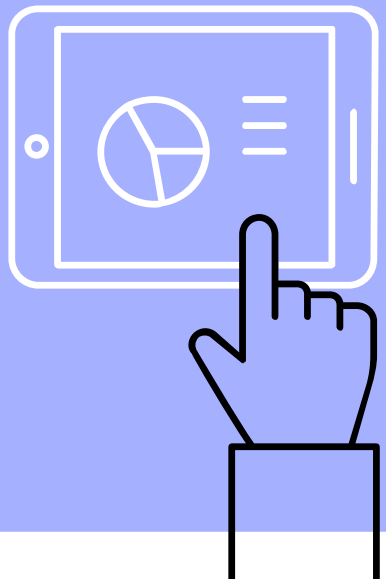
History of Software Testing





5. Keep the build and test process short



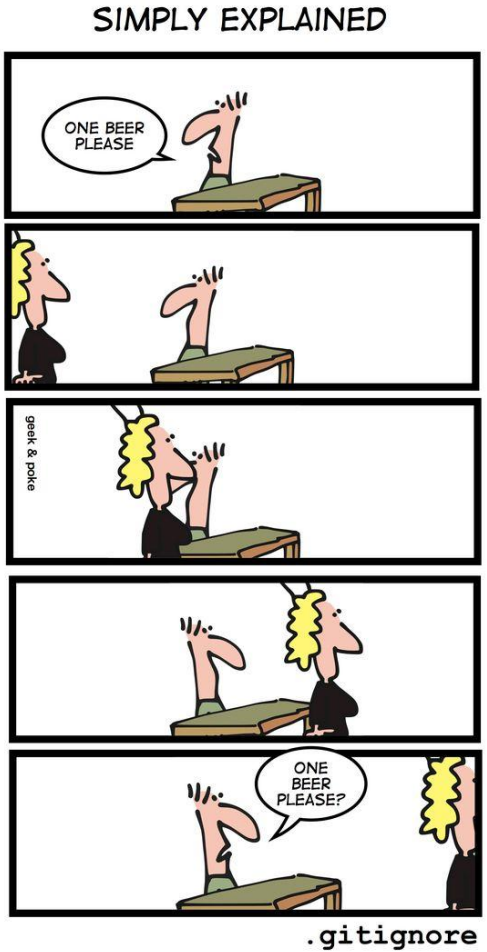


- Longer - development team has to make incorrect assumption
- longer it will take them to fix it
- Speed is essential
  - opportunity cost associated with not delivering software
- Verify whether - features & bug fixes are really useful
- Important part of usefulness is quality
  - fit for its purpose
  - Quality does not equal perfection

“The perfect is the enemy of the good”—Voltaire.

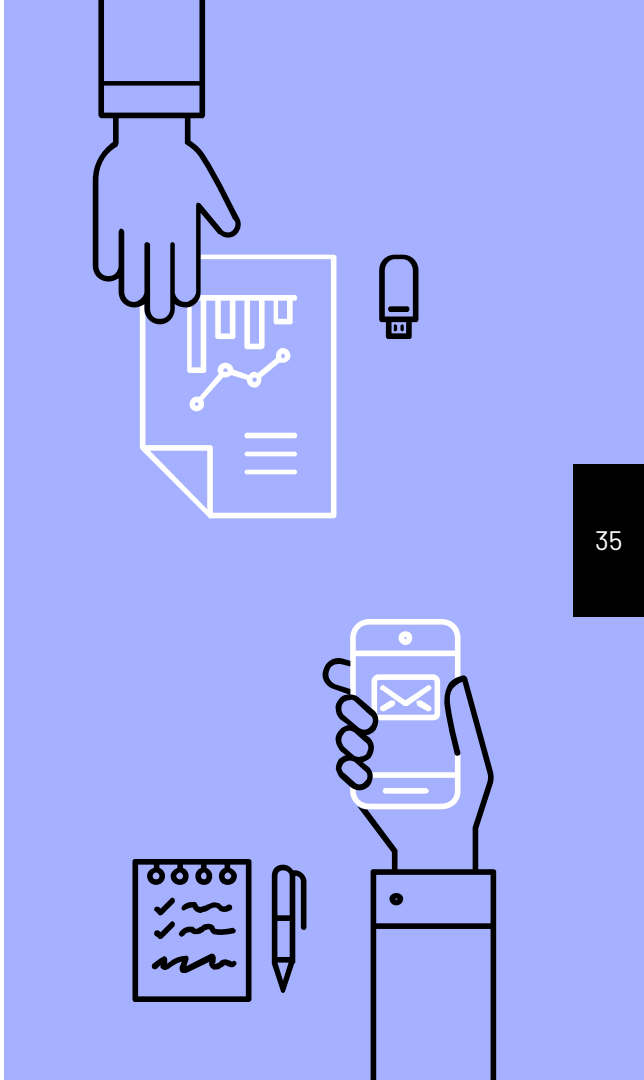
Our goal

- Deliver software of sufficient quality
  - Bring value to its users
- Check-in frequently
- Least check-in your code a couple of times a day



