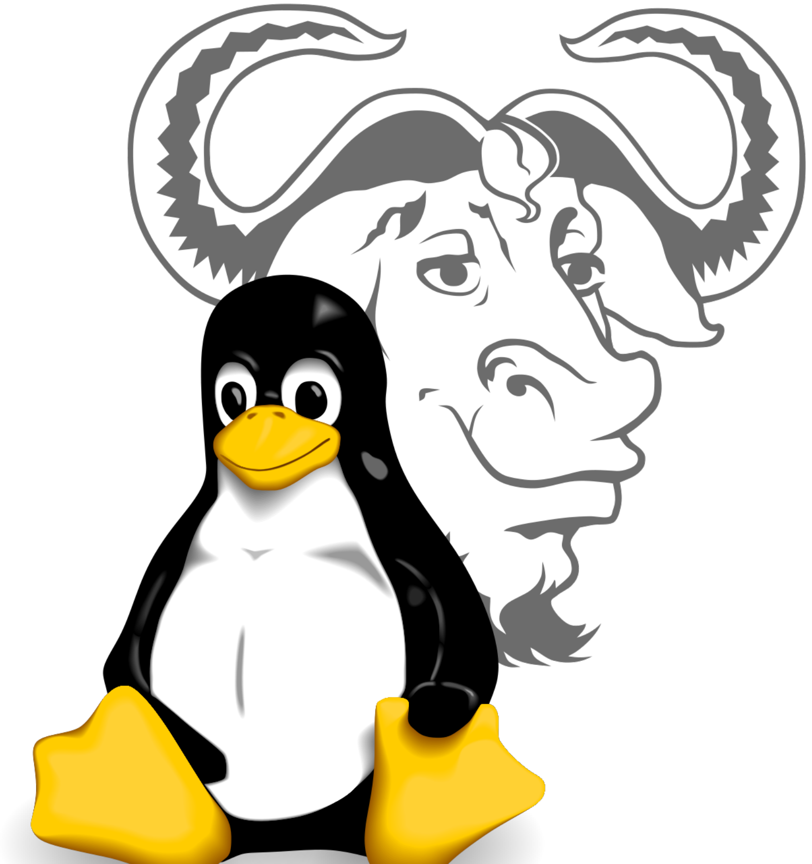
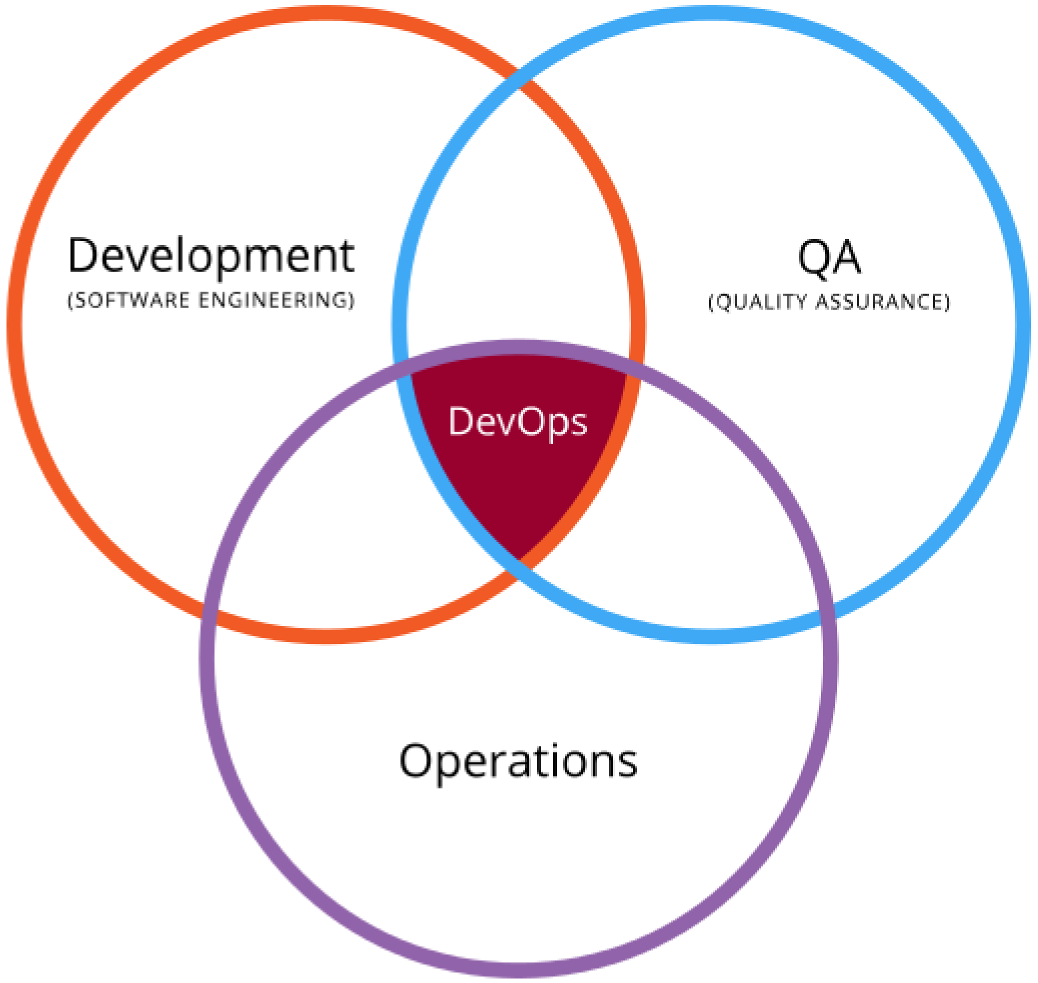
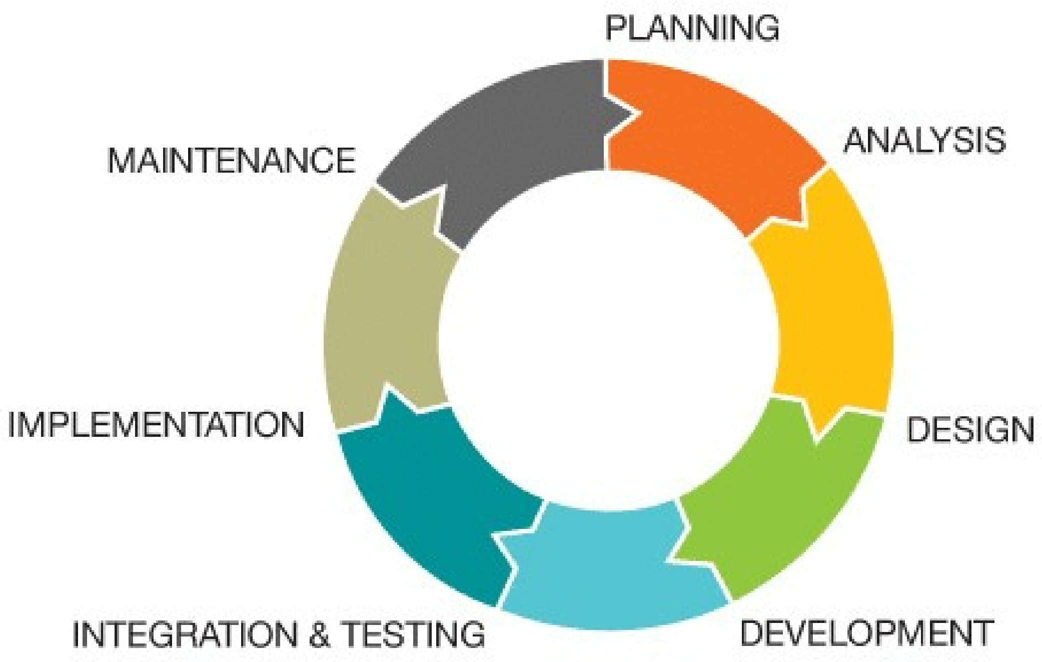
DevOps course: Introduction to Linux

```
while True:
  if time in working_hours:
    follow(DevOps)
    do_the_job
```





