

# Open Source Development Course

## Continuous Integration and Delivery

---

Vojtěch Trefný

vtrefny@redhat.com

26. 3. 2020

 [twitter.com/vojtechtrefny](https://twitter.com/vojtechtrefny)

 [github.com/vojtechtrefny](https://github.com/vojtechtrefny)

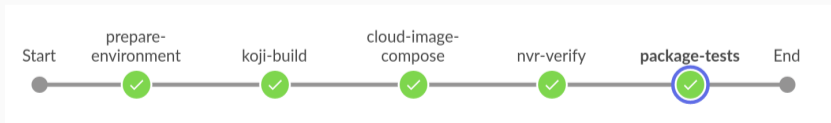
 [gitlab.com/vtrefny](https://gitlab.com/vtrefny)

# Pipeline

---

# CI/CD Pipeline

- Steps that need to be performed to test and deliver new version of the software.
- Defines what needs to be done: when, how and in what order.
- Steps can vary for every project.
- Multiple pipelines or steps can run in parallel.



Na obrázku je ukázka testovací pipeline z Fedora CI. Je zde vidět postup pro otestování nového buildu RPM balíčku, který se (v případě úspěšného projetí testů) dostane do Fedory. Podrobněji si tento proces popíšeme v části demo, kde jsou i podrobnější poznámky.

## 1. Testing environment

Preparation of the environment to run the tests: deploying containers, starting VMs...

## 2. Static Analysis

Finding defects by analyzing the code without running it.

## 3. Codestyle

Checking for violations of the language or project style guides.

## 4. Build

Building the project from source.

## 5. Tests

Running project test suite or test suites.

## 6. Packaging and Deployment

Building source archives, packages or container images.

Výše popsané kroky jsou jen příkladem CI pipeline. V různých projektech se mohou jednotlivé kroky lišit, některé chybět a naopak přibýt některé jiné, záleží podle typu projektu a použitého programovacího jazyka.
























Jednotlivé kroky nemusí být prováděny v daném pořadí a celá pipeline nemusí být lineární – kroky, které se dají paralelizovat se obvykle paralelizují pro urychlení celé operace – například testy a statická analýza mohou běžet souběžně zcela bez problémů.

Část „CD“, tedy deployment nebo delivery, je často samostatná a nemusí vždy proběhnout – u pull requestů se často použít jen testy a balíčky/image se řeší pouze při vydání.

# Testing Environment

---

# Testing Environment

Configuration Matrix	x86_64	i686	arm64
f_30	 	 	
f_31	 		 
f_rawhide	 		
centos_7	 		
debian_10	 	 	
debian_t	 		
rhel_8	 		

## 1. Preparation of VMs/containers to run the tests

We might want to run tests in different environments on multiple different distributions or architectures.

## 2. Installation of the test dependencies

Test dependencies are usually not covered by the project dependencies.

## 3. Getting the code

Clone the PR or get the latest code from the master branch.



Na obrázku je příklad testovací matice jednoho z našich projektů. Pro co nejlepší pokrytí testy se snažíme testovat na různých distribucích i architekturách. Každá kombinace představuje jeden virtuální stroj, na kterém běží testy.

Většina projektů si vystačí se spouštěním testů v kontejnerech nebo jiných jednodušších prostředích, ale někdy je vhodné testovat na co nejširší škále systémů. Takové testy pak mohou pomoci odhalit problémy se závislostmi (chybějící knihovny nebo staré verze knihoven na „stabilních“, tedy starších, distribucích a v případě různých architektur je takto možné odhalit například chyby při ukládání dat s nesprávnou endiánitou.

Rozsáhlejší testovací matice jsou časté také u webových projektů, kdy se testuje na mnoha různých webových prohlížečích.

