DEVOPS

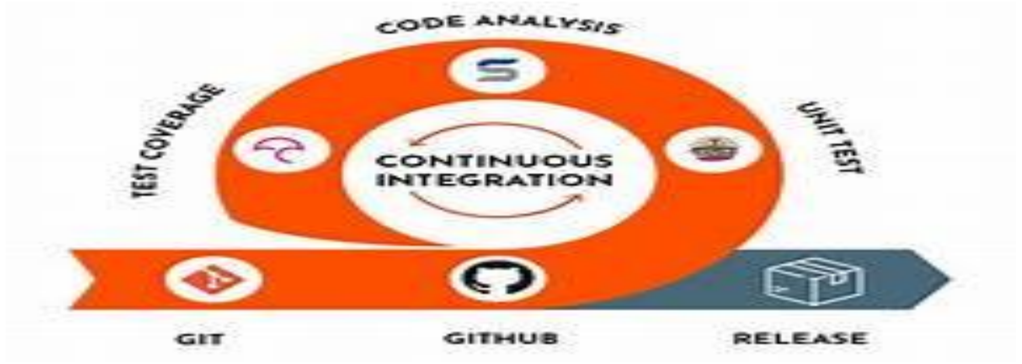
Naresh babu Jaya chand

What is DevOps?

- DevOps combines development (Dev) and operations (Ops) to unite people, process, and technology in application planning, development, delivery, and operations.
- DevOps enables coordination and collaboration between formerly siloed roles like development, IT operations, quality engineering, and security.
- Teams adopt DevOps culture, practices, and tools to increase confidence in the applications they build, respond better to customer needs, and achieve business goals faster.
- DevOps helps teams continually provide value to customers by producing better, more reliable products.

Continuous integration(CI)

- Continuous Integration (CI) is the practice used by development teams to automate, merge, and test code.

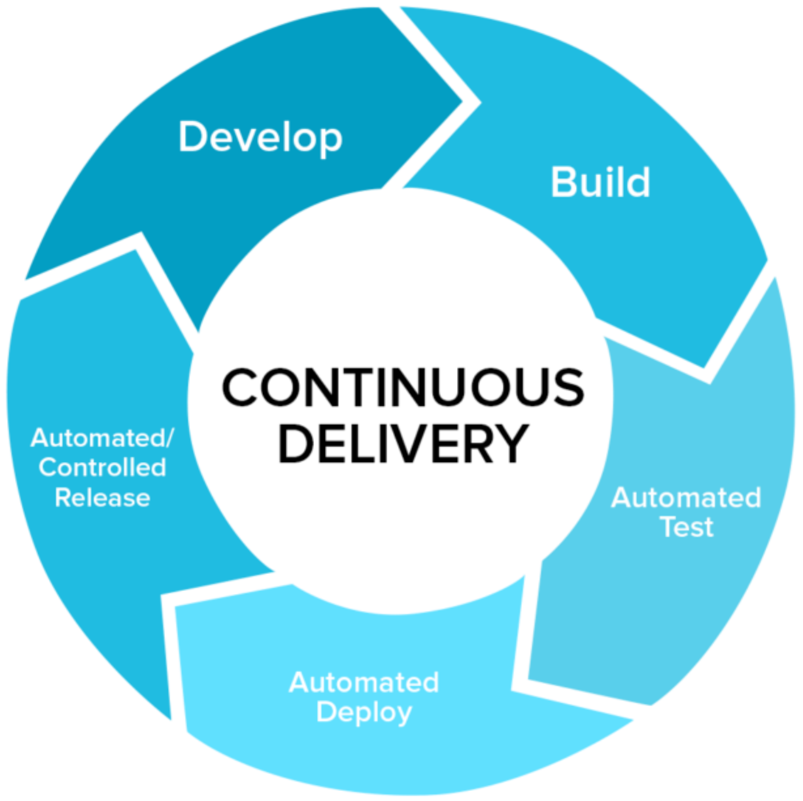


- CI helps to catch bugs early in the development cycle, which makes them less expensive to fix. Automated tests execute as part of the CI process to ensure quality.
- CI systems produce artifacts and feed them to release processes to drive frequent deployments.

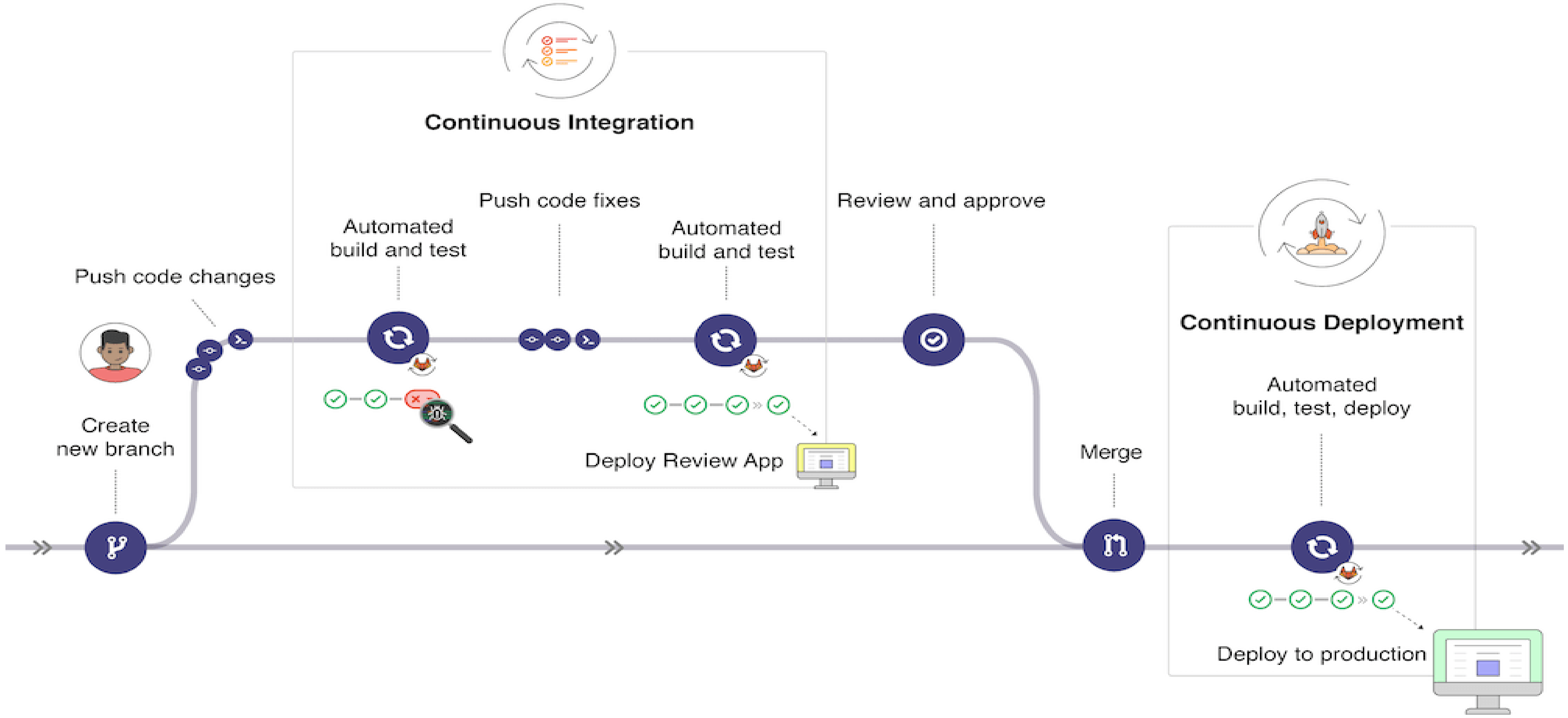
Continuous Delivery (CD)

- Continuous Delivery (CD) is a process by which code is built, tested, and deployed to one or more test and production environments.
- Deploying and testing in multiple environments increases quality. CD systems produce deployable artifacts, including infrastructure and apps.
- Automated release processes consume these artifacts to release new versions and fixes to existing systems.
- Systems that monitor and send alerts run continually to drive visibility into the entire CD process.