If the build takes too long

- Multiple commits would have taken place
  - won't know which check-in broke the build
- Build output
  - inform - commit that triggered the build
- Developers will check in less often
  - have to sit around for ages waiting - build, test and deploy



WAIT

- Tests to pass before moving forward
- Don’t compound the failure - more problems

Remember

- Remember, build a little, test a little, deploy a little.

Don’t

- Check-in on a broken build
- Cardinal sin of continuous integration
- If continue -
  - much longer for the build to be fixed
  - added more complexity to the problem

Test locally on your machine before checking in your code.

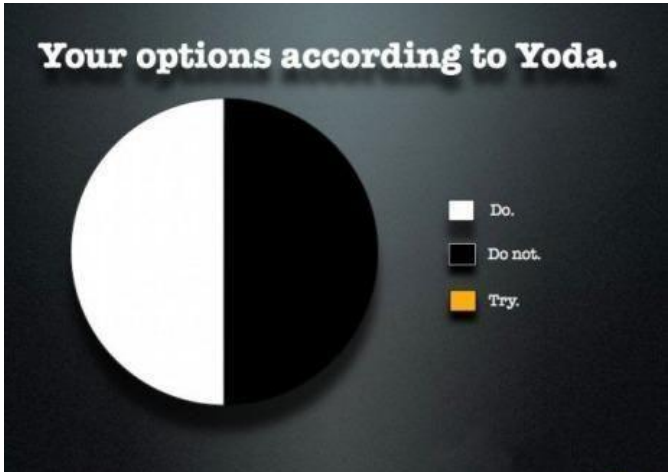
Never go home on a broken build

For example, at 5:30 pm on a Friday, all your colleagues are leaving the office and you are itching to leave as well. You decide to check-in your code and leave. Your checked in code triggers a build and it fails. What do you do next?

You have two options:

- Revert the commit to a working version
  - Always be prepared to revert to the previous revision

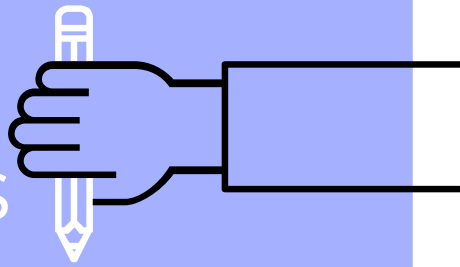
- Fix the broken test and build before you leave the door

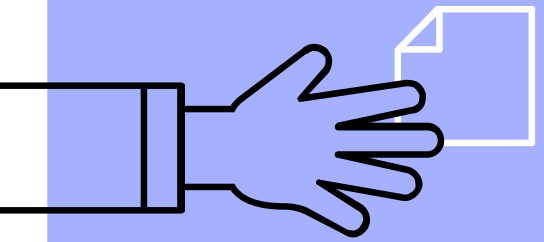


If the build is broken...

- Feedback loop should be fast and efficient
- Notify the appropriate people - look at build
- Heart of any software delivery process
- Best way to improve feedback
  - feedback cycle short and the result visible