What is **SDLC?**



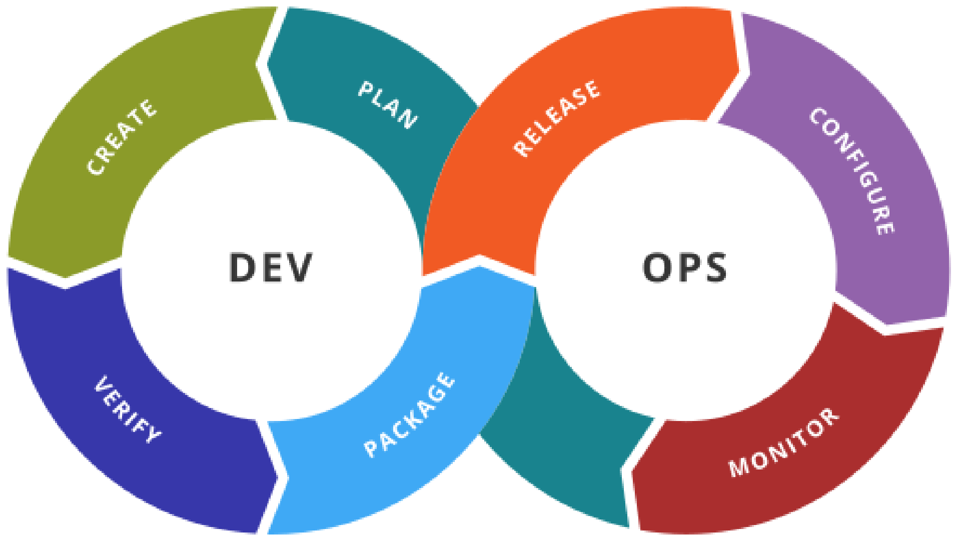
- The systems development life cycle (**SDLC**), also referred to as the application development life-cycle, is a term used in systems engineering, information systems and software engineering to describe **a process for planning, creating, testing, and deploying an information system.**

Software Development Lifecycle in 9 minutes!

- 3 **Phases**
- 3.1   System investigation
- 3.2   System analysis
- 3.3   Design
- 3.4   Environments
- 3.5   Testing
- 3.6   Training and transition
- 3.7   Operations and maintenance
- 3.8   Evaluation

https://en.wikipedia.org/wiki/Systems_development_life_cycle

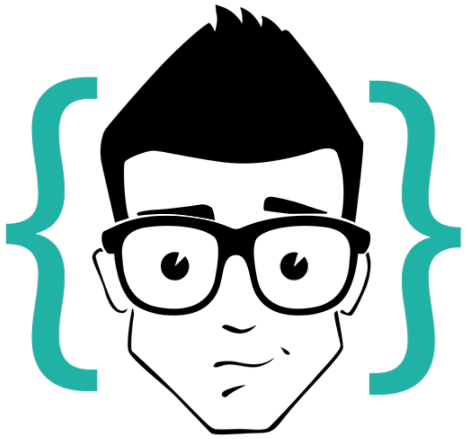
What is DevOps?



- **DevOps** (a clipped compound of "development" and "operations") is a **software engineering practice that aims at unifying software development (Dev) and software operation (Ops).**
- The main characteristic of the DevOps movement is to strongly advocate **automation and monitoring at all steps of software construction, from integration, testing, releasing to deployment and infrastructure management.** DevOps aims at **shorter development cycles, increased deployment frequency, more dependable releases**, in close **alignment with business objectives.**

- **DevOps is NOT**
- ✗ A role, person, or organization
- ✗ Something only systems administrators do
- ✗ Something only developers do
- ✗ Writing Chef/Puppet/Ansible code
- ✗ Tools

https://en.wikipedia.org/wiki/DevOps



- Who we are then?
- → ENGINEERS!!!
- What’s our GOAL within DevOps methodology?
- Help our organization to be high performing by applying our knowledge to SDLC processes.