Continuous Delivery (CD)



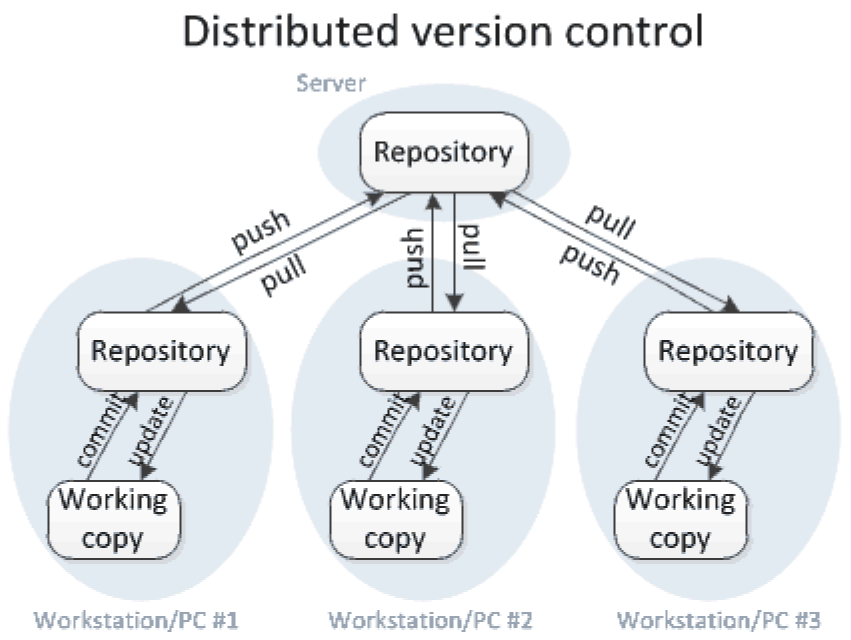
Continuous Integration & Continuous Delivery (CI/CD)



Version Control

- Version control is the practice of managing code in versions—tracking revisions and change history to make code easy to review and recover.
- This practice is usually implemented using version control systems such as Git, which allow multiple developers to collaborate in authoring code.
- These systems provide a clear process to merge code changes that happen in the same files, handle conflicts, and roll back changes to earlier states.
- The use of version control is a fundamental DevOps practice, helping development teams work together, divide coding tasks between team members, and store all code for easy recovery

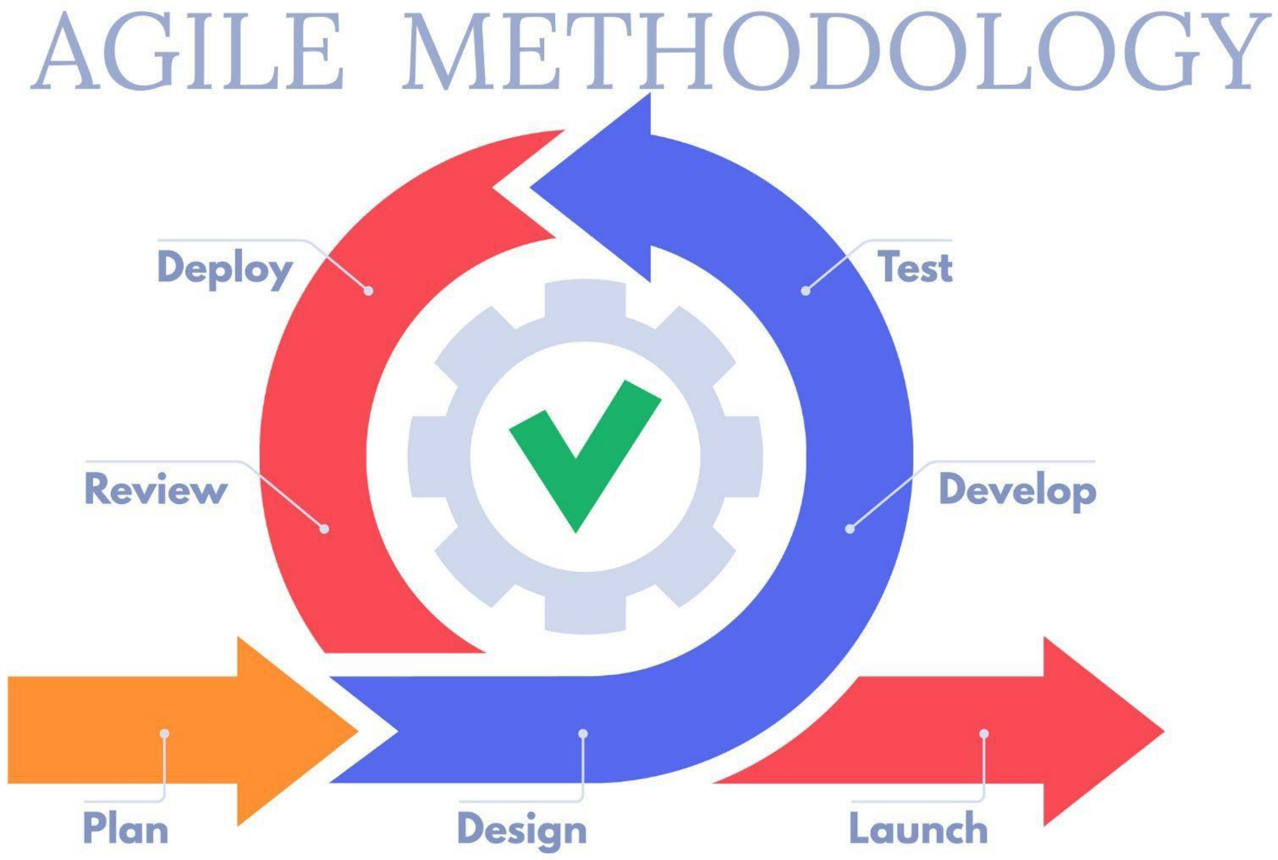
Distributed Version Control



Agile software development

- Agile is a software development approach that emphasizes team collaboration, customer and user feedback, and high adaptability to change through short release cycles.
- Teams that practice Agile provide continual changes and improvements to customers, collect their feedback, then learn and adjust based on customer wants and needs.
- Agile is substantially different from other more traditional frameworks such as waterfall, which includes long release cycles defined by sequential phases

