



Published in devconnected—DevOps, Sysadmins & Engineering



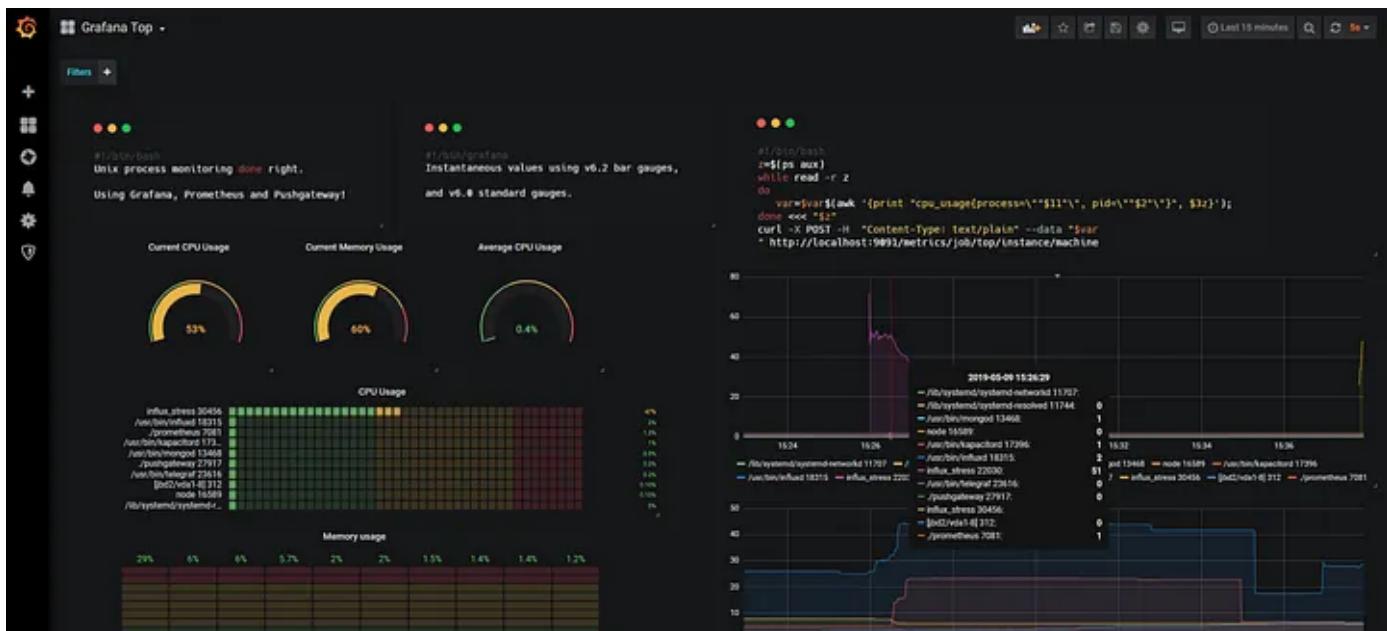
Antoine Solnichkin

Follow

May 18, 2019 · 13 min read · Listen



Monitoring Linux Processes using Prometheus and Grafana



Open in app ↗

Sign up

[Sign In](#)



Search Medium



Whether you are a Linux system administrator or a DevOps engineer, you spend a lot of time **tracking performance metrics on your servers**.

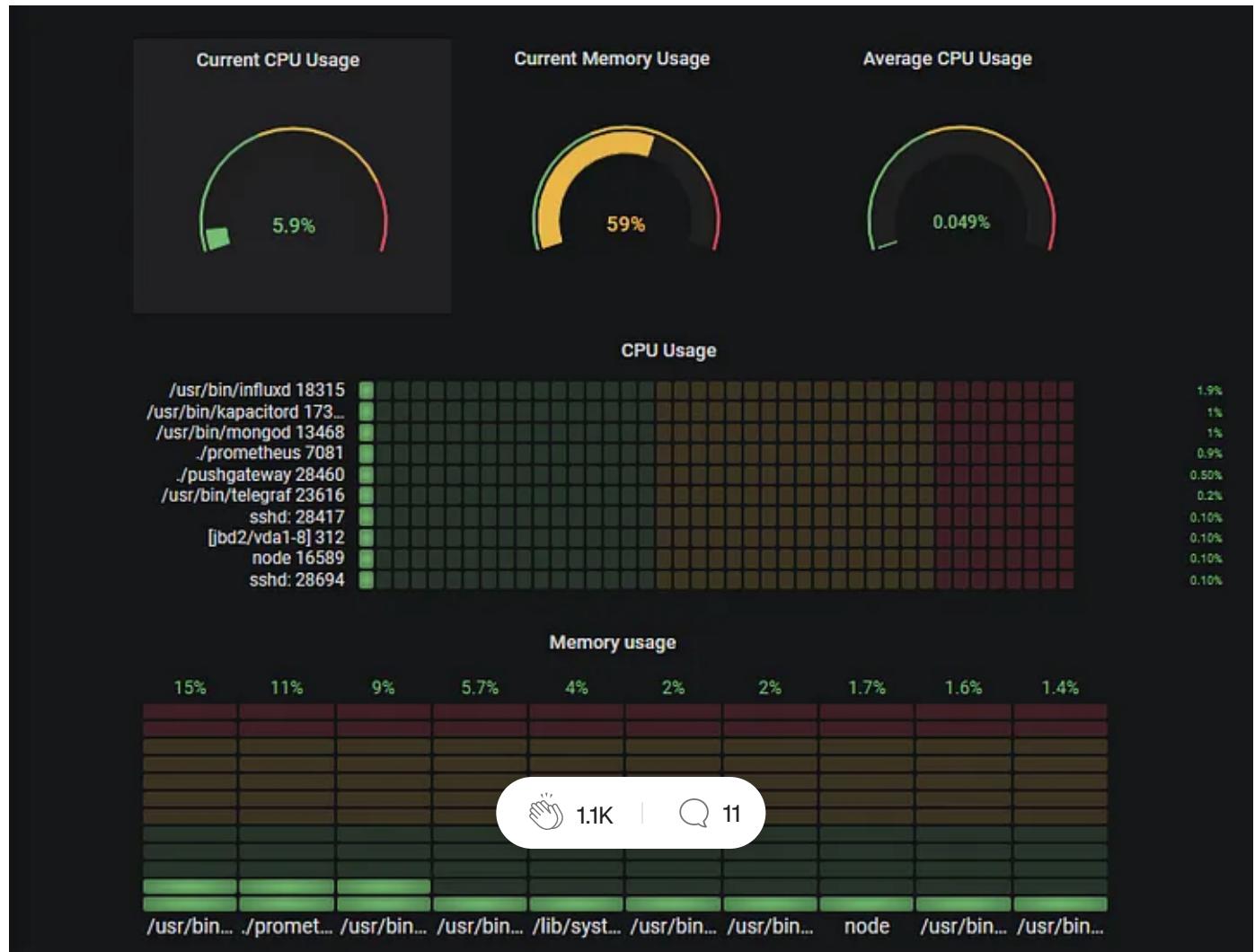
You may sometimes have instances that are running very slow without having any real clues of what the issues might be.