# Static Analysis

---

- Tools that can identify potential bugs by analyzing the code without running it.
- Can detect problems not covered by the test suite – corner cases, error paths etc.
  - Coverity (C/C++, Java, Python, Go... )<sup>1</sup>
  - Cppcheck (C/C++)<sup>2</sup>
  - Pylint (Python)<sup>3</sup>
  - RuboCop (Ruby)<sup>4</sup>

---

<sup>1</sup> <https://scan.coverity.com>

<sup>2</sup> <http://cppcheck.sourceforge.net/>

<sup>3</sup> <https://www.pylint.org>

<sup>4</sup> <https://docs.rubocop.org>

**Error: USE\_AFTER\_FREE (CWE-825):**

libblockdev-2.13/src/plugins/lvm-dbus.c:1163: freed\_arg: "g\_free"  
frees "output".

libblockdev-2.13/src/plugins/lvm-dbus.c:1165: pass\_freed\_arg: Passing freed  
pointer "output" as an argument to "g\_set\_error".

```
# 1163|         g_free (output);  
# 1164|         if (ret == 0) {  
# 1165|->             g_set_error (error, BD_LVM_ERROR, BD_LVM_ERROR_PARSE,  
# 1166|                 "Failed to parse number from output: '%s'",  
# 1167|                 output);
```

Ukázka výstupu nástroje coverity na Céčkovém zdrojáku. Statická analýza zde odhalila chybu, která by jinak vedla k pádu programu. Paměť uvolněná na řádce 1163 se používá na řádce 1165. Protože se jedná o chybový stav dané funkce je možné, že tento stav není pokrytý jednotkovými testy, které by tak tento problém neodhalily.

## Code Style

---

- Coding conventions – naming, code lay-out, comment style. . .
- Language specific (PEP 8<sup>5</sup>), project specific (Linux kernel coding style<sup>6</sup>) or library/toolkit specific (GTK coding style<sup>7</sup>).
- Automatic checks using specific tools (pycodestyle) or (partially) by the static analysis tools.

---

<sup>5</sup> <https://www.python.org/dev/peps/pep-0008/>

<sup>6</sup> <https://www.kernel.org/doc/html/v5.3/process/coding-style.html>

<sup>7</sup> <https://developer.gnome.org/programming-guidelines/stable/c-coding-style.html.en>

Pokud budete (nejen v rámci tohoto předmětu) přispívat do nějakého open source projektu, vždy si napřed zjistěte, jestli má nějaký code style. I pokud nemá žádný formalizovaný, pokuste se dodržovat styl, jakým je kód napsaný. Nedodržení zavedených stylů případného reviewera vašeho pull requestu minimální rozladí, v některých případech bude takový příspěvek automaticky zamítnut bez ohledu na to, zda je jinak přínosný či ne.





Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Readability counts.

**Tim Peters**

The Zen of Python

# Linux kernel coding style

<https://www.kernel.org/doc/html/v4.10/process/coding-style.html>

Don't put multiple statements on a single line unless you have something to hide:

```
if (condition) do_this;  
    do_something_everytime;
```

The preferred form for allocating an array is the following:

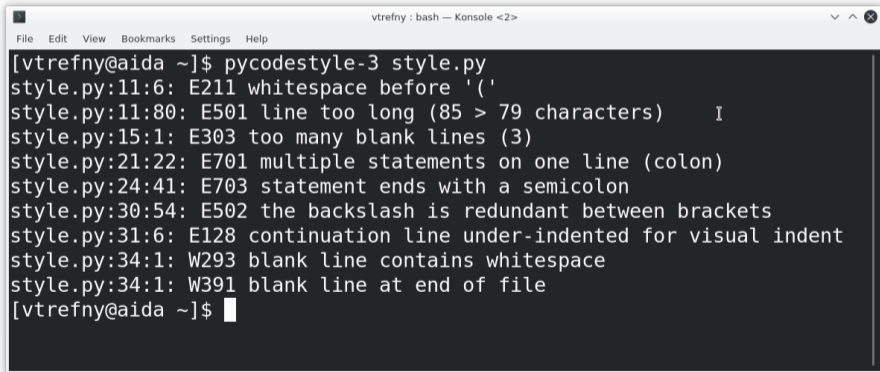
```
p = kcalloc_array(n, sizeof(...), ...);
```

Do not unnecessarily use braces where a single statement will do.

```
if (condition)  
    action();
```

