

# Deployment

Yudistira Asnar

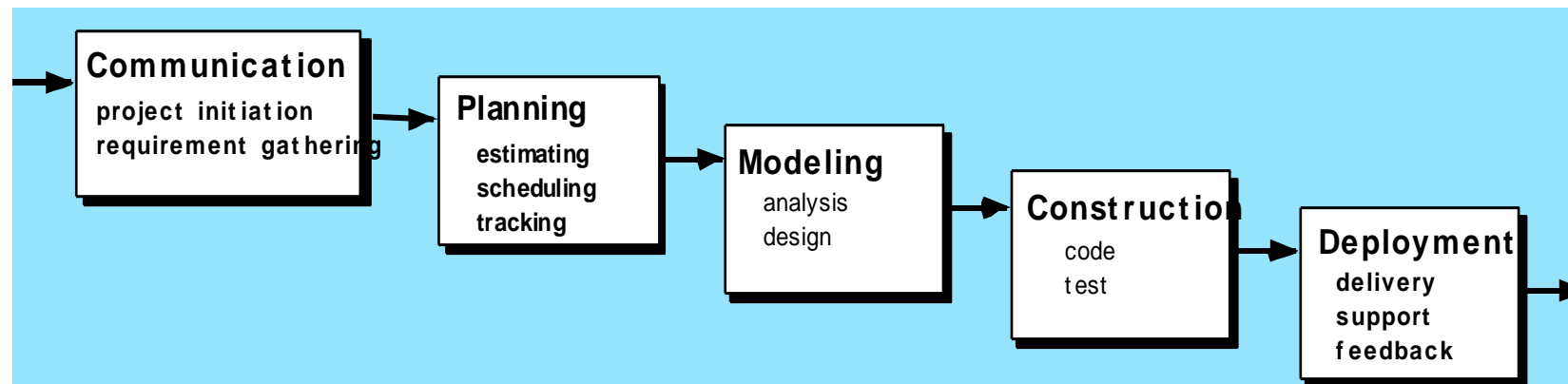
yudis@staff.stei.itb.ac.id

# Objective

- Students understand the role of deployment of a software systems
- Students understand CI/CD

# Deployment – in The Waterfall Model

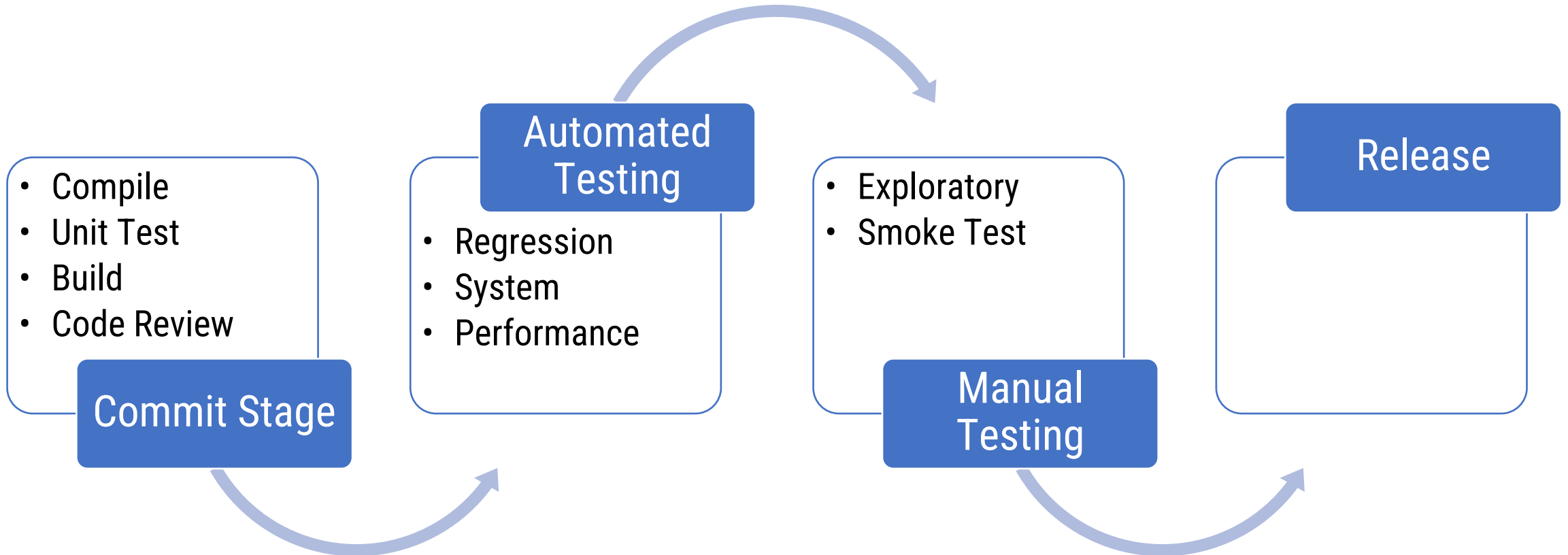
- Deployment
  - The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation



# Deployment Principles

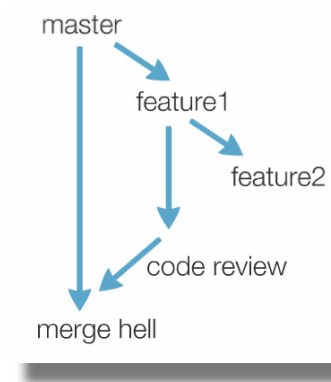
- **Principle #1. *Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately.
- **Principle #2. *A complete delivery package should be assembled and tested.***
- **Principle #3. *A support regime must be established before the software is delivered.*** An end-user expects responsiveness and accurate information when a question or problem arises.
- **Principle #4. *Appropriate instructional materials must be provided to end-users.***
- **Principle #5. *Buggy software should be fixed first, delivered later.***

# Deployment Pipeline



# Developer Now

- Each developer has feature branches
  - *If* the version control is used at all
- Features are deployed when completed
- Integration issues
- Small test suite



# Problems

- Code keeps evolving
  - Feature development
  - Bug fixing
  - Testing, etc.
- Bringing software into production is hard
- Takes a lot of time
- Error prone

# Problems (2)

- Manual deployment...can lead to
  - Operation teams waiting for documentation or fixes
  - Tester waiting for “good” builds of the software
  - Developer receiving bug reports weeks after, and s/he has move on to a new functionality
  - The software architecture hinders the system testing (non-functional reqs.)

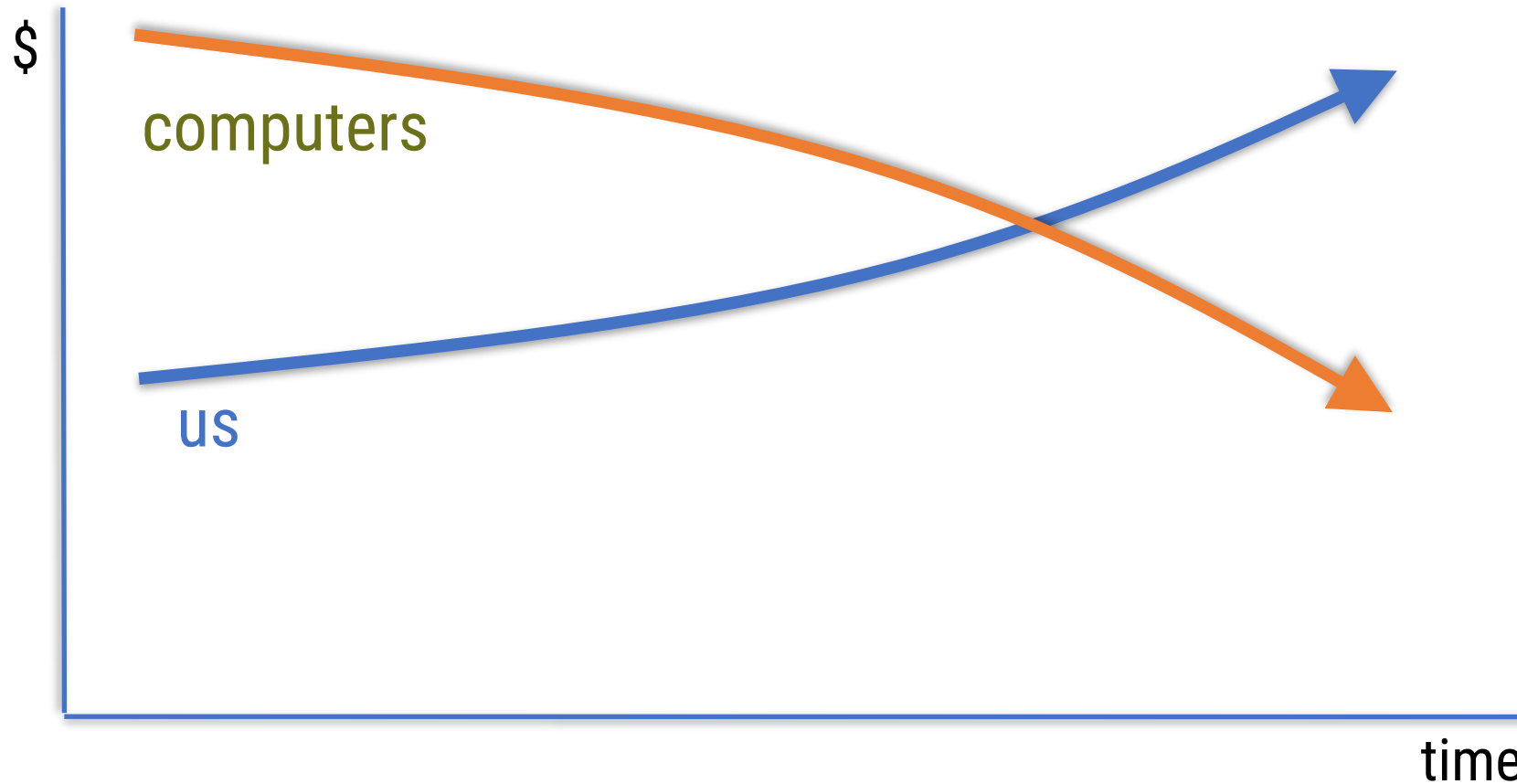


# Some Antipatterns on Deployment

- Deploying software manually
- Deploying to a production-like environment ONLY after development is COMPLETE
- Manual configuration management of production environments