CI vs **CD** vs **CD**

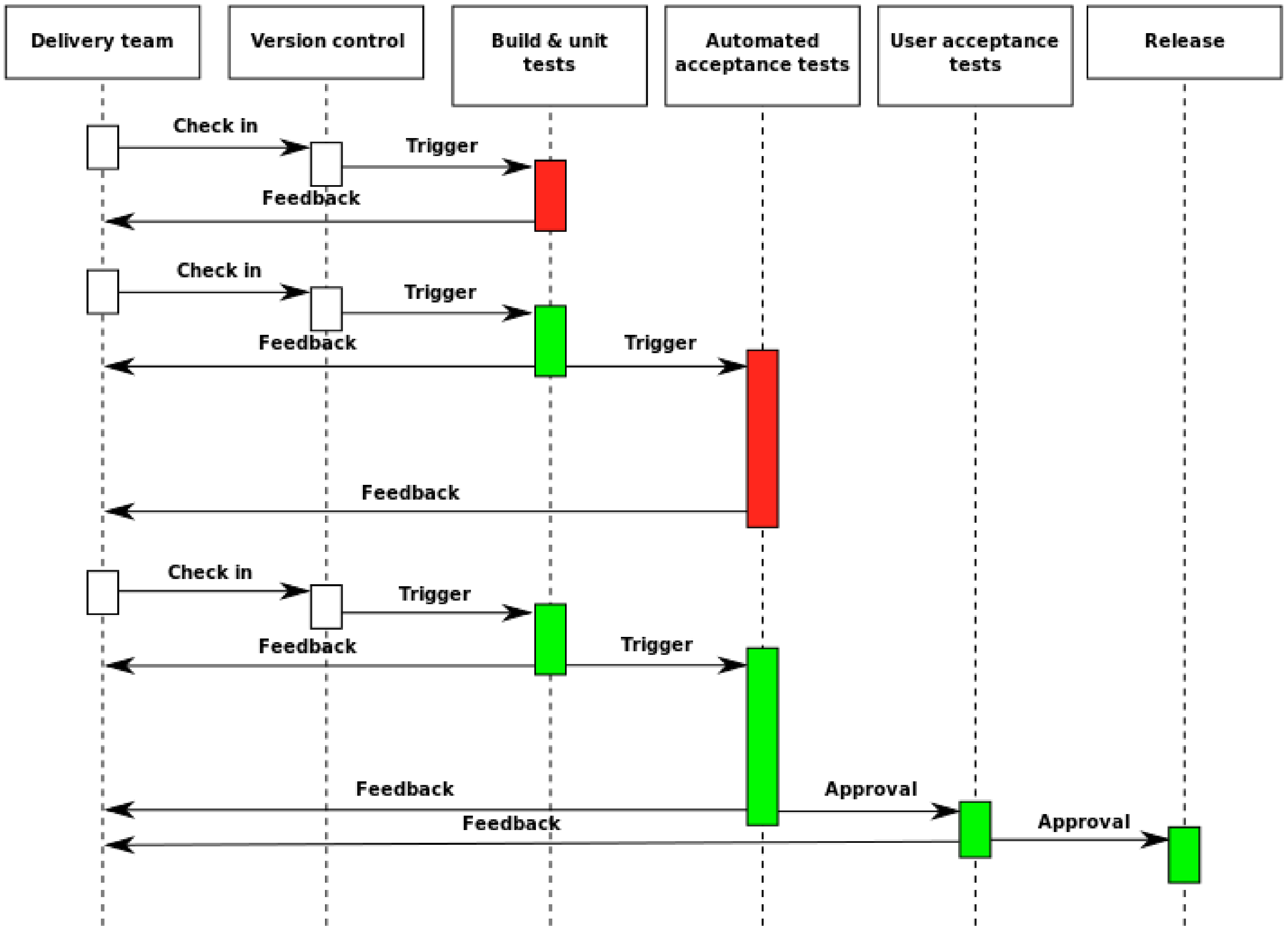
- What is CI?
  - In software engineering, **continuous integration** (CI) is the practice of merging all developer working copies to a shared mainline several times a day

- What is **another CD**?
  - **Continuous deployment** means that every change is automatically deployed to production.
  - Continuous delivery means that the **team ensures every *change* can be deployed** to production but may choose not to do it, usually due to business reasons. In order to do continuous deployment one must be doing continuous delivery.

- What is **CD**?
  - **Continuous delivery** (CD) is a software engineering approach in which teams produce software in short cycles, **ensuring that the *software* can be reliably released at any time.**
  - It aims at **building, testing, and releasing software faster and more frequently**.
  - The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery.

https://en.wikipedia.org/wiki/Continuous_integration
https://en.wikipedia.org/wiki/Continuous_delivery

Continuous deployment diagram



Git Flow

Key Benefits

Parallel Development

One of the great things about GitFlow is that it makes parallel development very easy, by isolating new development from finished work. New development (such as features and non-emergency bug fixes) is done in feature branches, and is only merged back into main body of code when the developer(s) is happy that the code is ready for release.

Although interruptions are a BadThing(tm), if you are asked to switch from one task to another, all you need to do is commit your changes and then create a new feature branch for your new task. When that task is done, just checkout your original feature branch and you can continue where you left off.

Collaboration

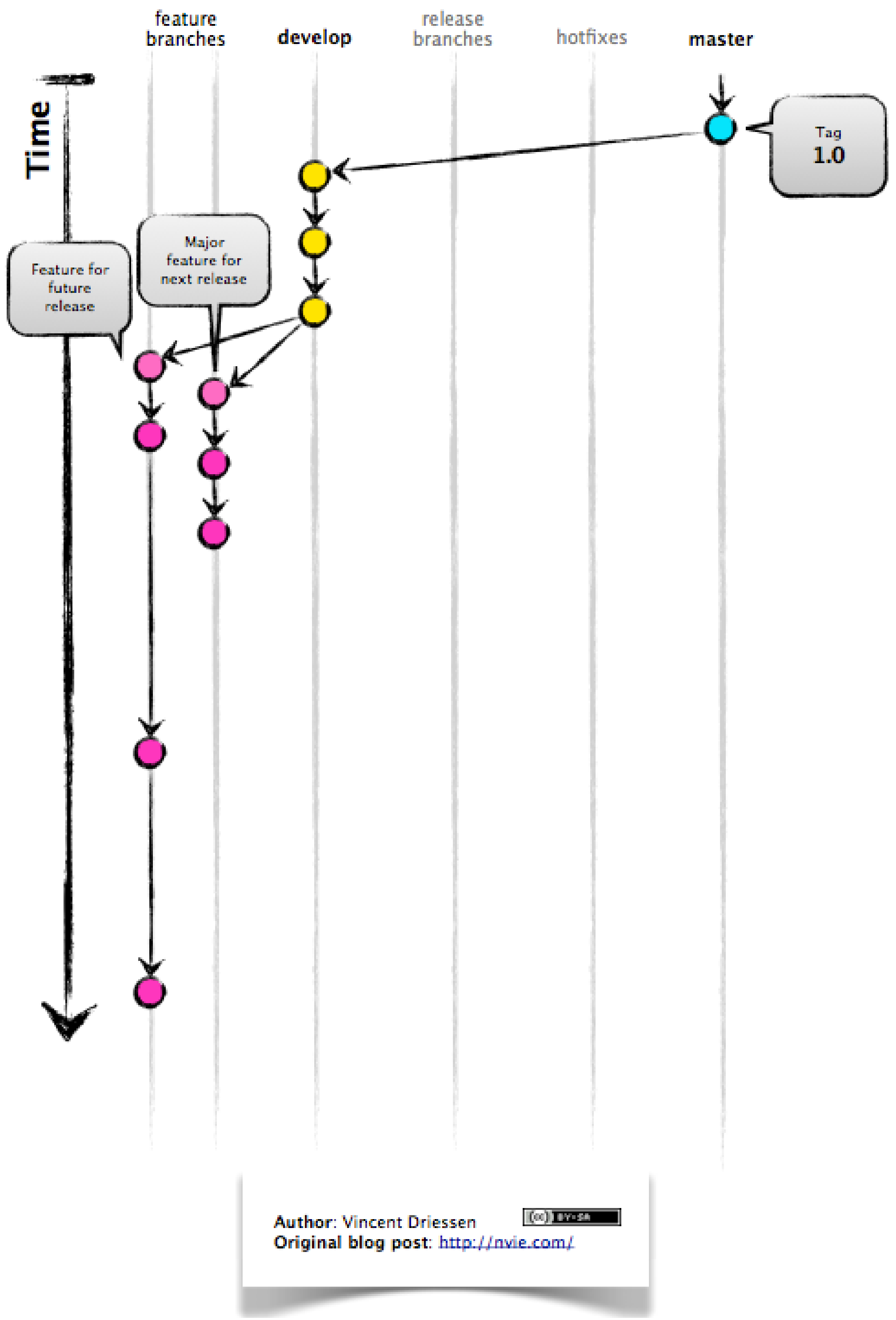
Feature branches also make it easier for two or more developers to collaborate on the same feature, because each feature branch is a sandbox where the only changes are the changes necessary to get the new feature working. That makes it very easy to see and follow what each collaborator is doing.