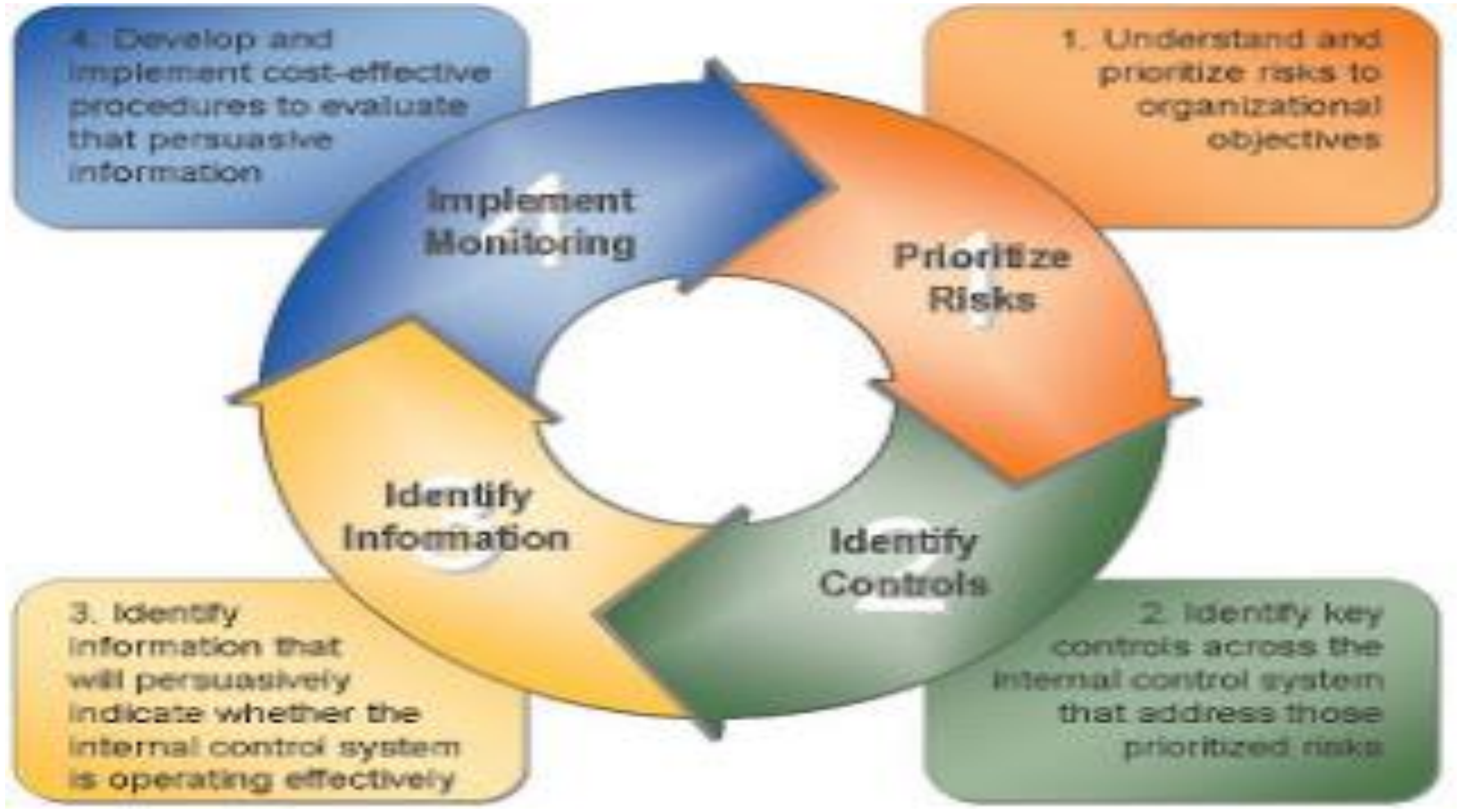
Agile Methodology



Continuous Monitoring

- Continuous monitoring means having full, real-time visibility into the performance and health of the entire application stack.
- This visibility ranges from the underlying infrastructure running the application to higher-level software components.
- Visibility is accomplished through the collection of telemetry and metadata and setting of alerts for predefined conditions that warrant attention from an operator.
- Telemetry comprises event data and logs collected from various parts of the system, which are stored where they can be analyzed and queried.
- High-performing DevOps teams ensure they set actionable, meaningful alerts and collect rich telemetry so they can draw insights from vast amounts of data.
- These insights help the team mitigate issues in real time and see how to improve the application in future development cycles.

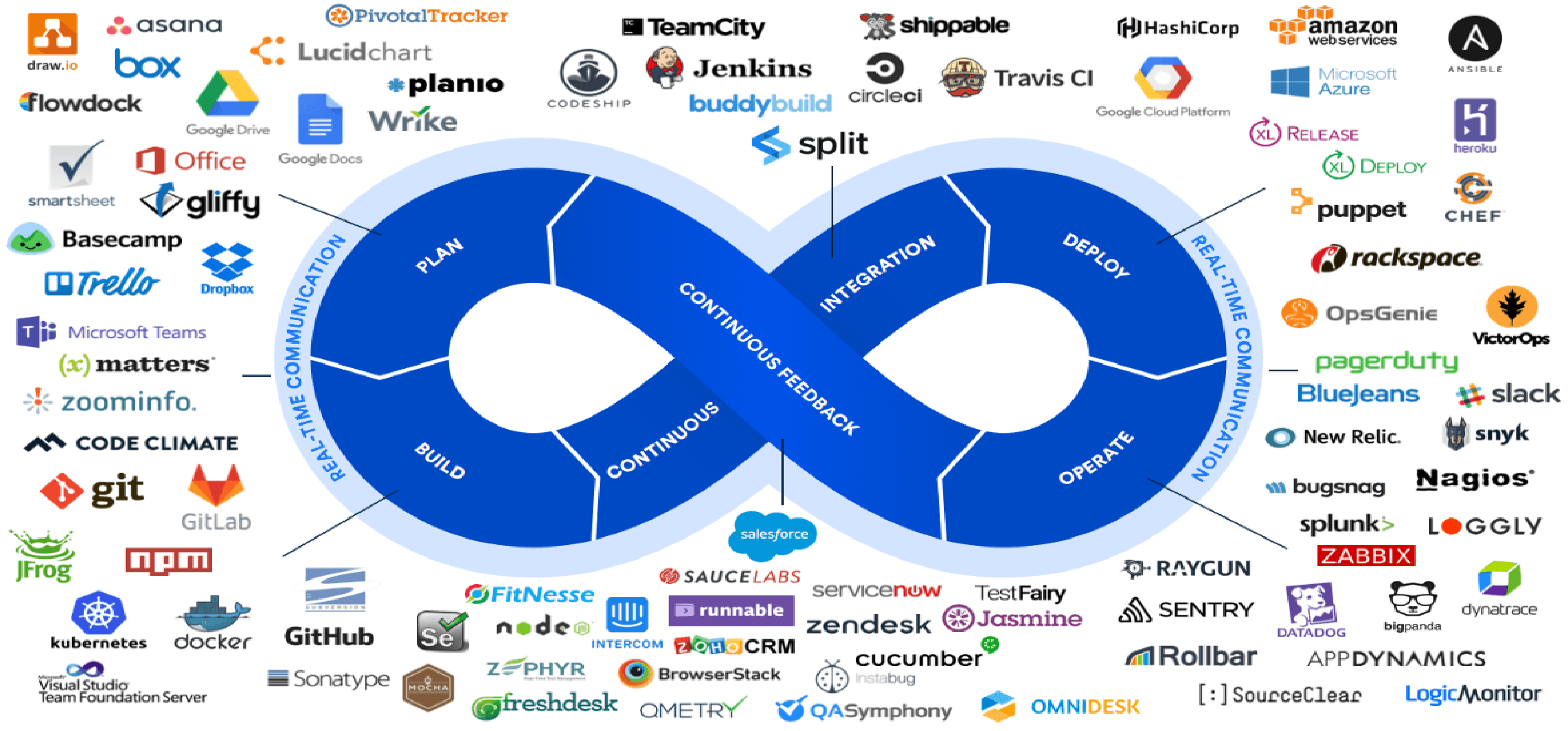
Continuous Monitoring


4. Develop and implement cost-effective procedures to evaluate that persuasive information
1. Understand and prioritize risks to organizational objectives
Implement Monitoring
4
Prioritize Risks
1
Identify Information
3
Identify Controls
2
3. Identify information that will persuasively indicate whether the internal control system is operating effectively
2. Identify key controls across the internal control system that address those prioritized risks

Most Demanding DevOps Skills

- **1. Linux Fundamentals And Scripting**
- **2. Knowledge On Various DevOps Tools And Technologies**
- 2. **1. Source Code Management-** Git, Github, Gitlab
- **2. 2. Configuration Management-** Puppet, Chef, and Ansible are the top players for configuration management.
- 2. **3.Continuous Integration-** Jenkins and Bamboo are the main tools for Continuous integration.
- 2. **4. Continuous Testing-** Selenium, TestComplete, and TestingWhiz are the most common tools for Continuous testing.
- **2.5. Continuous Monitoring-** Nagios, Zabbix, Splunk, etc.
- 2.**6. Containerization- Kubernetes, OpenShift, ,Nomad ,Podman**