# Rise of Continuous Integration

- Offload from people, push to computers



# Spend more CPU power to help you

- ... even if it only helps a little
- First on your laptops and workstations
  - IDEs are at the forefront
- And then to the servers
  - a.k.a. “Continuous Integration”
  - More frequent build/test executions
  - Static code analysis tools
  - And more to come

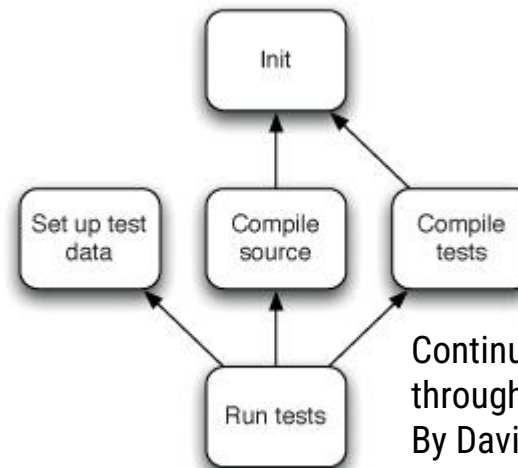
# Approach on Deployment

- Automate
  - Repeatable, consistent
- Frequent
  - Small delta between release
  - Easier to rollback
  - Faster feedback
- Every change should trigger the feedback process

# Build Tools

- Toolkit to prepare the workable software
- It comprises into several steps
  - Dependency
  - Goal-oriented (target)
- Example
  - Make in C/C++
  - Ant, Maven in Java
  - Gradle in Java, Groovy

- Best practices
  - Use the same script to every env
  - Use “standards” toolset
    - OS, Framework-centric



Apps / services / components	Application configuration
Middleware	Middleware configuration
Operating system	Operating system configuration
Hardware	

Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation  
By David Farley, Jez Humble

# CI/CD

Prepared by Satrio Adi Rukmono

Revised & Presented by Yudistira Asnar

# CI/CD in a nutshell...

- **Continuous Integration:** automation process for developers. Code changes are regularly built, tested, and merged to a shared repository.
- **Continuous Delivery:** automates the release of validated code to a repository, which can then be deployed quickly and easily.
- **Continuous Deployment:** automatically releasing a developer's changes from the repository to production, where it is usable by customers.



source: [www.redhat.com](http://www.redhat.com)

IN SHORT,  
CI/CD is a **process**, often visualized as a pipeline,  
that involves adding a high degree of ongoing **automation**  
and continuous **monitoring** to app development.



# Continuous Integration

# Continuous Integration – Definition

*“Continuous Integration is a software development practice where members of a team **integrate** their work **frequently**, usually each person integrates **at least daily** - leading to multiple integrations per day. Each integration is **verified** by an **automated build** (including test) to detect integration errors as quickly as possible.”*

*- Martin Fowler*

# Continuous Integration

- A development methodology that involves frequent integration of code into a shared repository.
- The integration may:
  - occur several times a day,
  - be verified by automated test cases and build sequence.  
(typically unit & integration tests)

# Benefits of Continuous Integration

## 1 Early Bug Detection

If there is an error in the local version of the code that has not been checked previously, a build failure occurs at an early stage.

Before proceeding further, the developer will be required to fix the error.

This also benefits the QA team since they will mostly work on builds that are stable and error-free.

# Benefits of Continuous Integration

## 2 Reduces Bug Count

Bugs are likely to occur in any application development lifecycle.

With CI and CD, the number of bugs is reduced a lot.

(Depends on the effectiveness of the automated testing scripts.)

Overall, the risk is reduced a lot since bugs are now easier to detect and fix early.

# Benefits of Continuous Integration

## 3 Automating the Process

The Manual effort is reduced a lot since CI automates build, sanity, and a few other tests.

This makes sure that the path is clear for a successful continuous delivery process.

## 4 The Process Becomes Transparent

The team gets a clear idea when a test fails, what is causing the failure, and whether there are any significant defects.

Enables the team to make a real-time decision on where and how the efficiency can be improved.

# Benefits of Continuous Integration

## 5 Cost-Effective Process

Low Bug Count + Reduced Manual Testing Time + Increased System Clarity = Optimized Budget.

## 6 Integration is less scary steps and become enforcer

Regular integration leads to success and codes becomes compiles and ready to deploy

- In ye olde days...
  - (“waterfall” era)
  - Code were integrated during “integration phase”, separate from coding phase.
  - The phase came after programmers spent weeks/months/years working separately(!!!)
  - This *painful* integration phase can take weeks or even months.