Release Staging Area

As new development is completed, it gets merged back into the develop branch, which is a staging area for all completed features that haven't yet been released. So when the next release is branched off of develop, it will automatically contain all of the new stuff that has been finished.

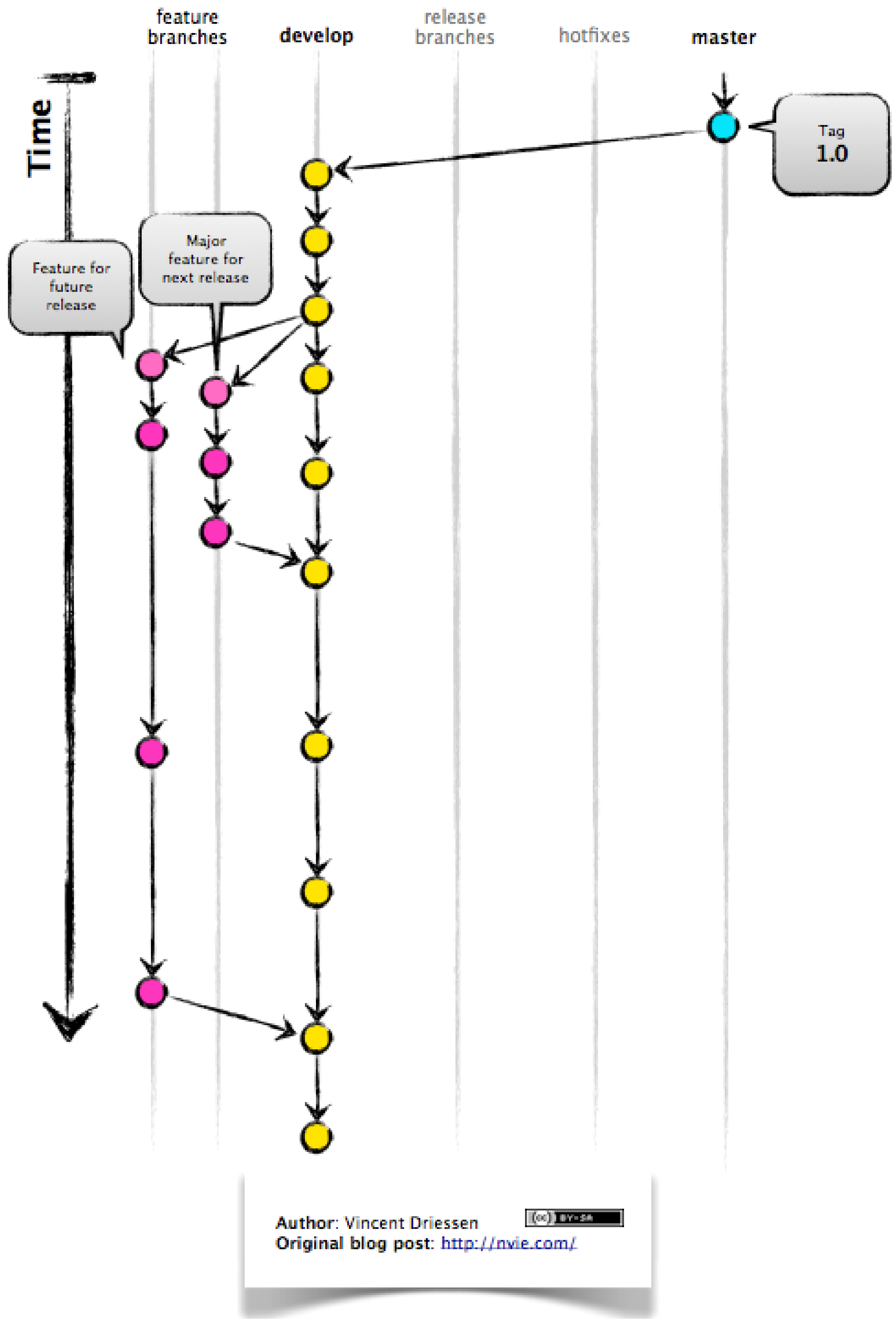
Support For Emergency Fixes

GitFlow supports hotfix branches - branches made from a tagged release. You can use these to make an emergency change, safe in the knowledge that the hotfix will only contain your emergency fix. There's no risk that you'll accidentally merge in new development at the same time.