[https:](https://gist.github.com/vojtechtrefny/435737417be003873a7f94aa7d53c4d2)

[//gist.github.com/vojtechtrefny/435737417be003873a7f94aa7d53c4d2](https://gist.github.com/vojtechtrefny/435737417be003873a7f94aa7d53c4d2)

A terminal window titled "vtrefny : bash -- Konsole <2>" with a menu bar (File, Edit, View, Bookmarks, Settings, Help). The terminal shows the command "pycodestyle-3 style.py" and its output: "style.py:11:6: E211 whitespace before '('", "style.py:11:80: E501 line too long (85 > 79 characters)", "style.py:15:1: E303 too many blank lines (3)", "style.py:21:22: E701 multiple statements on one line (colon)", "style.py:24:41: E703 statement ends with a semicolon", "style.py:30:54: E502 the backslash is redundant between brackets", "style.py:31:6: E128 continuation line under-indented for visual indent", "style.py:34:1: W293 blank line contains whitespace", and "style.py:34:1: W391 blank line at end of file". The prompt "[vtrefny@aida ~]" is shown at the end.

```
vtrefny : bash -- Konsole <2>
File Edit View Bookmarks Settings Help
[vtrefny@aida ~]$ pycodestyle-3 style.py
style.py:11:6: E211 whitespace before '('
style.py:11:80: E501 line too long (85 > 79 characters)
style.py:15:1: E303 too many blank lines (3)
style.py:21:22: E701 multiple statements on one line (colon)
style.py:24:41: E703 statement ends with a semicolon
style.py:30:54: E502 the backslash is redundant between brackets
style.py:31:6: E128 continuation line under-indented for visual indent
style.py:34:1: W293 blank line contains whitespace
style.py:34:1: W391 blank line at end of file
[vtrefny@aida ~]$
```



**pep8speaks** commented on 18 Feb



Hello @vojtechrefny! Thanks for updating this PR. We checked the lines you've touched for [PEP 8](#) issues, and found:

- In the file `copr_builder/copr_builder.py` :

| [Line 31:54: E261](#) at least two spaces before inline comment

Ačkoli je PEP8 standard, ne všechny Python projekty jej používají, u některých může být zaveden jiný code style. I projekty, které PEP8 dodržují nemusí striktně vyžadovat dodržování všech jeho pravidel. Například omezení na 79 znaků na řádek je velice kontroverzní a mnohé projekty si stanovují vlastní omezení.

Na obrázcích výše je výstup kontroly pomocí programu `pycodestyle` a kontrola pull requestu na GitHubu pomocí aplikace `pep8speaks`, kterou je možné u open source projektů zapnout zdarma.

- Documentation might be checked in the same way code is.
- Similar style documents and tools for checking documentations exist (for example PEP 257<sup>8</sup> and pydocstyle<sup>9</sup> for Python).
- In some cases wrong or missing documentation (docstrings in the code) can lead to a broken build or missing features.

---

<sup>8</sup> <https://www.python.org/dev/peps/pep-0257/>

<sup>9</sup> <http://www.pydocstyle.org>

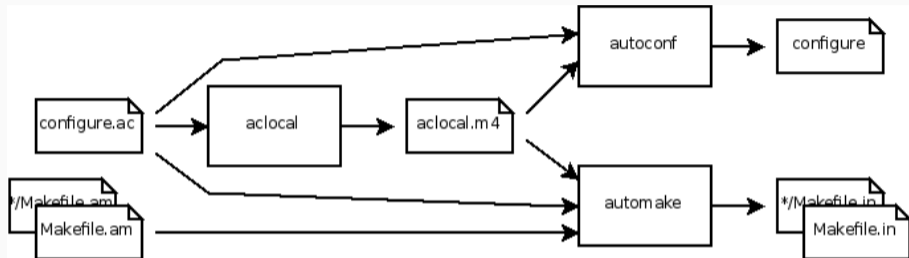
## Build

---

- Building the project, a preparation to run the test suite.
- Depends on language – mostly no-op for interpreted languages, more complicated for compiled ones.
- Build in the CI environment can detect issues with dependencies.
- Builds on different architectures can help detect issues related to endianness or data types sizes.