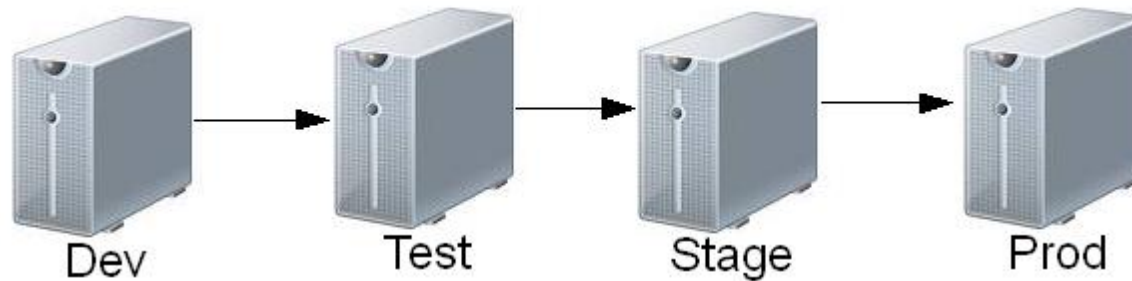
- Along came Extreme Programming and agile methods...
  - Integration became frequent.
  - Developers integrate ASAP, usually after a unit (class, module) is complete.
  - Integration is done on shared code repository.
  - Frequent integration became automated and continuous, prompting the need for **checks** before new code is integrated.
  - *Thus CI was born...*

# CI's Principles

- Environments based on stability
- Maintain a code repository
- Commit frequently and build every commit
- Make the build self-testing
- Store every build

# Environments based on stability

- Create server environments to model code stability
- Promote code to stricter environments as quality improves.

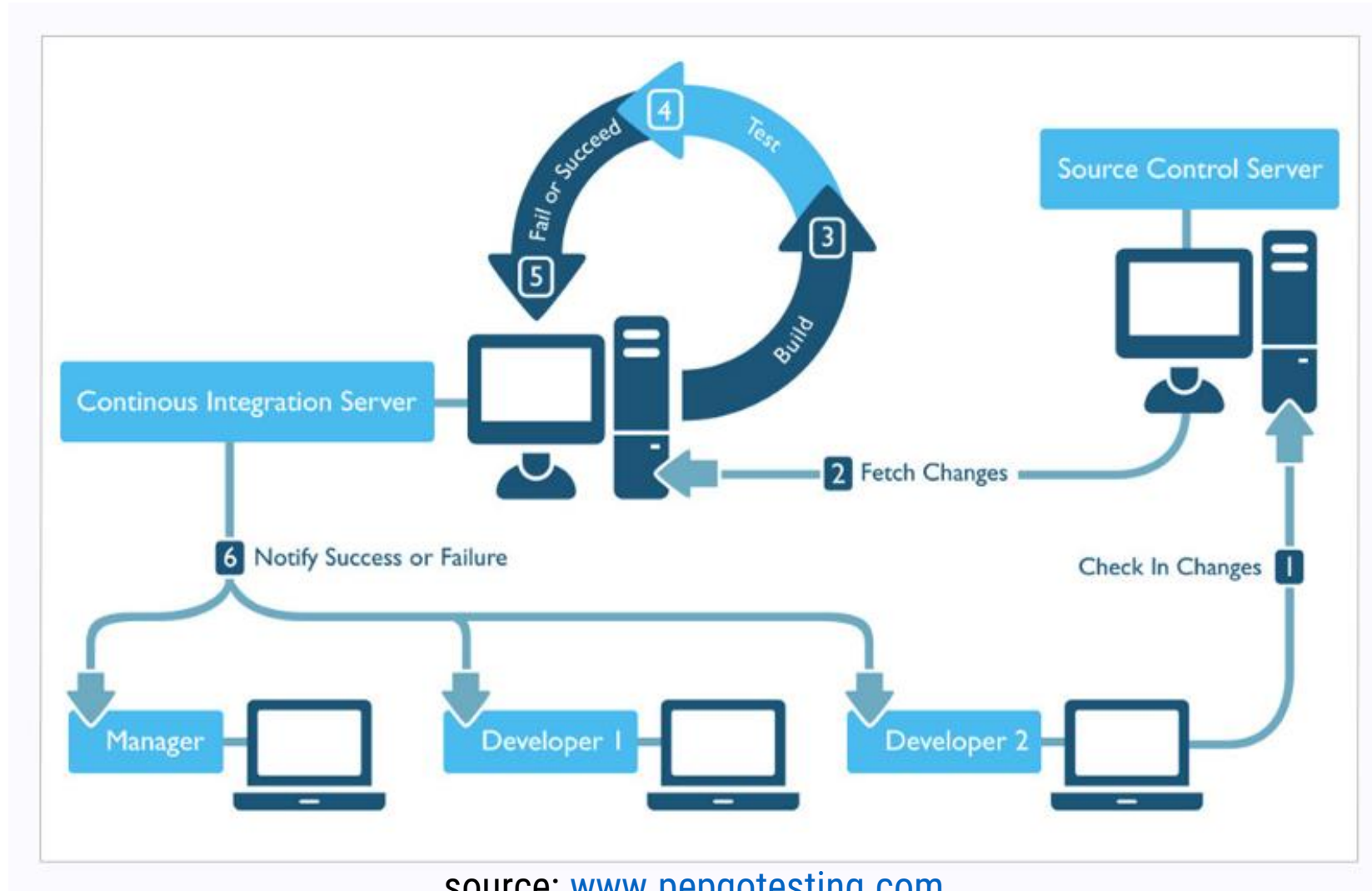


# Continuous Testing

- Automate everything
  - If it hurts, do it more often. Continuously.
  - Fail fast.
- Integration testing
- Unit testing
- System Testing
- Acceptance Testing