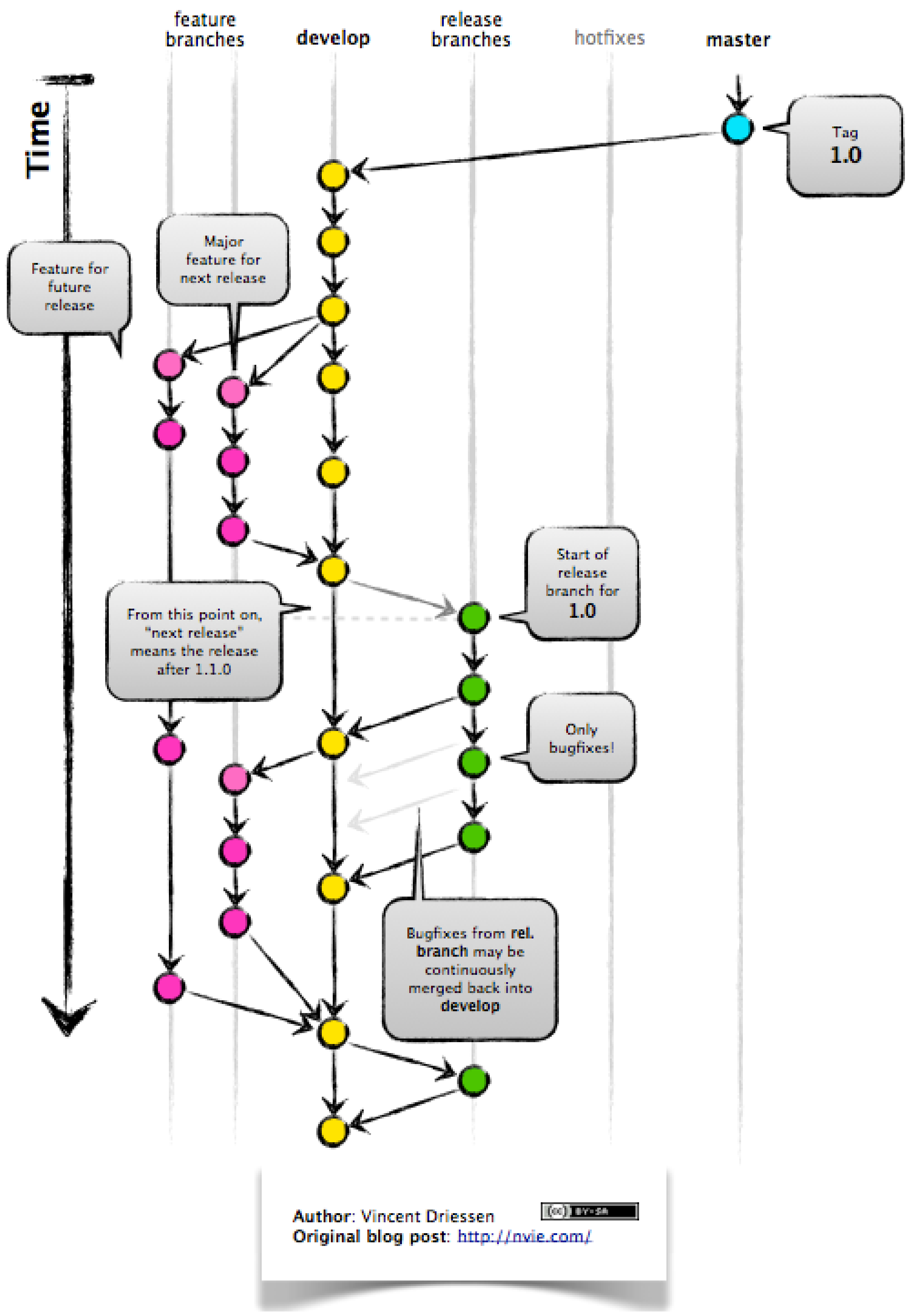

feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Major feature for next release
Feature for future release
Author: Vincent Driessen
Original blog post: http://nvie.com/

How it works?

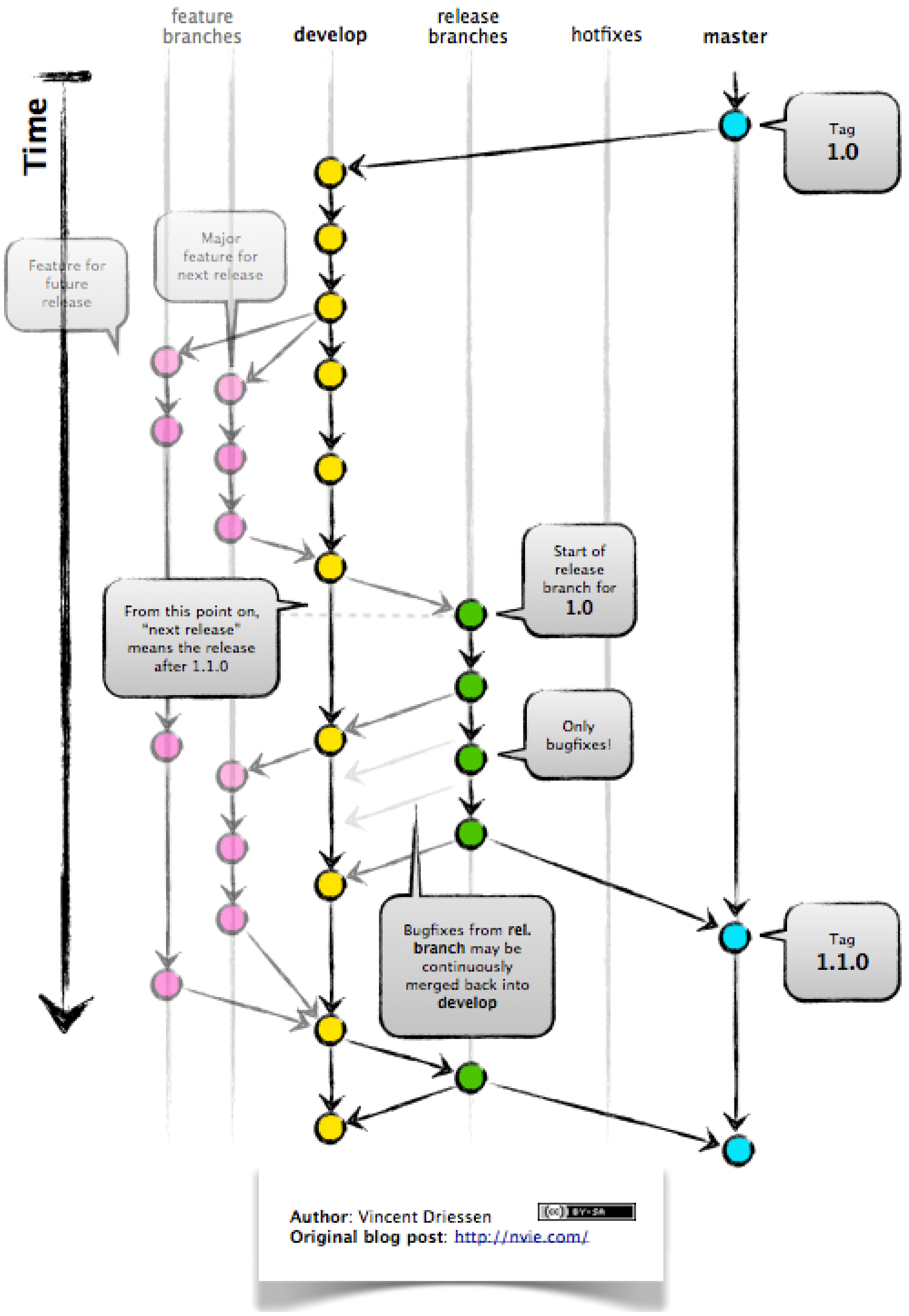
New development (new features, non-emergency bug fixes) are built in **feature branches**