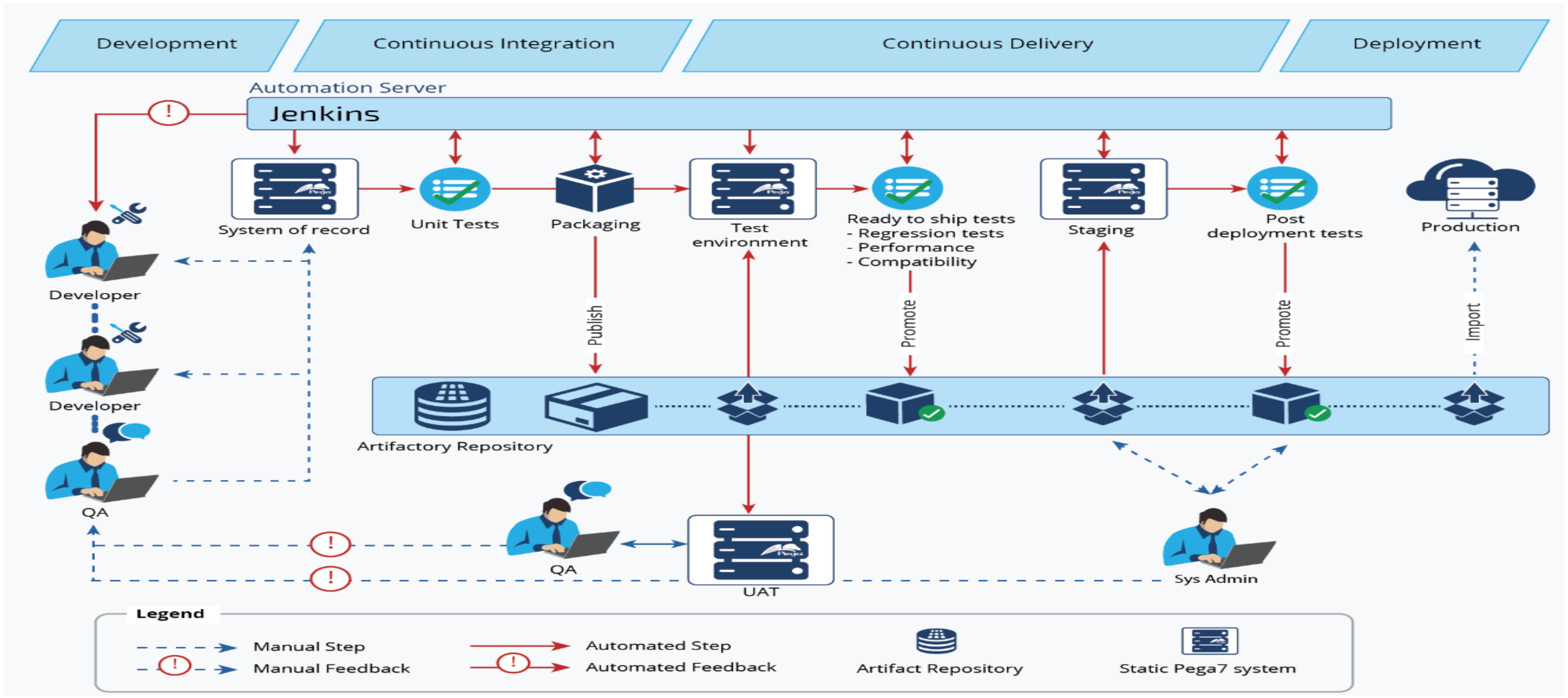
Most Demanding DevOps Skills



DevOps Pipeline

- A DevOps pipeline is a combination of automation, tools, and practices across the SDLC to facilitate the development and deployment of software into the hands of end users.
- Critically, there is no one-size-fits-all approach to building a DevOps pipeline and they often vary in design and implementation from one organization to another.
- Most DevOps pipelines, however, involve automation, continuous integration and continuous deployment (CI/CD), automated testing, reporting, and monitoring.

DevOps Pipeline



DevOps Technical Challenges

- **1. The Rise Of Low-Code And No-Code Platforms-** no-code platforms will grow as software demand rises, owing to these platforms' ability to enable developers to construct apps without writing code
- **2. A Focus On Security And Compliance**
- **3. Serverless Architecture Adoption**
- **4. Adoption Of Containerization**
- **5. Automating Manual Processes**
- **6. Increased Focus On Observability And Monitoring**
- **7. Increased Adoption Of GitOps-** GitOps uses a Git repository as the single source of truth for infrastructure definitions. Git is an open source version control system that tracks code management changes, and a Git repository is a .git folder in a project that tracks all changes made to files in a project over time

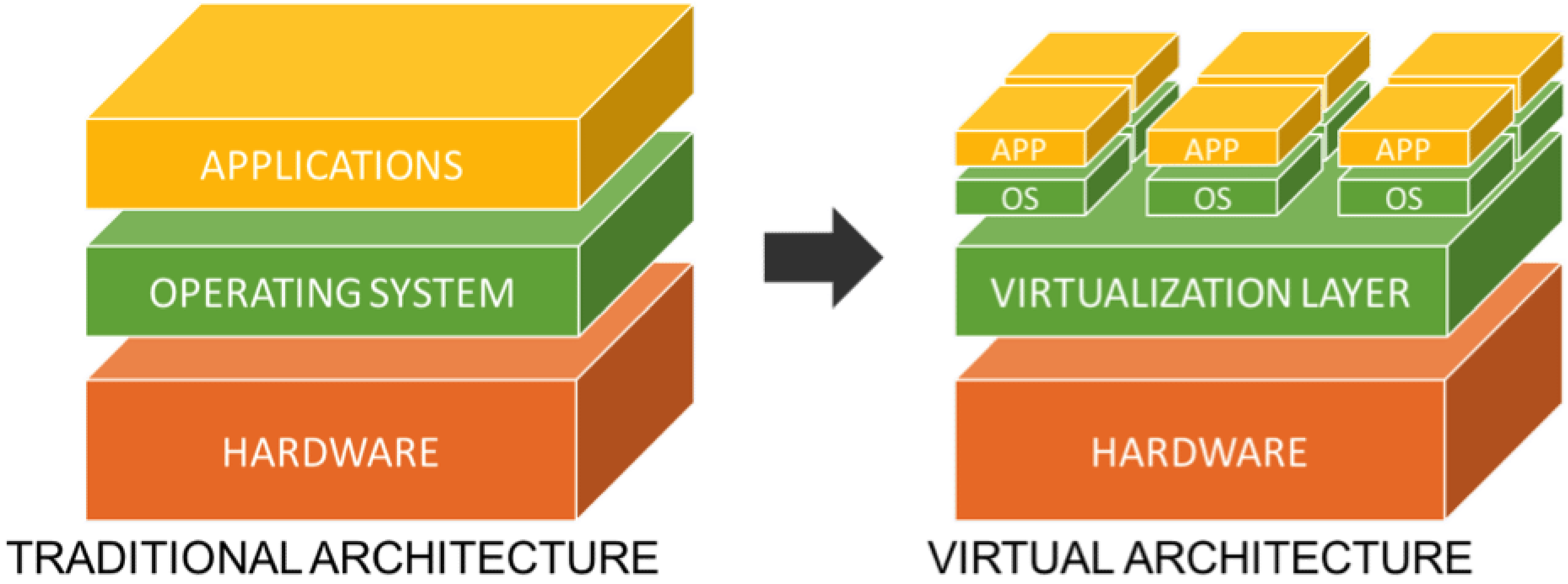
Cloud Computing

- **Cloud computing**[1] is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user.[2]
- Large clouds often have functions distributed over multiple locations, each of which is a data center.
- Cloud computing relies on sharing of resources to achieve coherence and typically uses a pay-as-you-go model, which can help in reducing capital expenses but may also lead to unexpected operating expenses for users.[3]