Whatever you don't test against, **will** happen

# A typical CI process



source: [www.pepgotesting.com](http://www.pepgotesting.com)

# Common steps in a CI workflow

1. Pushing to the code repository (must be built and tested locally)
  - In a branch
  - merge
2. Static analysis
3. Pre-deployment testing
4. Packaging and deployment to the test environment
5. Post-deployment testing

# Step 1: Pushing to the code repository

1. The developer builds their code on the local system that has all the new changes or new requirements.
2. Once coding is completed, the developer needs to write automated unit testing scripts that will test the code. (This process can be done by the testing team as well.)
3. A local build is executed which ensures that no breakage is occurring in the application because of the code.
4. After a successful build, the developer checks if any of his team members or peers have checked-in anything new.
5. If there are any incoming changes, they should be accepted by the developer to make sure that the copy he is uploading is the most recent one.
6. Because of the newly merged copies, syncing the code with the main branch may cause certain conflicts.
7. In case there is any conflict, they should be fixed to make sure the changes made are in sync with the main branch.
8. The changes are now ready to be checked in. This process is known as a “code commit.”
  - Typically there is a code repository and a workflow (git-flow, etc.) for contributing new code.
  - Committing code starts the CI pipeline.
    - Often starts with static code analysis.

# Step 2: Static analysis

- Static code analysis: checking the code for possible bugs automatically without actually executing it.
- Also, to help conforming to a standard formatting and styling.
- More about static code analysis later.



# Step 3: Pre-deployment testing

- Run all tests that can be run without deploying to a server.
- This includes unit tests and maybe functional and integration tests.
- This phase is used to ensure that the change set doesn't break functionalities and works well with other parts of the code since the tests are run on the whole code base, not just the new changes.

## Step 4: Packaging and deployment to the test environment

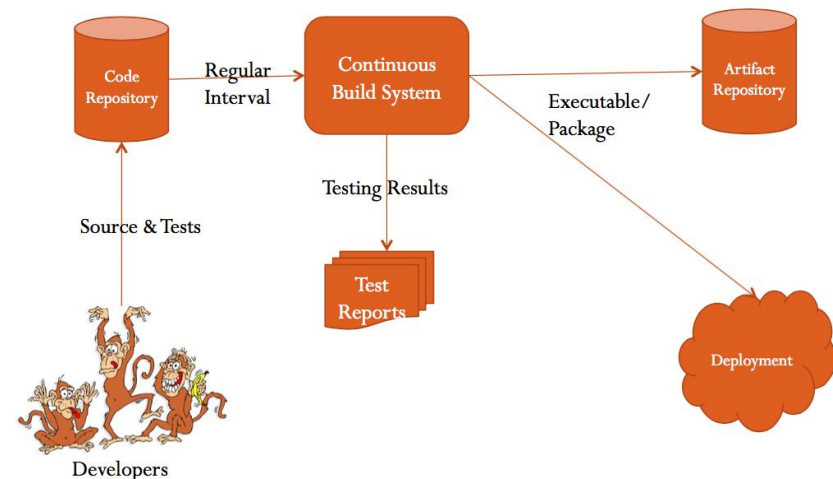
- The application is built, packaged, sent to a test or staging environment (that mimics production).
- Ensures that the integrated changes build well with other parts and can be deployed for a functional test can to be carried out.
- Verifies that the new changes are compatible with other libraries and the deployment environment.
- Also automated.

# Step 5: Post-deployment testing

- Run tests that need the application to be deployed.
- Usually functional integration and performance tests.
- Successful execution of this phase ends the CI pipeline for the changeset, signalling it's good enough for users.

# What's next?

- Once the CI pipeline completes successfully, the deployed application could undergo a manual test by a “user” or the QA team to ensure that it fits the client’s requirements.
- The packages or artefacts generated by the CI pipeline can now be taken/deployed to production.
- This can also be automated with a successful implementation of a Continuous Delivery pipeline.



# Continuous Delivery

# Continuous Delivery – Definition

“The essence of my philosophy to software delivery is to build software so that it is **always** in a **state** where it could be put into **production**. We call this *Continuous Delivery* because we are continuously running a *deployment pipeline* that tests if this software is in a state to be delivered.”

– Jez Humble, Thoughtworks

# Continuous Delivery

- Following CI, continuous delivery automates the release of validated code to a repository.
- CI must have already been built into the development pipeline.
- Goal: to have a codebase that is always ready for deployment to a production environment.
  - May include configuration changes, new features, error fixes etc.
- Every stage involves test automation and code release automation.

# Benefits of Continuous Delivery

## ① Reducing the Risk

CD makes it possible to deploy code at very low risk and almost no downtime, making deployment totally undetectable to the users.



# Benefits of Continuous Delivery

## ② High-Quality Application

Since most of the process is automated, testers now have a lot of time to focus on important testing phases like exploratory, usability, security and performance testing.

These activities can be continuously performed during the delivery process, ensuring a higher quality application.