6. Make every part of the process of building, deploying, testing and releasing software visible to everybody involved





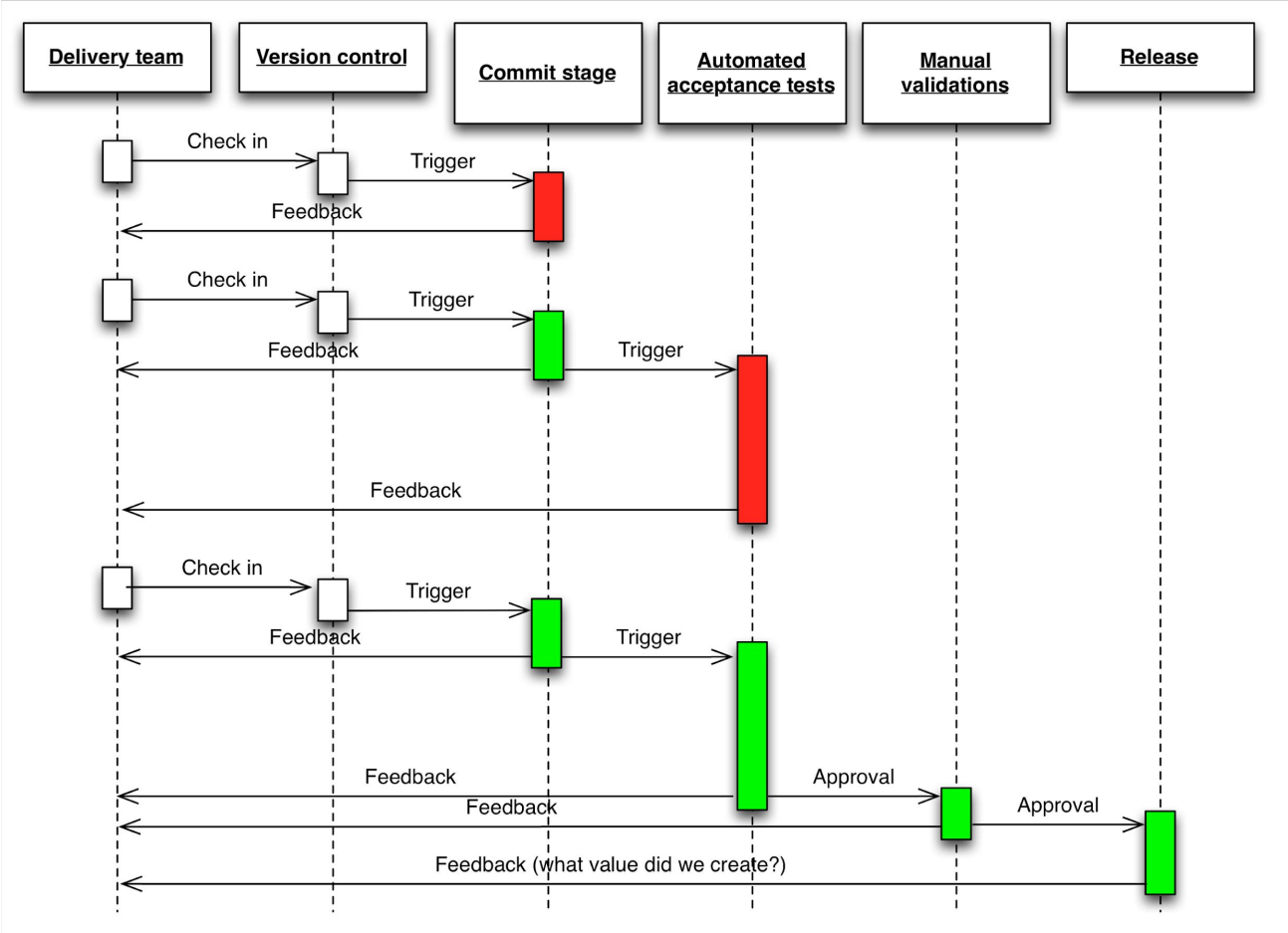
Improve Feedback

- Improve feedback so that problems
  - Identified,
  - Resolved
  - As early as in the process as possible
- Enable teams to deploy and release any version - software
  - any environment at will through a fully automated process

Who acts on feedback...

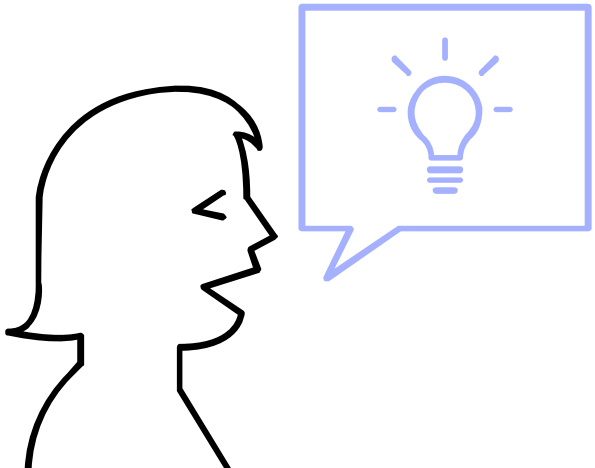
- The entire team - essential that everybody - process of delivering software
  - Software developers
  - Release engineers
  - DevOps engineers
  - Database administrators
  - Security team
  - Tester
  - Product managers.