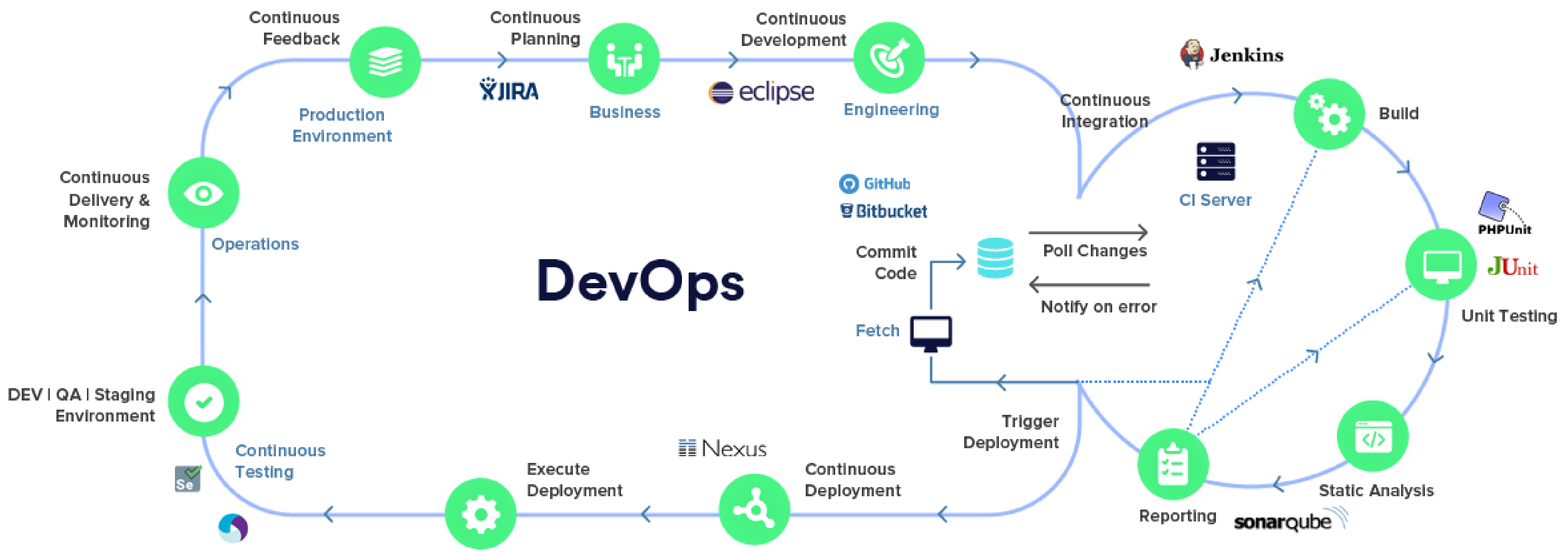
Cloud & Virtualization Architecture

- In cloud computing, virtualization facilitates the creation of virtual versions of hardware such as desktops, as well as virtual ecosystems for OS, storage, memory and networking resources.
- A virtualization architecture runs multiple OS on the same machine using the same hardware and also ensures their smooth functioning.
- Virtualization: facilitates the creation of virtual machines and ensures the smooth functioning of multiple operating systems. It also helps create a virtual ecosystem for server operating systems and multiple storage devices, and it runs multiple operating systems.
- Cloud Computing is identified as an application or service that involves a virtual ecosystem. Such an ecosystem could be of public or private nature. With Virtualization, the need to have a physical infrastructure is reduced.

Cloud & Virtualization Architecture



Devops Architecture



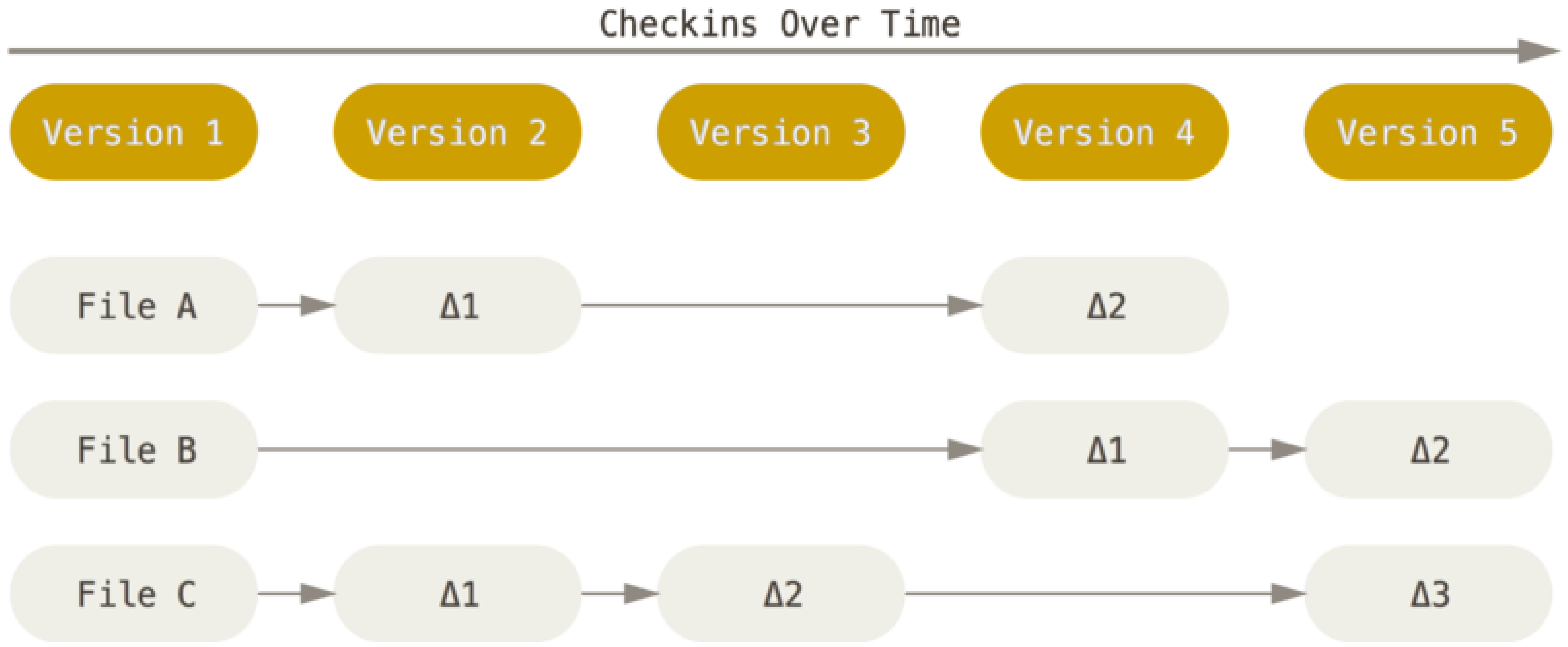
The Cloud Service

- **What is a cloud service provider?**
- Cloud service providers allow you to allocate the management of one, several, or all of the parts of your infrastructure to a third party. Instead of buying and maintaining your own infrastructure, you access it as a service.
- **Why do DevOps teams need a cloud provider?**
- And, therefore associating with a cloud provider helps reduce the upfront development and ongoing maintenance costs. It also offers opportunities for DevOps teams to create effective automation strategies and their automated program tasks.

GIT – A Version controlling tool

- **What is Git?**
- **Git** is a distributed version control system that tracks changes in any set of computer files, usually used for coordinating work among programmers who are collaboratively developing source code during software development.
- Its goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different computers).
- Git stores and thinks about information in a very different way
- The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes

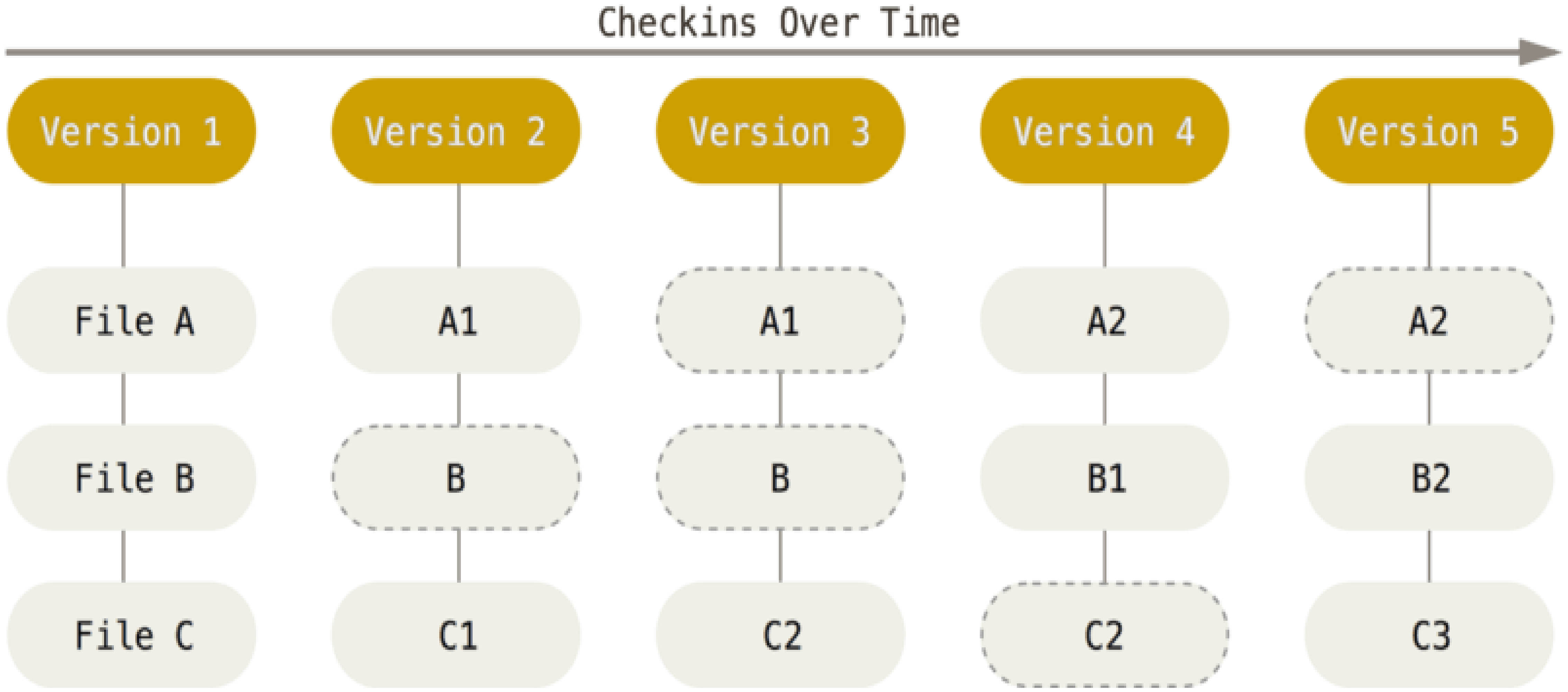
Other VCS Snapshots



Git Snap shot

- Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem.
- With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
- To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.