- Helps creating portable source packages.
- Two steps:
  - configure (scans the build environment)
  - make (compiles the source)
- Complicated for developers, easy for users.
- Takes care of dependency checking, dynamic linking, installation destinations etc.



I saw a book entitled "Die GNU Autotools" and I thought "My feelings exactly". Turns out the book was in German.

Tim Martin<sup>10</sup>

<sup>10</sup><https://twitter.com/timmartin2/status/23365017839599616>

Image source: <https://developer.gnome.org/anjuta-build-tutorial/stable/create-autotools.html.en>

Složitost jednotlivých build systémů se liší podle použitého jazyka a frameworku. Nejsložitější je nejspíše u kompilovaných jazyků. Jako příklad (částečně odstrašující) může posloužit rodina nástrojů GNU Autotools, na kterou narazíte především u projektů napsaných v jazyce C. Kromě samotné kompilace daného projektu mají Autotools na starosti především zjištění informací o systému, kontrolu závislostí a jejich správné přilinkování k danému projektu. Cílem Autotools je aby pro uživatele bylo snadné daný projekt/program nainstalovat a obvykle díky tomu stačí obecné `./configure && make && make install`, ale za cenu velké složitosti pro programátory, kteří musí s Autotools často bojovat a díky tomu se netěší zrovna velké oblíbenosti. Velkou složitostí se ale vyznačují všechny pokročilé build systémy a pokud budete přispívat do některého linuxového nebo GNU projektu, tak na Autotools stoprocentně narazíte.

# Tests

---

- Running tests that are part of the project.
- New tests should be part of every change to the codebase.
  - New features require new unit and integration tests.
  - Bug fixes should come with a regression test.
- For some project (like libraries) running test suites of their users might be an option.

- Code coverage (or Test coverage) represents how much of the code is covered by the test suite.
- Usually percentual value that shows how many lines of the code were “visited” by the test.
- Generally a check that all functions and branches are covered by the suite.
- Used as a measure of the test suite “quality”.

# Coverage

```
1 def div(a, b):
2     if b == 0:
3         raise ValueError
4     else:
5         return a / b
6
7 assert div(2, 2) == 1
```

```
$ coverage3 report -m
```

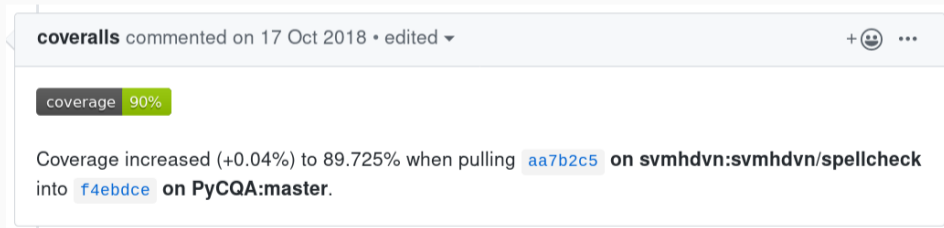
Name	Stmts	Miss	Cover	Missing
div.py	5	1	80%	3

Resulting coverage is 80 %, because 1 of 5 statements is not covered.

Coverage je občas vnímáno jako jediná správná míra kvality testů a mnoho projektů se hodí za 100% coverage, ale občas se přitom zapomíná, že plné pokrytí neznamena, že testy jsou skutečně kvalitní. Na obrázku je nadsazená ukázka testu, který pokrývá 80 % kódu, ale kdyby daná funkce vracela vždy 1, tak je pokrytí stejné, ale test neodhalí, že funkce ve skutečnosti vůbec nefunguje. Na druhou stranu je vhodné coverage sledovat a ujistit se, že testy pokrývají skutečně všechny stavy, do kterých se daná funkce může dostat.



- Automated coverage tests might be part of the CI.
- Decrease in coverage can be viewed as a reason to reject contribution to the project.