Often the poor collaboration that causes so many problems in deployment to staging is shored up with ad-hoc phone calls, emails, and quick fixes


Delivery team
Version control
Commit stage
Automated acceptance tests
Manual validations
Release
Check in
Trigger
Feedback
Check in
Trigger
Feedback
Trigger
Feedback
Check in
Trigger
Feedback
Trigger
Feedback
Approval
Feedback
Approval
Feedback (what value did we create?)

Any changes or commits should trigger a feedback process

- Delivered as soon as possible
- Key? - fast feedback is automation (see point 2)
- People are expensive and valuable
  - focused on producing software.

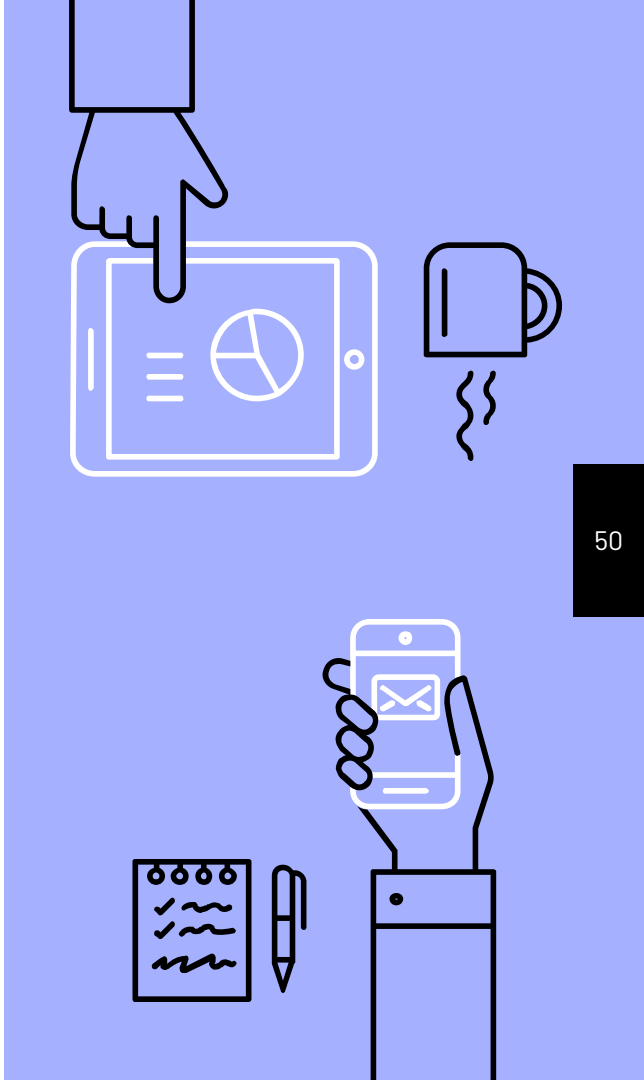



7. Done Means
Released

“done” doesn’t really mean “DONE!”

- X necessarily tested
- Certainly X styled
- X accepted by the product owner

It just means developed!