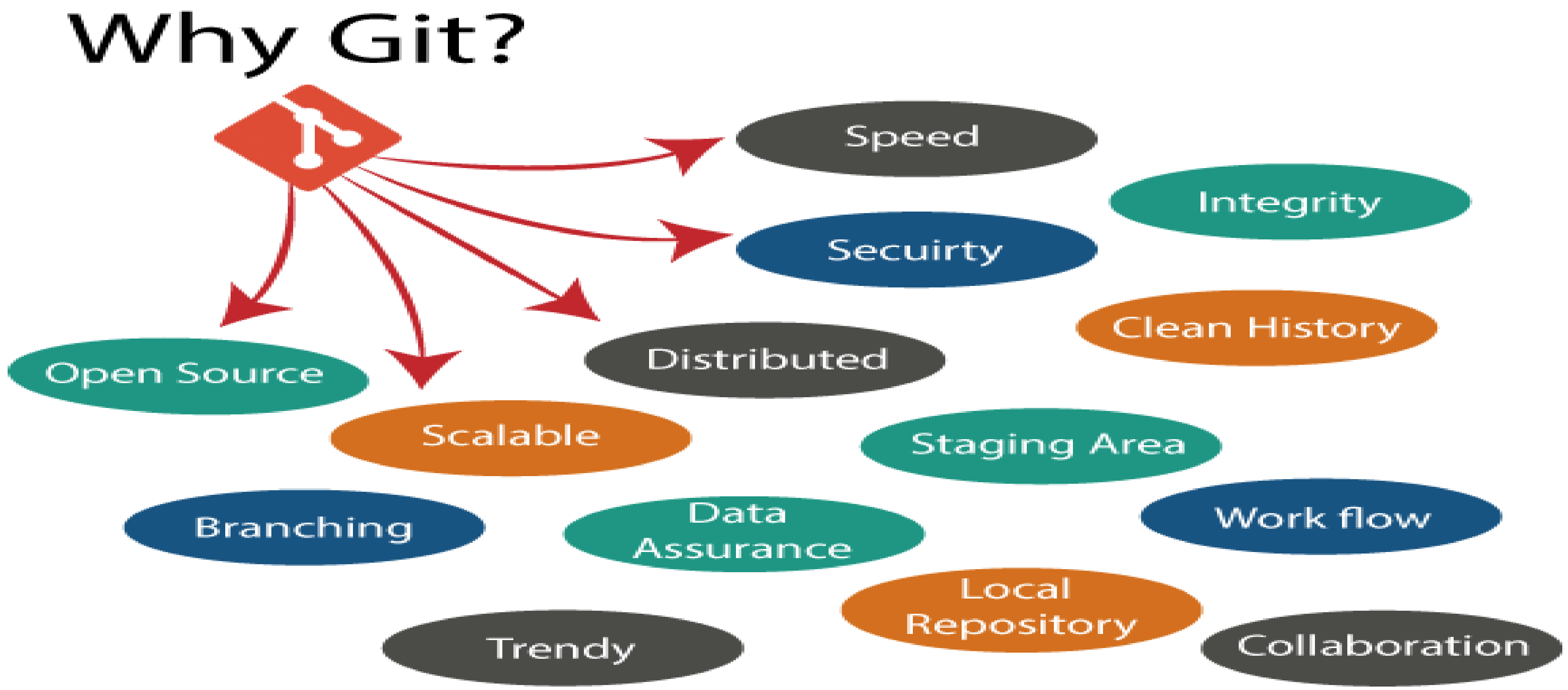
Git Snap Shot



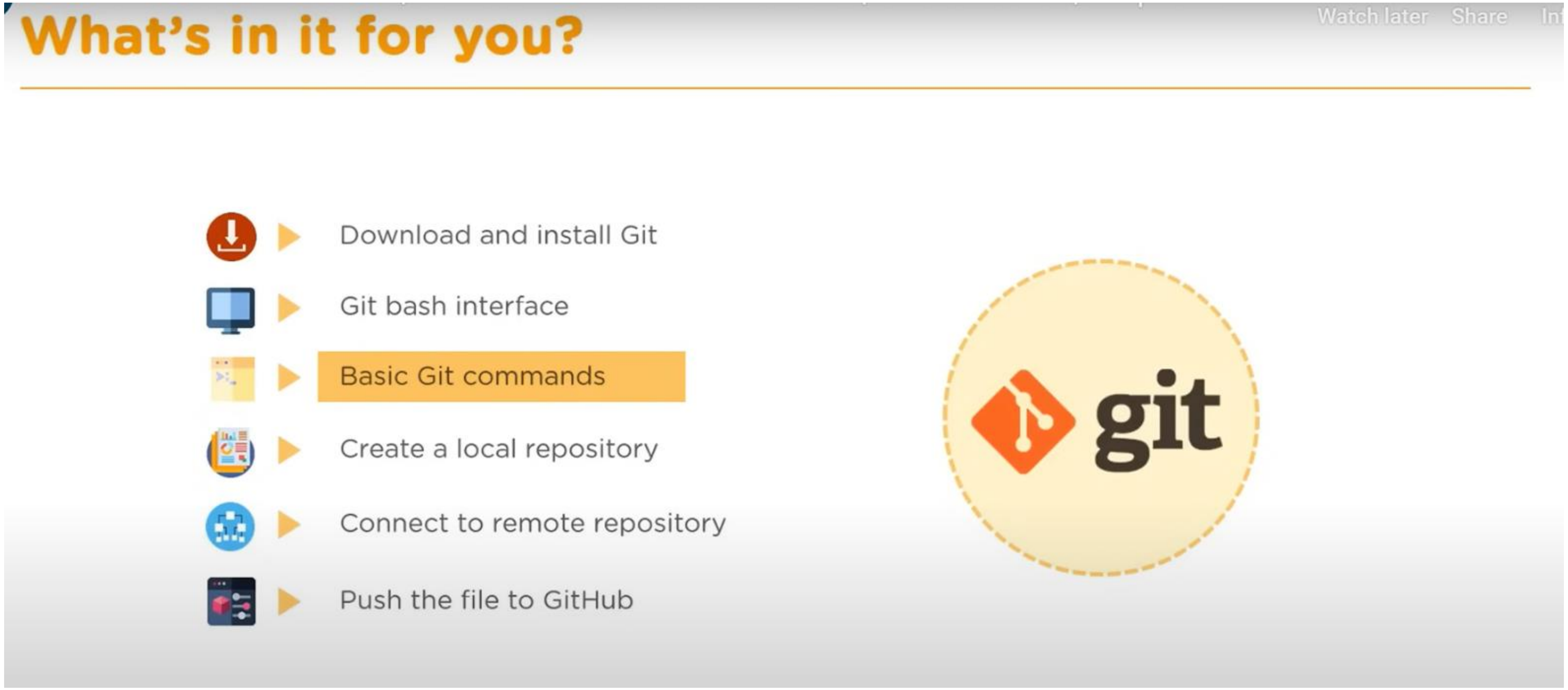
Essentials of GIT in industry

- Git is an essential part of DevOps. It’s a distributed version control system that enables non-linear workflows in a distributed way by offering data assurance to develop first-quality software.
- Let’s assume there are three developers A, B, and C doing the same project. They work in isolation and saving the files in a shared folder. Each and every changes of file has bee monitor and record as snapshot then reflect the conflict and revert back to old version

Essential of Git industry



Git Installation



What's in it for you?