A screenshot of a GitHub comment from user 'coveralls' dated 17 Oct 2018. The comment contains a coverage report for a pull request. The report shows a coverage of 90% for the current state. The text below the bar chart states: 'Coverage increased (+0.04%) to 89.725% when pulling `aa7b2c5` on `svmhdvn:svmhdvn/spellcheck` into `f4ebdce` on `PyCQA:master`.'

**coveralls** commented on 17 Oct 2018 • edited ▾ +😊 ⋮

coverage 90%

Coverage increased (+0.04%) to 89.725% when pulling `aa7b2c5` on `svmhdvn:svmhdvn/spellcheck` into `f4ebdce` on `PyCQA:master`.

Obecně, pokud budete přispívat do nějaké projektu, součástí vašeho příspěvku by měly vždy být testy. Obzvláště, pokud přidáváte nějakou novou funkcionalitu. Pokud projekt sleduje coverage svých testů a po přidání vašich změn coverage klesne, může to být důvod k zamítnutí navrhovaných změn.

# Delivery and Deployment

---

## Packaging and publishing

- **Delivery** – releasing new changes quickly and regularly (daily, weekly...).
- **Deployment** – delivery with automated push to production, without human interaction.
  
- Usually after merging the changes, not for the PRs.
- Building packages, container images, ISO images. . .
- Built packages can be used for further testing (manually by the Quality Assurance or in another CI infrastructure) or directly pushed to production or included in testing/nightly builds of the project.

„CD“ část celého CI/CD procesu hodně záleží na daném projektu a jeho výstupech. Pokud projekt jen poskytuje knihovnu nebo GUI aplikaci, výstupem může být prostě tarball se zdrojovými kódy nebo balíček pro určitou distribuci a poslední krok už se řeší zcela jinde (například v distribuci jako samostatný CI/CD proces). Projekt může ale také nabízet nightly buildy, tedy každodenní buildy s nejnovějšími změnami a pak je třeba řešit, jak tyto buildy uživatelům zpřístupnit. Způsoby opět záleží na okolnostech – Python projekty mohou nabízet daily buildy v PyPI, Fedora má Copr repozitáře, Ubuntu PPA atp.

Proces se může například lišit i v rámci jednoho projektu – například u Fedory je možné do aktuálně vyvíjené verze (rawhide) publikovat nové verze balíčků okamžitě prakticky bez omezení, u stabilních distribucí musí projít ještě dalším testováním a jsou umisťovány do speciálního repozitáře a teprve po 14 dnech se dostanou ke všem uživatelům. V některých případech (u projektů zaměřených na vysokou stabilitu) může být tento postup i částečně manuální a nová verze musí projít manuálním testováním a schválením.

# CI Tools

Demo

---

- Probably most popular CI service nowadays.
- Can be integrated in your projects on GitHub.
- Free for opensource projects.
- Configured using `.travis.yml` file in the project
- <https://travis-ci.org>





## All checks have passed

1 successful check

[Hide all checks](#)



**Travis CI - Pull Request** Successful in 44s — Build Passed

[Details](#)



## This branch has no conflicts with the base branch


Merging can be performed automatically.


Merge pull request




or view [command line instructions](#).



vojtechtrefny / copr-builder  build: passing

[Current](#) [Branches](#) [Build History](#) [Pull Requests](#) More options 


 **Pull Request #41** Add a first simple test for copr\_builder

Parsing of config files is covered.


[↔ Commit ef796cc](#)


[🔗 #41: Add a first simple test for copr\\_builder](#)


[📁 Branch master](#)


 **Vojtech Trefny**

---

 Python

 #25 passed

 Ran for 44 sec

 3 days ago

[Restart build](#)

Travis je oblíbený především díky velmi snadné integraci u projektů hostovaných na GitHubu nebo GitLabu. Na GitHubu lze Travis přidat do projektu v Marketplace téměř bezpracně. Samotné spouštění testů se pak nastavuje pomocí YAML souboru umístěného v daném repositáři. Ukázka spouštění testů u velice jednoduchého Python projektu: <https://github.com/vojtechtrfny/copr-builder/blob/master/.travis.yml> – projekt má vlastní Makefile, který umožňuje spouštění testů a stačí tedy říct Travisu, jaké závislosti je třeba doinstalovat a jakými příkazy testy pustit a na jaké branchi (branchích).