# Benefits of Continuous Delivery

## ③ Reduced Cost

The cost of frequent bug fixes and enhancements are reduced since certain fixed costs that are associated with the release is eliminated because of continuous delivery.

# Benefits of Continuous Delivery

## ④ Happier Team, Better Product

Since the aim of Continuous Delivery is to make a product release painless, the team can work in a relaxing manner.

Because of frequent release, the team works closely with users and learn what ideas work and what new can be implemented to delight the users.

Continuous user feedback and new testing methodologies also increase the product's quality.

# How To Perform Continuous Delivery

1. Pull tested-branch
2. Build artefacts
3. Click to Deploy

# Continuous Delivery checklist

- ✓ Before any changes are submitted, ensure that the current build is successful. If there are some issues, fix the build before any new code is submitted.
- ✓ If the build is in the successful state, rebase your workspace to the configuration in which the build was successful.
- ✓ In your local system, build and test the code to check if any functionality is impacted because of the changes you made.
- ✓ If everything goes well, check in the code.
- ✓ Allow competition of continuous integration with the new code changes.
- ✓ If somehow the build fails, stop and go back to the 3rd step in the checklist.
- ✓ If the build is successful, work on your next code.

# Cont. Delivery vs. Deployment




## Continuous Deployment

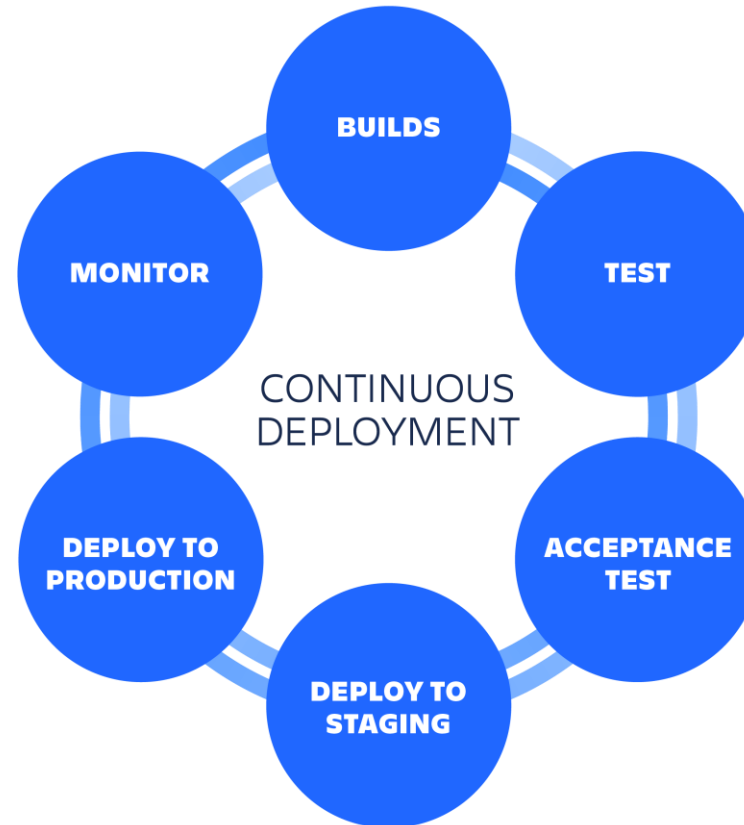


Continuous delivery is doable.  
Continuous deployment is a hard problem

# Continuous Deployment

# Continuous Deployment

- Continuous Deployment is the final stage of a mature CI/CD pipeline.
  - Automates releasing an app to production.
  - Relies heavily on well-designed test automation.
- 
- The diagram illustrates the Continuous Deployment cycle as a circular flow of five stages, each represented by a blue circle. The stages are: **BUILDS** (top), **TEST** (top-right), **ACCEPTANCE TEST** (bottom-right), **DEPLOY TO PRODUCTION** (bottom-left), and **MONITOR** (top-left). The circles are connected by a light blue ring, and the text "CONTINUOUS DEPLOYMENT" is centered within the cycle.



source: [www.atlassian.com](http://www.atlassian.com)



# What are needed?

- **Automated testing**

- Prevents any regressions when new code is introduced.
- Replaces manual reviews of new code changes.

- **Rolling deployments**

- Automated step of activating new code within a live environment.
- Ability to undo a deployment in the event that bugs or breaking changes are deployed.

- **Monitoring and alerts**

- Provides visibility into the health of the overall system and into the before and after state of new code deployments.
- Alerts can be used to trigger a rolling deployment 'undo' to revert a failed deploy.

# Continuous Deployment best practices

- Test-driven development
  - Actual deliverable code should be written to satisfy the test cases and match the spec.
- Single method of deployment
  - Developers should **not** be manually copying code to production or live editing things.
- Containerization
  - Containerizing a software application ensures that it behaves the same across any machine it is deployed on.

# Deployment strategies

- Zero-downtime deployment (and rollback or rollover)
- Deployment Schedule
  - Release when a feature is complete
  - Release every day
- Blue-green
  - Two environments
  - Install on one. Switch. Switch back on problems.
- Canary release
  - Deploy to subset of servers
- Real-time application state monitor!