- Download and install Git
- Git bash interface
- Basic Git commands
- Create a local repository
- Connect to remote repository
- Push the file to GitHub

Getting Started - First-Time Git Setup

Your Identity :

- The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating

$ git config --global user.name “Nareshbabu”

$git config –global user.email naresh@itech.com

- By Default Git will create branch called master .when you create a new repository with Git init inside the git practise directory (create by user for Git practise then) navigate inside the new folder then run $ git init

Git Basics - Getting a Git Repository

- Initializing a Repository in an Existing Directory
- If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project’s directory. If you’ve never done this, it looks a little different depending on which system you’re running:

for Linux: $ cd /home/user/my_project

for macOS: $ cd /Users/user/my_project

for Windows: $ cd C:/Users/user/my_project

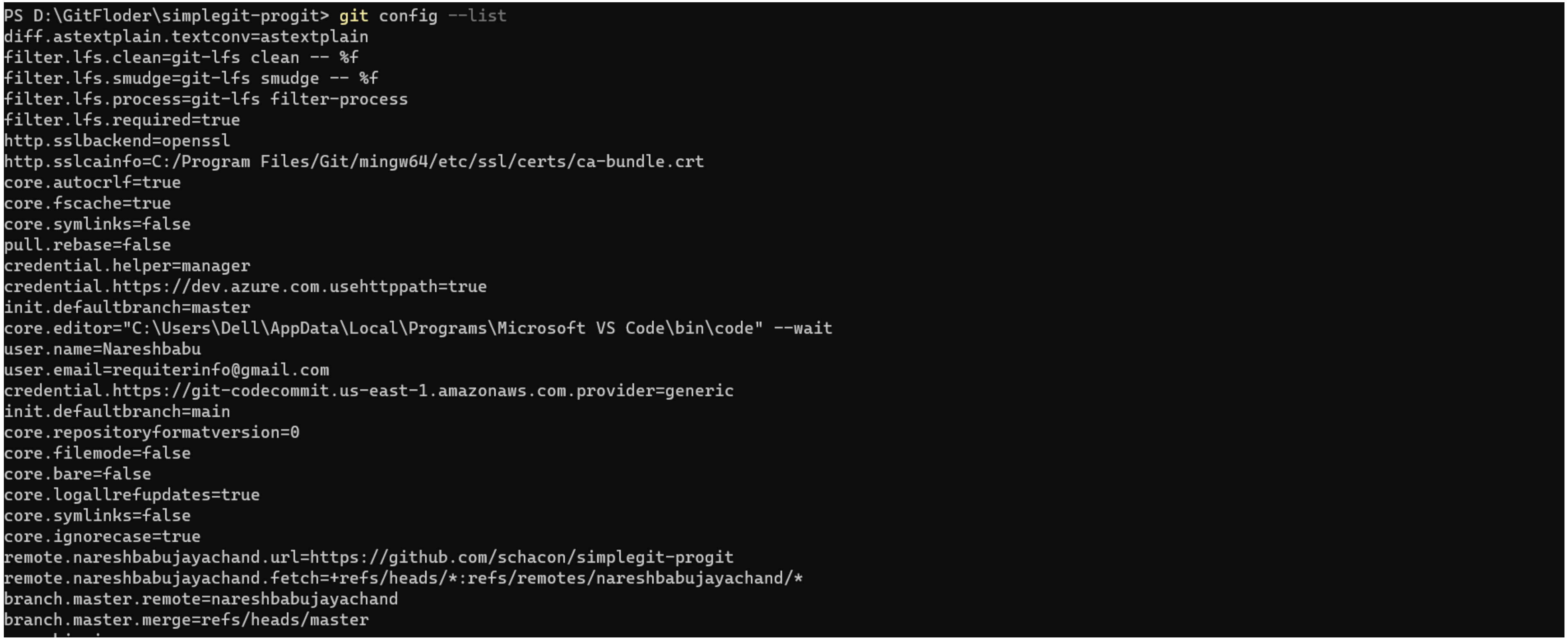
and type: $ git init

- This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet. See Git Internals for more information about exactly what files are contained in the .git directory you just created.
- To reset the main as default branch name do

$ git config –global init.defaultBranch main

Checking Your Settings

- If you want to check your configuration settings, you can use the **git config --list** command to list all the settings Git can find at that point: $ git config user.name



```
PS D:\GitFloder\simplegit-progit> git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
core.editor="C:\Users\Dell\AppData\Local\Programs\Microsoft VS Code\bin\code" --wait
user.name=Nareshbabu
user.email=requiterinfo@gmail.com
credential.https://git-codecommit.us-east-1.amazonaws.com.provider=generic
init.defaultbranch=main
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
remote.nareshbabujayachand.url=https://github.com/schacon/simplegit-progit
remote.nareshbabujayachand.fetch=+refs/heads/*:refs/remotes/nareshbabujayachand/*
branch.master.remote=nareshbabujayachand
branch.master.merge=refs/heads/master
```

Getting Help

- If you ever need help while using Git, there are three equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

- For example, you can get the manpage help for the git config command by running this:

```
$ git help config
```

- These commands are nice because you can access them anywhere, even offline. If the manpages and this book aren’t enough and you need in-person help, you can try the #git, #github, or #gitlab channels on the Libera Chat IRC server, which can be found at https://libera.chat/. These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.
- In addition, if you don’t need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can ask for the more concise “help” output with the -h option, as in:

Cloning an Existing Repository

If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is git clone.

If you're familiar with other VCSs such as Subversion, you'll notice that the command is "clone" and not "checkout".

This is an important distinction — instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone.

In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there — see Getting Git on a Server for more details).

You clone a repository with git clone <url>. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

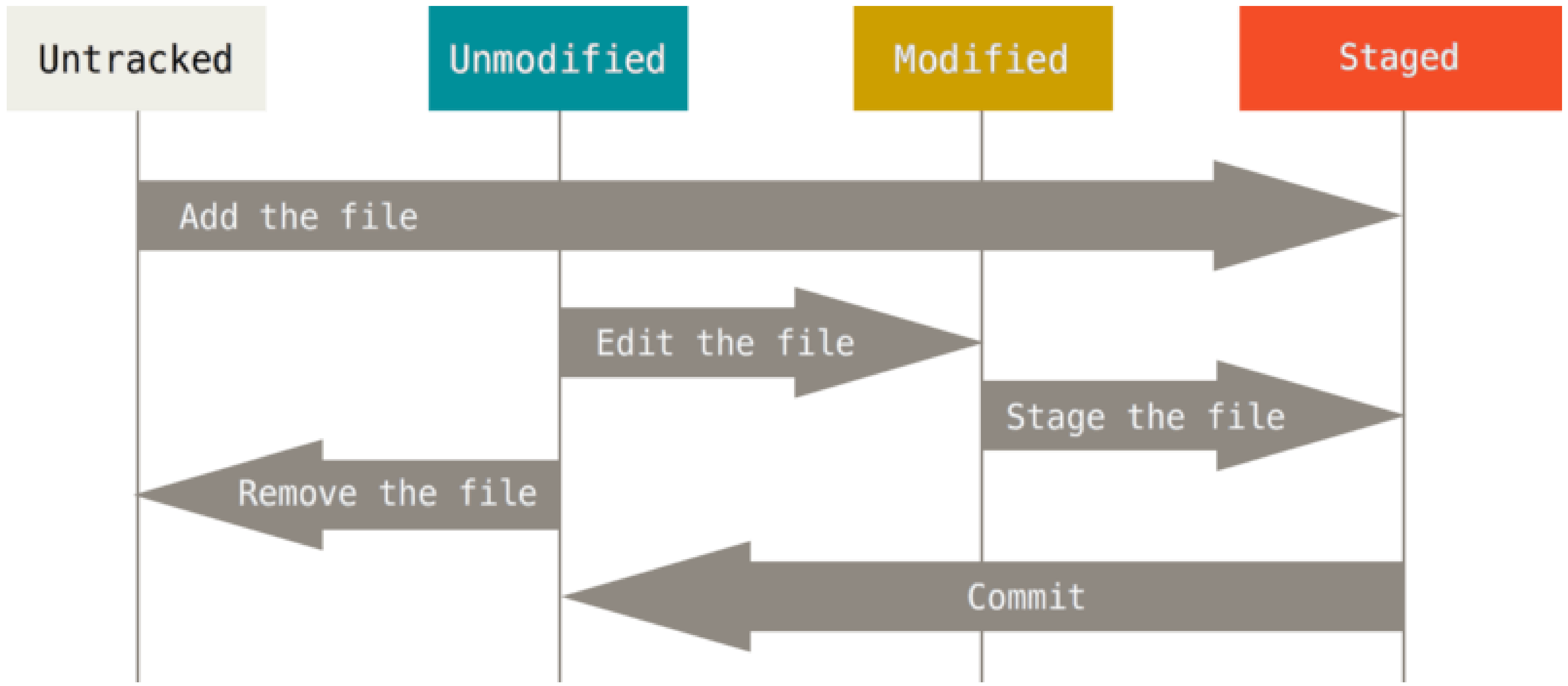
```
$ git clone https://github.com/libgit2/libgit2
```

That creates a directory named libgit2, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2 directory that was just created, you'll see the project files in there, ready to be worked on or used.