- Automation system, not a “true” CI/CD tool.
- Can automatically run given tasks on a node or set of nodes.
- Tasks can be started on time basis or triggered by an external event (like new commit or PR on GitHub).
- <https://jenkins.io/>



- Complex CI system with task to deliver an “Always Ready Operating System”.
- Packages are tested after every change and “gated” if the CI pipeline fails.
- The goal is to prevent breaking the distribution. CI will stop the broken package before it can affect the distribution.





package-tests - 5m 19s



✓ > Currently checking if package tests exist — Print Message	<1s
✓ > Deleting old packages	<1s
✓ > Cloning <a href="https://src.fedoraproject.org/rpms/vim/">https://src.fedoraproject.org/rpms/vim/</a> into the f30 branch	3s
✓ > rpm -q standard-test-roles — Checking if standard-test-roles are installed	<1s
✓ > Getting list of tags	2s
✓ > Print Message	<1s
✓ > Print Message	<1s
✓ > CI Notifier	5s
✓ > Print Message	<1s
✓ > CI Notifier	5s
✓ > Creating directory /workDir/workspace/fedora-f30-build-pipeline/package-tests	<1s
✓ > /tmp/package-test.sh — Shell Script	4m 33s
✓ > logs/ — Verify if file exists in workspace	<1s

*Prepare Environment* je příprava testovacího prostředí.

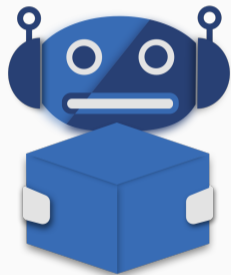
*Koji Build* z vývojářem pushnutých změn vybuildí RPM balíček (případně více balíčků).

*Cloud Image Compose* vytváří z nově sestavených balíčků a z existující distribuce cloud image Fedory, na kterém poběží testy (testy se použijí oproti nainstalovaným balíčkům, aby běžely ve stejném prostředí, v jakém následně bude daný balíček uživateli používán).

*NVR verify* kontroluje správnost verze balíčku. Mimo jiné také, aby se zajistilo, že půjde Fedora upgradovat (novější vydání musí mít vyšší verzi balíčků).

*Package Tests* jsou již samotné testy. Testy jsou buď součástí samotného balíčku, nebo jsou umístěny v git repozitáři daného Fedora balíčku. Ten také obsahuje YAML soubor popisující závislosti pro testy a způsob jejich spouštění. Příklad pro balíček libblockdev: <https://src.fedoraproject.org/rpms/libblockdev/blob/master/f/tests/tests.yml>.

- Tool for integrating upstream projects to Fedora.
- RPM packages are automatically build on every pull request.
- New releases can be automatically build and pushed to Fedora.





**packit-as-a-service** bot commented 24 days ago



Congratulations! One of the builds has completed. 🎉

You can install the built RPMs by following these steps:

- `sudo yum install -y dnf-plugins-core` on RHEL 8
- `sudo dnf install -y dnf-plugins-core` on Fedora
- `dnf copr enable packit/storaged-project-blivet-gui-157`
- And now you can install the packages.

Please note that the RPMs should be used only in a testing environment.



Packit je ukázka Continuous Delivery nástroje. Podobně jako Travis je možné využívat ho jako službu na GitHubu, ale je možné ho používat také jako samostatnou aplikaci na vlastním stroji. Packit na GitHubu automaticky sestaví RPM balíčky pro zvolený projekt (podobně jako Travis k tomu potřebuje YAML soubor s konfigurací) a vytvoří také Copr repozitář, který se dá použít pro další testování. Packit umožňuje nové releasy automaticky pushovat do aktuální vývojové verze Fedory (rawhide) a pracuje se na dalších vlastnostech.

# Questions

---

Thank you for your attention.

<https://github.com/crocs-muni/open-source-development-course>

Budete-li mít nějaké dotazy, můžete se na mě obrátit také mailem na [vtrefny@redhat.com](mailto:vtrefny@redhat.com)