Feature branches are branched off of the **develop branch**, and finished features and fixes are merged back into the **develop branch** when they're ready for release

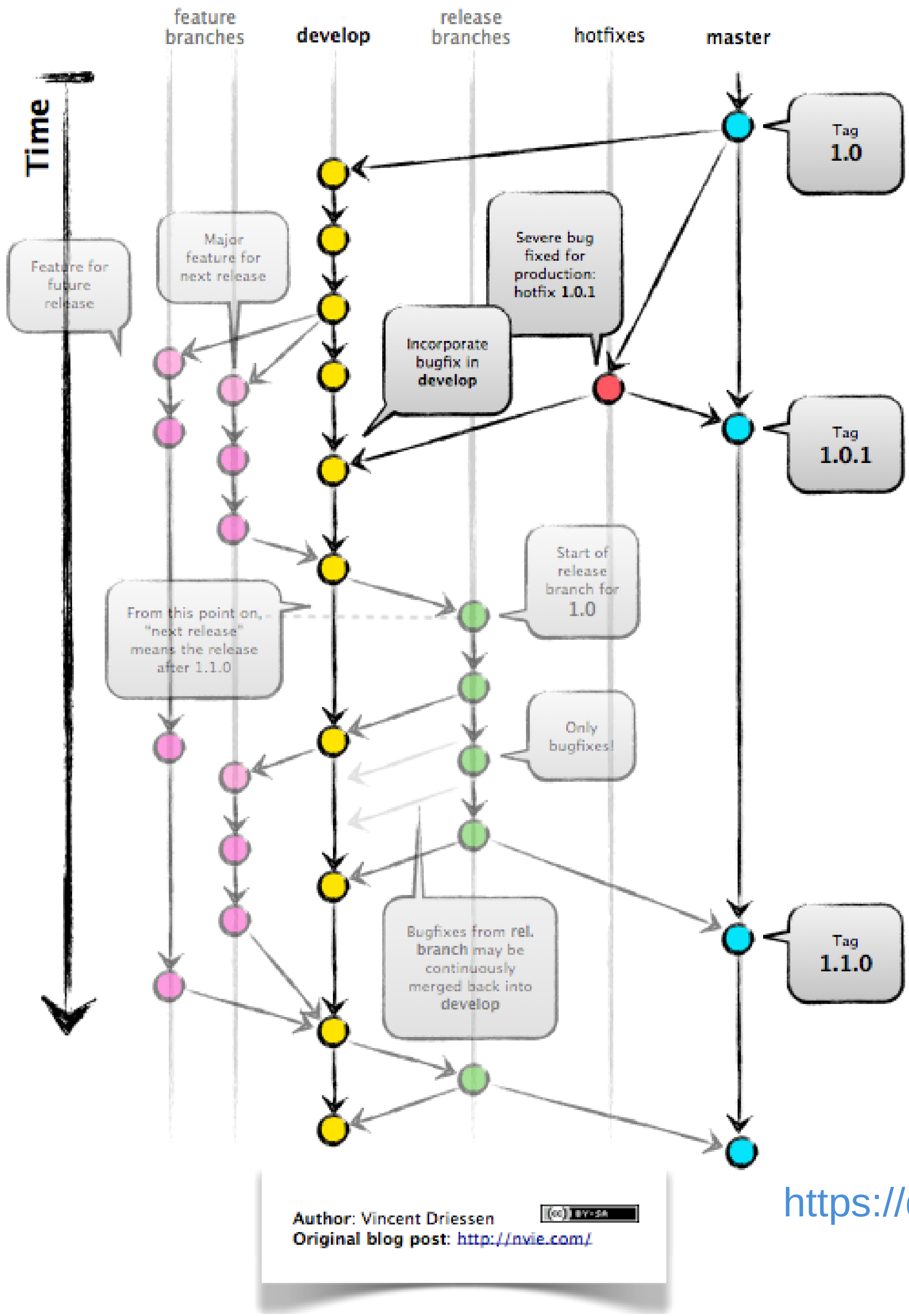

feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Author: Vincent Driessen
Original blog post: http://nvie.com/

When it is time to make a release, a **release branch** is created off of **develop**


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Major feature for next release
Feature for future release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/

- The code in the **release branch** is deployed onto a suitable test environment, tested, and any problems are fixed directly in the release branch.
- This **deploy -> test -> fix -> redeploy -> retest** cycle continues until you're happy that the release is good enough to release to customers.

- When the release is finished, the **release branch** is merged into **master** and into **develop** too, to make sure that any changes made in the **c** aren't accidentally lost by new development.


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 1.0.1
Incorporate bugfix in develop
Tag 1.0.1
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/

- The **master branch** tracks released code only. The only commits to **master** are merges from **release** branches and **hotfix branches**.
- **Hotfix branches** are used to create emergency fixes
- They are branched directly from a tagged release in the **master branch**, and when finished are merged back into both **master** and **develop** to make sure that the hotfix isn't accidentally lost when the next regular release occurs.

https://datasift.github.io/gitflow/IntroducingGitFlow.html

Phew, that was already a lot! But **how about Linux**?

- What is it?
- **Linux** is a name which broadly denotes a family of **free and open-source software operating system distributions built around the Linux kernel**.
- The defining component of a Linux distribution is the Linux kernel, an operating system kernel first released on September 17, 1991 by Linus Torvalds.

https://en.wikipedia.org/wiki/Linux

- How it's designed?
- A Linux-based system is a **modular** Unix-like operating system, deriving much of its basic design from principles established in Unix during the 1970s and 1980s.
- Such a system uses a **monolithic kernel**, the Linux kernel, which handles process control, networking, access to the peripherals, and file systems.
- **Device drivers** are either integrated directly with the kernel, or **added as modules** that are loaded while the system is running.
- Separate **projects** that interface with the kernel **provide much of the system's higher-level functionality**.
- The **GNU userland** is an important part of most Linux-based systems, providing the most common implementation of the C library, a popular **CLI shell**, and many of the common Unix tools which carry out many basic operating system tasks.
- The graphical user interface (or **GUI**) used by most Linux systems is built on top of an implementation of the **X Window System**.
- More recently, the Linux community seeks to advance to **Wayland** as the new display server protocol in place of **X11**.