Git Basics - Recording Changes to the Repository

- At this point, you should have a **`bona fide`** Git repository on your local machine, and a checkout or **`working copy`** of all of its files in front of you.
- Typically, you’ll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.
- Remember that each file in your working directory can be in one of two states: **`tracked`** or **`untracked`**. Tracked files are files that were in the last snapshot, as well as any newly staged files; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.
- Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.
- As you edit files, Git sees them as modified, because you’ve changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

Git Basics - Recording Changes to the Repository



Checking the Status of Your Files

- The main tool you use to determine which files are in which state is the git status command.
- If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

- This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn’t see any untracked files, or they would be listed here.
- Finally, the command tells you which branch you’re on and informs you that it has not diverged from the same branch on the server. For now, that branch is always master, which is the default.

Add new file or Modify the Existing file

- Let’s say you add a new file to your project, a simple README file. If the file didn’t exist before, and you run git status, you see your untracked file like so:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README
```

- nothing added to commit but untracked files present (use "git add" to track)

Tracking New Files

- In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:
- $ git add README
- If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   README
```

- You can tell that it’s staged because it’s under the “Changes to be committed” heading. If you commit at this point, the version of the file at the time you ran git add is what will be in the subsequent historical snapshot.

Staging Modified Files

- Let’s change a file that was already tracked. If you change a previously tracked file called CONTRIBUTING.md and then run your git status command again, you get something that looks like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
new file:   README
    modified:   CONTRIBUTING.md
```

- Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it.

Staging Modified Files

- The CONTRIBUTING.md file appears under a section named “Changes not staged for commit” — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the git add command. git add is a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”. Let’s run git add now to stage the CONTRIBUTING.md file, and then run git status again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

Short Status

- While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

```
$ git status -s
 M README→ Modifed File
A  lib/git.rb--→ add file to Stagging
M  lib/simplegit.rb -→ Modify File
D      unmerged, both deleted
A      unmerged, added by us
D        unmerged, deleted by them
```

- New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Viewing Your Staged and Unstaged Changes

- If the git status command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the git diff command.
- We'll cover git diff in more detail later, but you'll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed — the patch, as it were.
- Let's say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your git status command, you once again see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   README
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

To see what you’ve changed but not yet staged, type git diff with no other arguments:

- **$ git diff**
- diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
- index 8ebb991..643e24f 100644
- --- a/CONTRIBUTING.md
- +++ b/CONTRIBUTING.md
- @@ -65,7 +65,8 @@ branch directly, things can get messy.
- Please include a nice description of your changes when you submit your PR;
- if we have to read the whole diff to figure out why you're contributing
- in the first place, you're less likely to get feedback and have your change
- -merged in.
- +merged in. Also, split your changes into comprehensive chunks if your patch is
- +longer than a dozen lines.
- If you are starting to work on a particular area, feel free to submit a PR

git diff to see what is still unstaged:

- **$ git diff**
- diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
- index 643e24f..87f08c8 100644
- --- a/CONTRIBUTING.md
- +++ b/CONTRIBUTING.md
- @@ -119,3 +119,4 @@ at the
- ## Starter Projects

git diff to see the changes in the file that are staged and the changes that are unstaged.

- See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
- +# test line
- and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):
- **$ git diff --cached**
- diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
- index 8ebb991..643e24f 100644
- --- a/CONTRIBUTING.md
- +++ b/CONTRIBUTING.md
- @@ -65,7 +65,8 @@ branch directly, things can get messy.
- Please include a nice description of your changes when you submit your PR;
- if we have to read the whole diff to figure out why you're contributing
- in the first place, you're less likely to get feedback and have your change

Committing Your Changes

- Now that your staging area is set up the way you want it, you can commit your changes.
- Remember that anything that is still unstaged — any files you have created or modified that you haven’t run git add on since you edited them — won’t go into this commit.
- They will stay as modified files on your disk. In this case, let’s say that the last time you ran git status, you saw that everything was staged, so you’re ready to commit your changes.
- The simplest way to commit is to type git commit:

```
$ git commit
$ git commit -m "Story 182: fix benchmarks for speed"
```

Skipping the Staging Area

- Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part:
- $ git commit -a -m 'Add new benchmarks'
- [master 83e38c7] Add new benchmarks
- 1 file changed, 5 insertions(+), 0 deletions(-)

Removing Files

- To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit.
- The git rm command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around
- If you simply remove the file from your working directory, it shows up under the "Changes not staged for commit" (that is, unstaged) area of your git status output:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md
```

Run git rm, it stages the file’s removal:

- Then, if you run git rm, it stages the file’s removal:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    deleted:    PROJECTS.md
```

- The next time you commit, the file will be gone and no longer tracked. If you modified the file or had already added it to the staging area, you must force the removal with the **-f** option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.
- Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:

```
$ git rm --cached README
```

Moving Files

- Unlike many other VCSs, Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we'll deal with detecting file movement a bit later.
- Thus it's a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

```
$ git mv file_from file_to
```

- and it works fine. In fact, if you run something like this and look at the status, you'll see that Git considers it a renamed file:

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    renamed:    README.md -> README
```

- However, this is equivalent to running something like this:

```
$ mv README.md README
$ git rm README.md //Run cmmnd for conirmation might be error
$ git add README
```

- Git figures out that it's a rename implicitly, so it doesn't matter if you rename a file that way or with the mv command. The only real difference is that git mv is one command instead of three — it's a convenience function. More importantly, you can use any tool you like to rename a file, and address the add/rm later, before you commit.

Viewing the Commit History

- After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.
- These examples use a very simple project called "simplegit". To get the project, run:

```
$ git clone https://github.com/schacon/simplegit-progit
```

When you run git log in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
    Change version number
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
    Remove unnecessary test
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
    Initial commit
```

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

Viewing the Commit History-Limited History Log

- A huge number and variety of options to the git log command are available to show you exactly what you’re looking for. Here, we’ll show you some of the most popular.
- One of the more helpful options is -p or --patch, which shows the difference (the patch output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
    Change version number              , diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644,--- a/Rakefile
+++ b/Rakefile,@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s| ,     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit“,-   s.version   =   "0.1.0"
+    s.version   =   "0.1.1“,     s.author    =   "Scott Chacon"
     s.email     =   schacon@gee-mail.com,     s.summary   =   "A simple gem for using Git in Ruby code."
```

Viewing the Commit History-on File Level

- This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with git log. For example, if you want to see some abbreviated stats for each commit, you can use the --stat option:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949 , Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700,
    Change version number , Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 , Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700 ,
    Remove unnecessary test , lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-) ,
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700 ,
    Initial commit ,
 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

Limiting Log Output

- Limiting Log Output
- In addition to output-formatting options, git log takes a number of useful limiting options; that is, options that let you show only a subset of commits. You’ve seen one such option already — the -2 option, which displays only the last two commits. In fact, you can do -<n>, where n is any integer to show the last n commits. In reality, you’re unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.
- However, the time-limiting options such as --since and --until are very useful. For example, this command gets the list of commits made in the last two weeks:
- $ git log --since=2.weeks
- This command works with lots of formats — you can specify a specific date like "2008-01-15", or a relative date such as "2 years 1 day 3 minutes ago".
- You can also filter the list to commits that match some search criteria. The --author option allows you to filter on a specific author, and the --grep option lets you search for keywords in the commit messages

Git Basics - Undoing Things