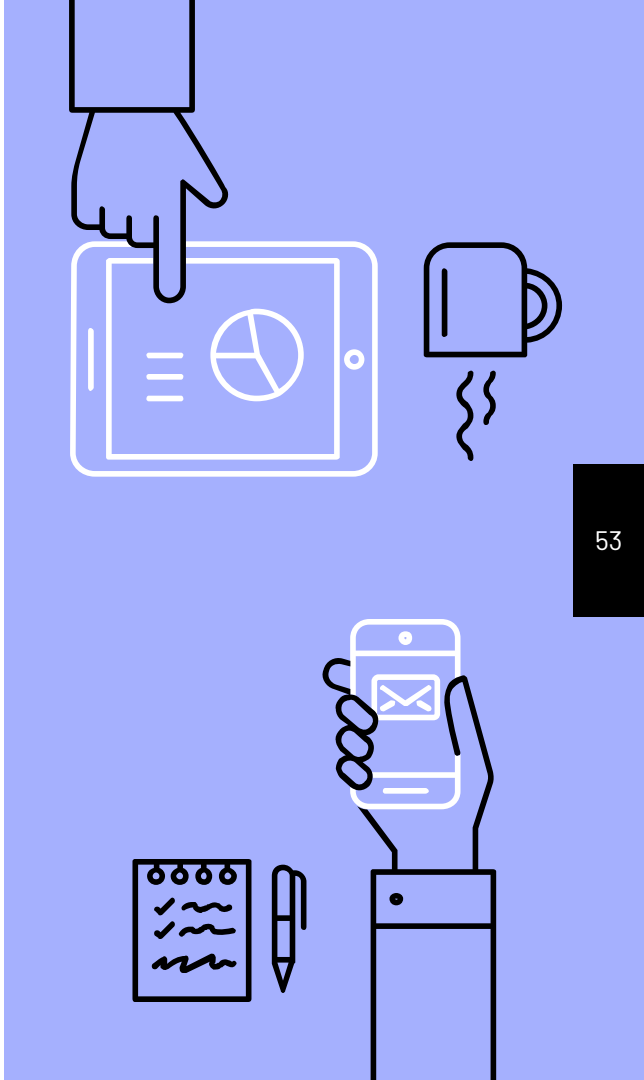
- Complete each feature before moving on to the next
- Multiple features can be developed in parallel - team situation
- Within the work of each developer
  - do not move on to a new feature until the last one is shippable
- Shippable state at the end of the Sprint
  - *not in a state where multiple features are 90% complete or untested*

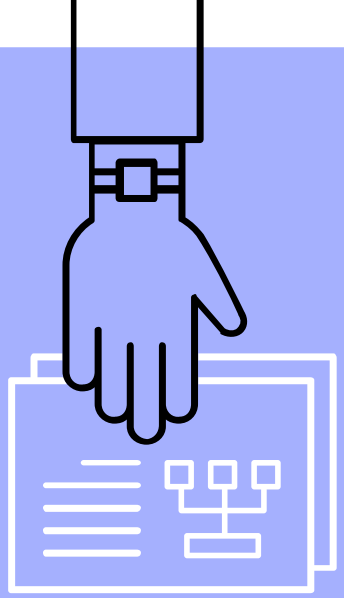
We should make sure that each feature is fully developed, tested, styled, and accepted by the product owner before counting it as "DONE!". And if there's any doubt about what activities should or shouldn't be completed within the Sprint for each feature, "DONE!" should mean shippable—by Kelly Waters

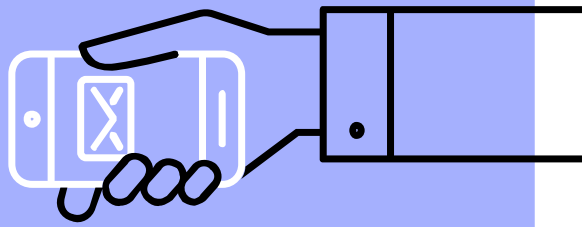
What done means?

- Monitoring is set up and working
- On-call schedule
- End users should be able to open a ticket to report a bug
- Software is well tested
- Scale up to handle high load
- Software is secure

DONE means shippable and functional.

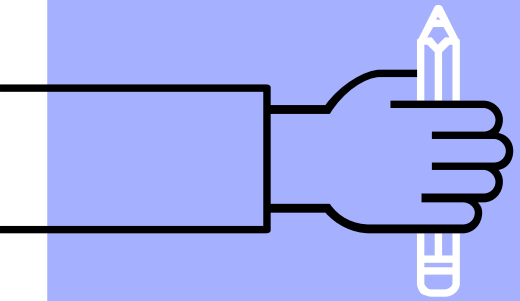


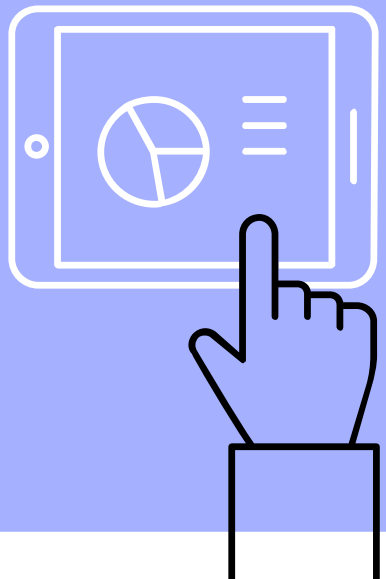




Conclusion

“Continuous is more often than you think”—Mike Roberts





THANKS!

Any questions?

You can find me at:

- warren.veerasingam@gmail.com
- At my desk