Various layers within Linux, also showing separation between the userland and kernel space

User mode	**User applications**	For example, bash, LibreOffice, GIMP, Blender, 0 A.D., Mozilla Firefox, etc.				
	Low-level system components:	**System daemons**: *systemd*, *runit*, *logind*, *networkd*, *PulseAudio*, ...	**Windowing system**: *X11*, *Wayland*, *Mir*, *SurfaceFlinger* *(Android)*	**Other libraries:** *GTK+*, *Qt*, *EFL*, *SDL*, *SFML*, *FLTK*, *GNUstep*, etc.		**Graphics**: *Mesa*, *AMD Catalyst*, ...
	C standard library	`open()`, `exec()`, `sbrk()`, `socket()`, `fopen()`, `calloc()`, ... (up to 2000 subroutines) *glibc* aims to be POSIX/SUS-compatible, *uClibc* targets embedded systems, *bionic* written for Android, etc.				
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Network subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*				
Hardware (CPU, main memory, data storage devices, etc.)						

Installed components of a Linux system

A bootloader, for example GNU GRUB, LILO, SYSLINUX, or Gummiboot. This is a program that loads the Linux kernel into the computer's main memory, by being executed by the computer when it is turned on and after the firmware initialization is performed.

An init program, such as the traditional sysvinit and the newer systemd, OpenRC and Upstart. This is the first processlaunched by the Linux kernel, and is at the root of the process tree: in other terms, all processes are launched through init. It starts processes such as system services and login prompts (whether graphical or in terminal mode).

Software libraries, which contain code that can be used by running processes. On Linux systems using ELF-format executable files, the dynamic linker that manages use of dynamic libraries is known as ld-linux.so. If the system is set up for the user to compile software themselves, header files will also be included to describe the interface of installed libraries. Besides the most commonly used software library on Linux systems, the GNU C Library (glibc), there are numerous other libraries.

C standard library is the library needed to run standard C programs on a computer system, with the GNU C Library being the most commonly used. Several alternatives are available, such as the EGLIBC (which was used by Debian for some time) and uClibc (which was designed for uClinux).

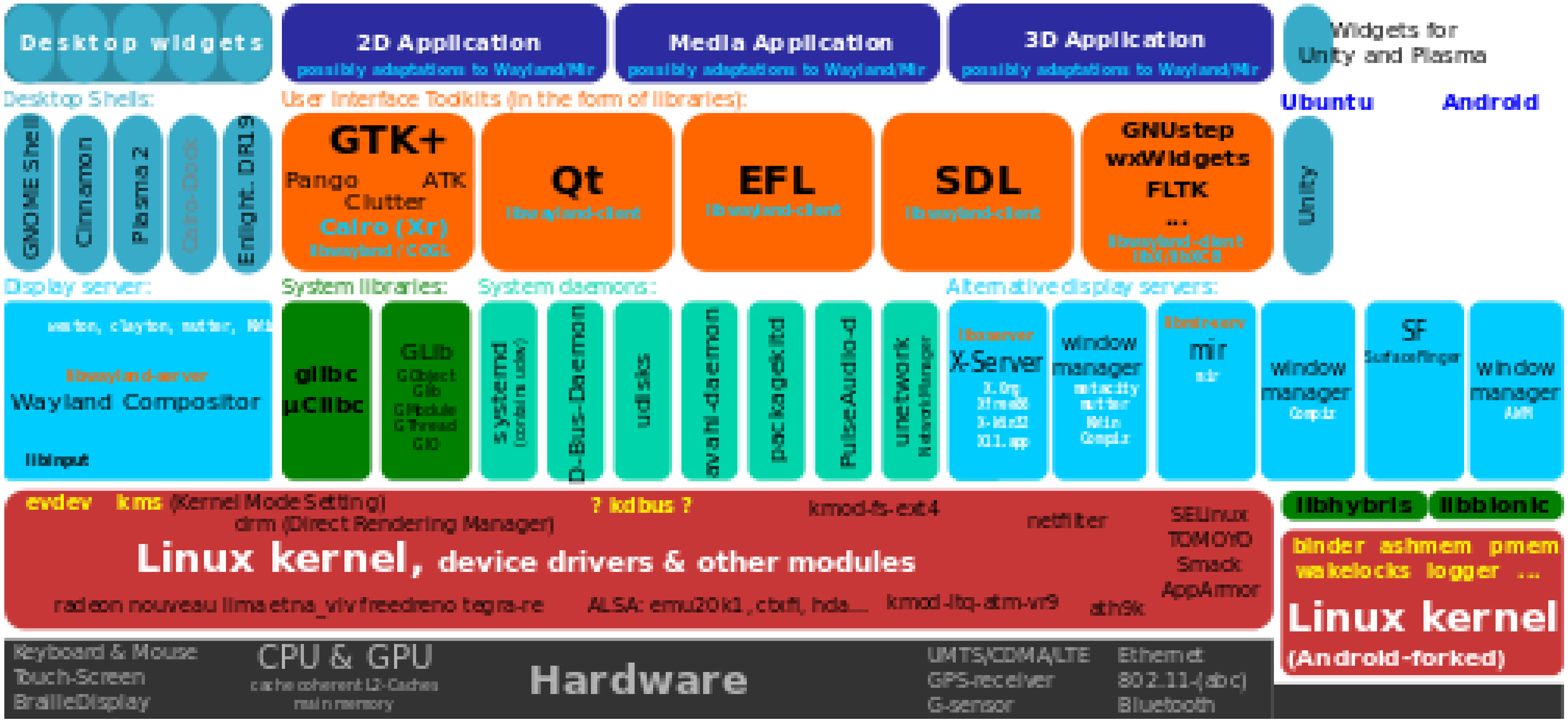
Widget toolkits are the libraries used to build graphical user interfaces (GUIs) for software applications. Numerous widget toolkits are available, including GTK+ and Clutter developed by the GNOME project, Qt developed by the Qt Project and led by Digia, and Enlightenment Foundation Libraries (EFL) developed primarily by the Enlightenment team.

User interface programs such as command shells or windowing environments.