# It's a wrap!

# Relevant tools to CI/CD

- Code Repositories
  - SVN, Mercurial, Git
- Continuous Build Systems
  - Jenkins, Bamboo, Cruise Control
- Test Frameworks
  - JUnit, Cucumber, CppUnit
- Artifact Repositories
  - Nexus, Artifactory, Archiva

# CI/CD tools

- **Jenkins:** Java-based, can be configured using console or GUI.
- **Team City:** cloud-based, developed by JetBrains.
- **Travis CI:** one of the oldest CI/CD tool, hosted on GitHub.
- **Gitlab**
- **Circle CI**
- *etc.*

# CI/CD best practices

- Model your value stream as the skeleton the CI/CD pipeline
- Keep a Central Repository
- Automated Deployment and Build
- Include Automated Unit Testing and Code Analysis
- Test in the Production's Clone
  - Acceptance test and Smoke-test
- Commit the Code every day, Not committing generated files
- Test target doesn't break the build
- Build Faster
- Everyone Can See What Others are Doing
- Ensure the deployment pipeline is idempotent

# Final Remarks

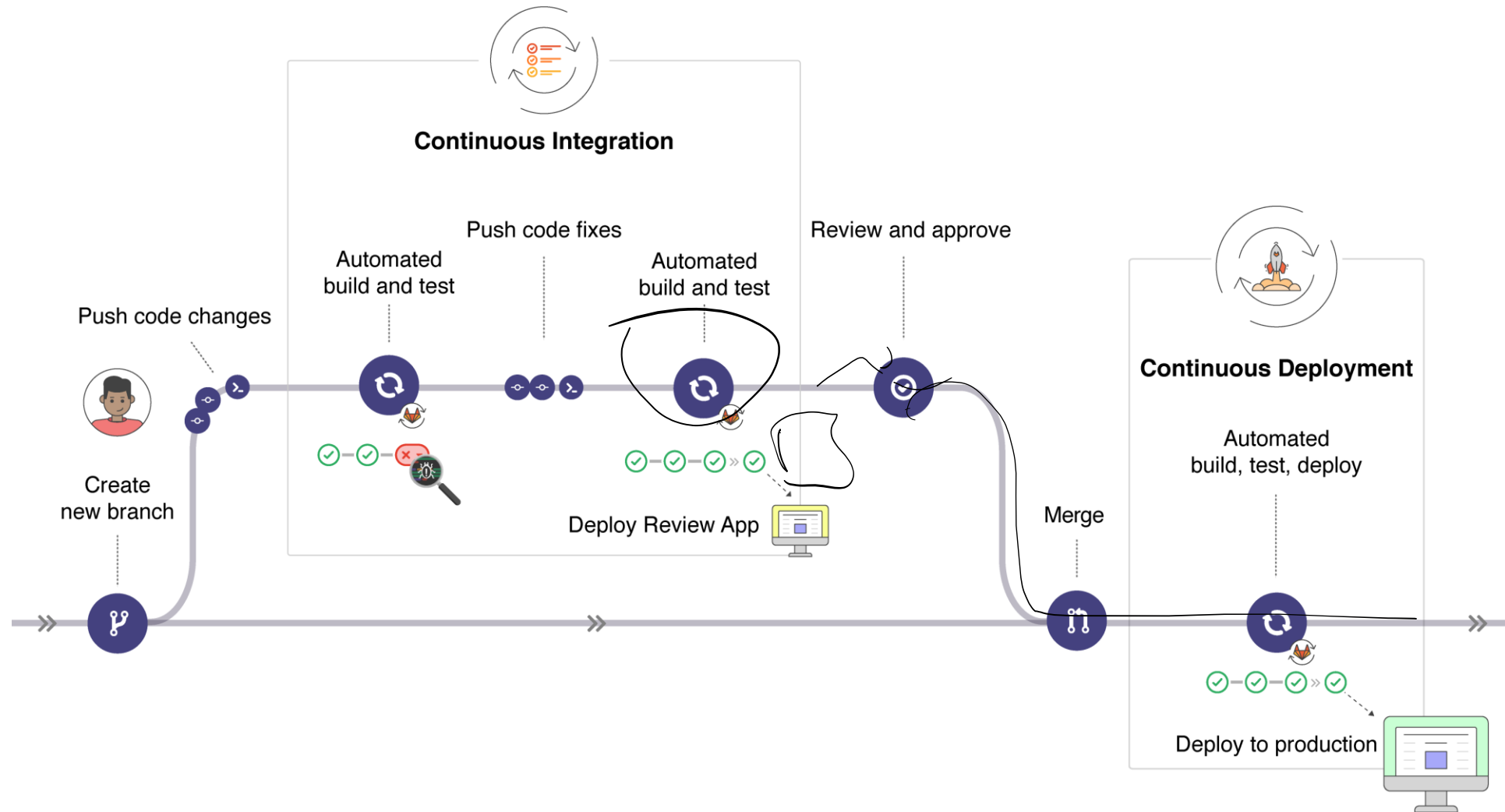
- Continuous integration is a necessity on complex projects due to the benefits it provides regarding early detection of problems
- A good continuous build system should be flexible enough to fit into pre-existing development environments and provide all the features a team expects from such a system
- Jenkins, a continuous build system, can be an integral part of any continuous integration system due to its core feature set and extensibility through a plugin system