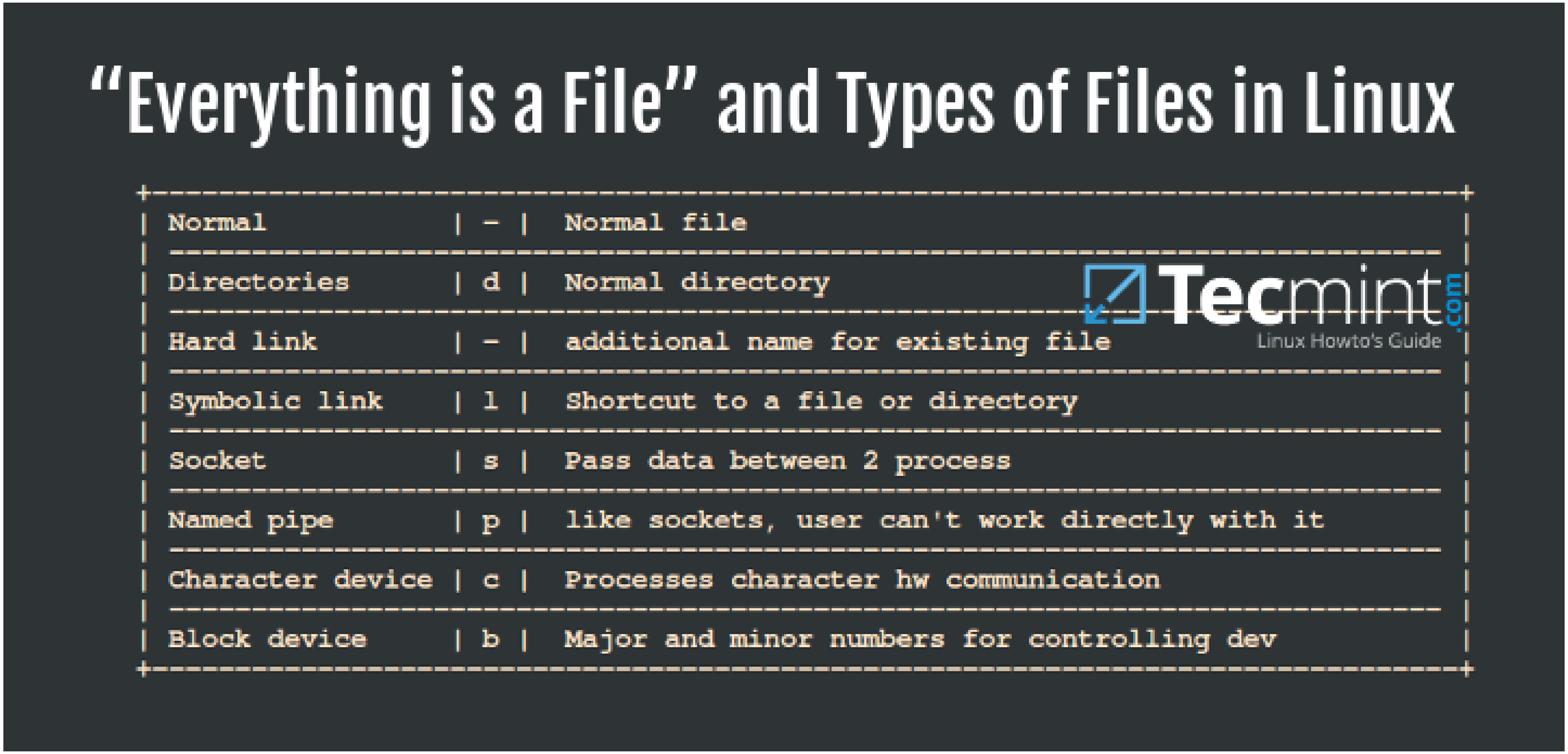
Desktop

Visible software components of the Linux desktop stack include the display server, widget engines, and some of the more widespread widget toolkits. There are also components not directly visible to end users, including D-Bus and PulseAudio.

See also: Desktop environmentand Linux adoption: Measuring desktop adoption



Everything is a file!



“Everything is a File” and Types of Files in Linux

Normal	-	Normal file
Directories	d	Normal directory
Hard link	-	additional name for existing file
Symbolic link	l	Shortcut to a file or directory
Socket	s	Pass data between 2 process
Named pipe	p	like sockets, user can't work directly with it
Character device	c	Processes character hw communication
Block device	b	Major and minor numbers for controlling dev

https://en.wikipedia.org/wiki/Unix_file_types

How many types of file are there in Linux/Unix?

By default Unix have only 3 types of files. They are..

1)Regular files

2)Directory files

3)Special files(This category is having 5 sub types in it.)

So in practical we have total 7 types(1+1+5) of files in Linux/Unix. And in Solaris we have 8 types.

You can see the file type indication at leftmost part of “ls -l” command.

Here are those files type.

`Regular file(-)`
`Directory files(d)`

`Special files:`

`Block file(b)`
`Character device file(c)`
`Named pipe` `file or just a` **`pipe file(p)`**
`Symbolic link file(l)`
`Socket file(s)`

https://www.linux.com/blog/file-types-linuxunix-explained-detail

File Types explained

Regular file

These are the files which are indicated with "-" in ls -l command output:

1. Readable file or
2. A binary file or
3. Image files or
4. Compressed files etc.

Directory file

These type of files contains metadata about regular files/folders/special files stored on a physical device.

And this type of files will be in blue in color with link greater than or equal 2.

Block device file

These files are storage hardware files.

Located in /dev

Character device file

Provides a serial stream of input or output.

Your terminals are classic example for this type of files.

Symbolic link files

These are linked files to other files.

They are either Directory/Regular File.

The inode number for this file and its parent files are same.

There are two types of link files available in Linux/Unix: **soft and hard link**.

Socket file

A socket file is used to pass information between applications for communication purpose

Pipes vs Sockets

- Both pipes and sockets handle byte streams, but they do it in different ways...

- pipes only exist within a specific host, and they refer to buffering between virtual files, or connecting the output / input of processes within that host. There are no concepts of packets within pipes.
- sockets packetize communication using IPv4 or IPv6; that communication can extend beyond localhost. Note that different endpoints of a socket can share the same IP address; however, they must listen on different TCP / UDP ports to do so.

- **Use pipes:**
- when you want to read / write data as a file within a specific server. If you're using C, you read() and write() to a pipe.
- when you want to connect the output of one process to the input of another process... see popen()
- **Use sockets**
- to send data between different IPv4 / IPv6 endpoints. Very often, this happens between different hosts, but sockets could be used within the same host
- BTW, you can use netcat or socat to join a socket to a pipe.