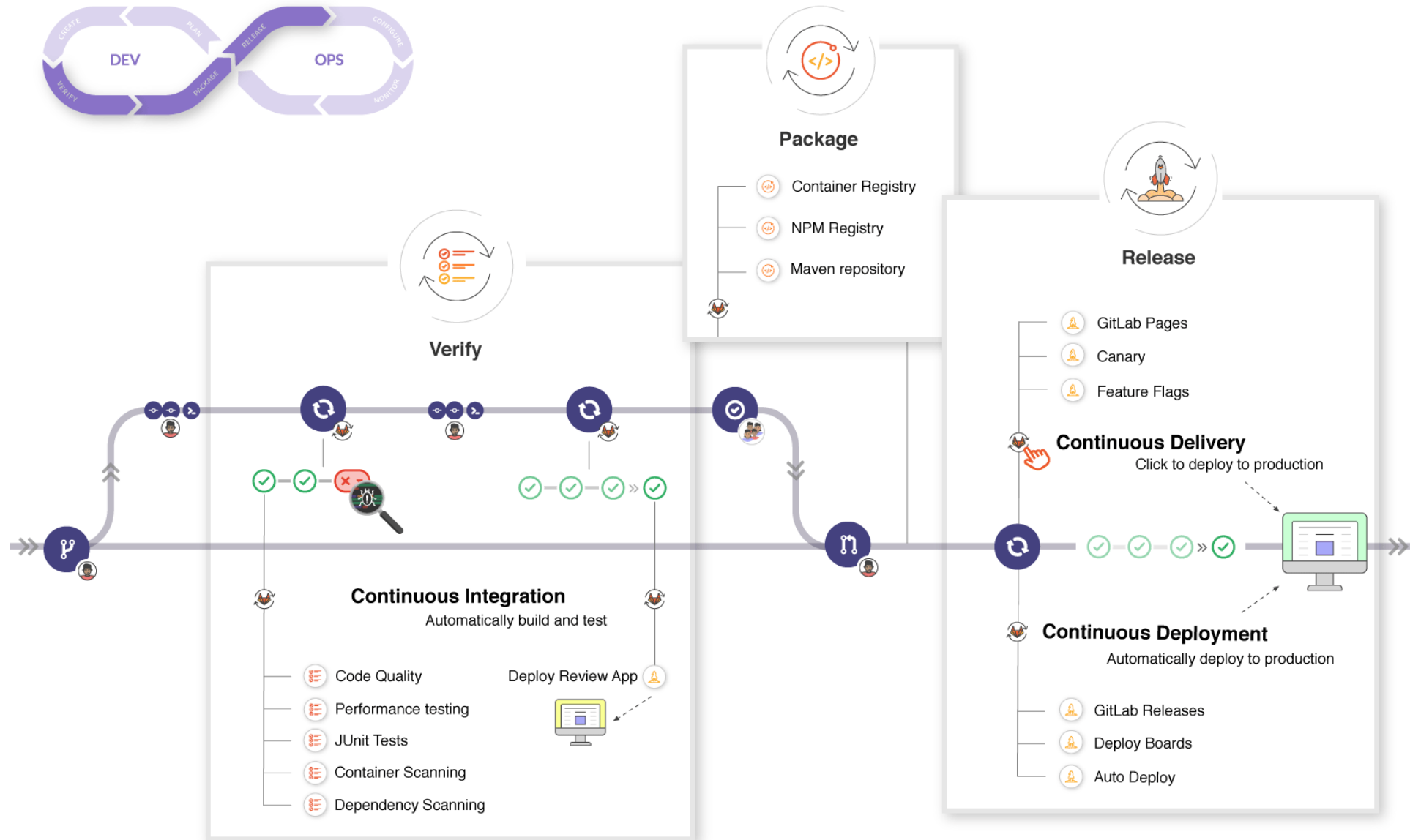
# Problems in Continuous XXX

- Technical
  - Databases
    - Schema migration
    - Revert!
    - Change management software
  - Configuration
- Human
  - Even more important
  - Automatic deployment = great fear
  - Customers don't want software to constantly change

# Gitlab CI/CD



# Gitlab CI/CD



# Exercise

- Perform CI/CD as in the following CI/CD Demo
  - <https://www.youtube.com/watch?v=1iXFbchozdY>
  - Documentation: <https://docs.gitlab.com/ce/ci/README.html>
  - <https://docs.gitlab.com/ce/ci/introduction/index.html#basic-cicd-workflow>

# References

- What is CI/CD? (<https://www.redhat.com/en/topics/devops/what-is-ci-cd>)
- Continuous Integration: A “Typical” Process (<https://developers.redhat.com/blog/2017/09/06/continuous-integration-a-typical-process/>)
- What is Continuous Integration and Continuous Delivery? (<https://dzone.com/articles/what-is-continuous-integration-andcontinuous-delive>)
- What is Continuous Deployment? (<https://www.atlassian.com/continuous-delivery/continuous-deployment>)