You may have **unresponsive instances** that might block you from running remote commands such as top or htop on them.

You may have a **simple bottleneck** on your server, but you cannot identify it in a simple and quick way.

What if we had a complete dashboard that would help us tracking general performance as well as individual processes?

See it live here : <http://grafana.devconnected.com/d/nZMDMoiZk/grafana-top?orgId=1&refresh=5s>



The goal of this tutorial is to create a **complete monitoring dashboard** for Linux sysadmins.

This dashboard will showcase different panels that are entirely customizable and scalable to multiple instances for distributed architectures.

What You Will Learn

Before jumping right into this technical journey, let's have a quick look at everything that you are going to learn by reading this article:

- Understanding current state-of-the-art ways to monitor process performance on Unix systems;
- Learn how to install the latest versions of **Prometheus v2.9.2**, **Pushgateway v0.8.0** and **Grafana v6.2**;
- Build a simple **bash script** that exports metrics to Pushgateway;
- Build a **complete Grafana dashboard** including the latest panels available such as the ‘Gauge’ and the ‘Bar Gauge’;
- Bonus : implementing **ad-hoc filters** to track individual processes or instances.

Now that we have an overview of everything that we are going to learn, and without further due, let's have an introduction on what's currently existing for Unix systems.

Unix Process Monitoring Basics

When it comes to process monitoring for Unix systems, you have multiple options.

The most popular one is probably ‘**top**’.

This command is widely used among sysadmins and is probably the first command run when a performance bottleneck is detected on a system (if you can access it of course!)

Unix top command

top - 16:17:31 up 127 days, 4:38, 5 users, load average: 0.21, 0.19, 0.13										
Tasks: 123 total, 1 running, 81 sleeping, 0 stopped, 0 zombie										
%Cpu(s): 4.7 us, 2.2 sy, 0.0 ni, 93.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
KiB Mem : 4039588 total, 137876 free, 2264716 used, 1636996 buff/cache										
KiB Swap: 0 total, 0 free, 0 used. 2006124 avail Mem										
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+ COMMAND
13468	mongodb	20	0	1414808	362516	416	S	3.7	9.0	1808:54 mongod
18315	root	20	0	3299048	622164	26084	S	1.0	15.4	3419:47 influxd
7081	schkn	20	0	715428	432856	16260	S	0.7	10.7	53:38.80 prometheus
22433	schkn	20	0	107984	5688	4684	S	0.7	0.1	0:01.25 sshd
23616	telegraf	20	0	1200720	28436	6660	S	0.7	0.5	389:22.88 telegraf
17644	schkn	20	0	113132	14432	7880	S	0.3	0.4	0:06.95 pushgateway
27819	schkn	20	0	44544	3908	3292	R	0.3	0.1	0:00.61 top
1	root	20	0	225544	6812	4188	S	0.0	0.2	3:33.77 systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.28 kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00 kworker/0:0H
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00 mm_percpu_wq
7	root	20	0	0	0	0	S	0.0	0.0	2:20.14 ksoftirqd/0
8	root	20	0	0	0	0	I	0.0	0.0	71:32.57 rcu_sched
9	root	20	0	0	0	0	I	0.0	0.0	0:00.00 rcu_bh
10	root	rt	0	0	0	0	S	0.0	0.0	0:00.74 migration/0
11	root	rt	0	0	0	0	S	0.0	0.0	0:23.83 watchdog/0
12	root	20	0	0	0	0	S	0.0	0.0	0:00.01 cpuhp/0
13	root	20	0	0	0	0	S	0.0	0.0	0:00.01 cpuhp/1
14	root	rt	0	0	0	0	S	0.0	0.0	0:25.11 watchdog/1
15	root	rt	0	0	0	0	S	0.0	0.0	0:00.73 migration/1
16	root	20	0	0	0	0	S	0.0	0.0	1:47.74 ksoftirqd/1
18	root	0	-20	0	0	0	I	0.0	0.0	0:00.00 kworker/1:0H
19	root	20	0	0	0	0	S	0.0	0.0	0:00.00 kdevtmpfs

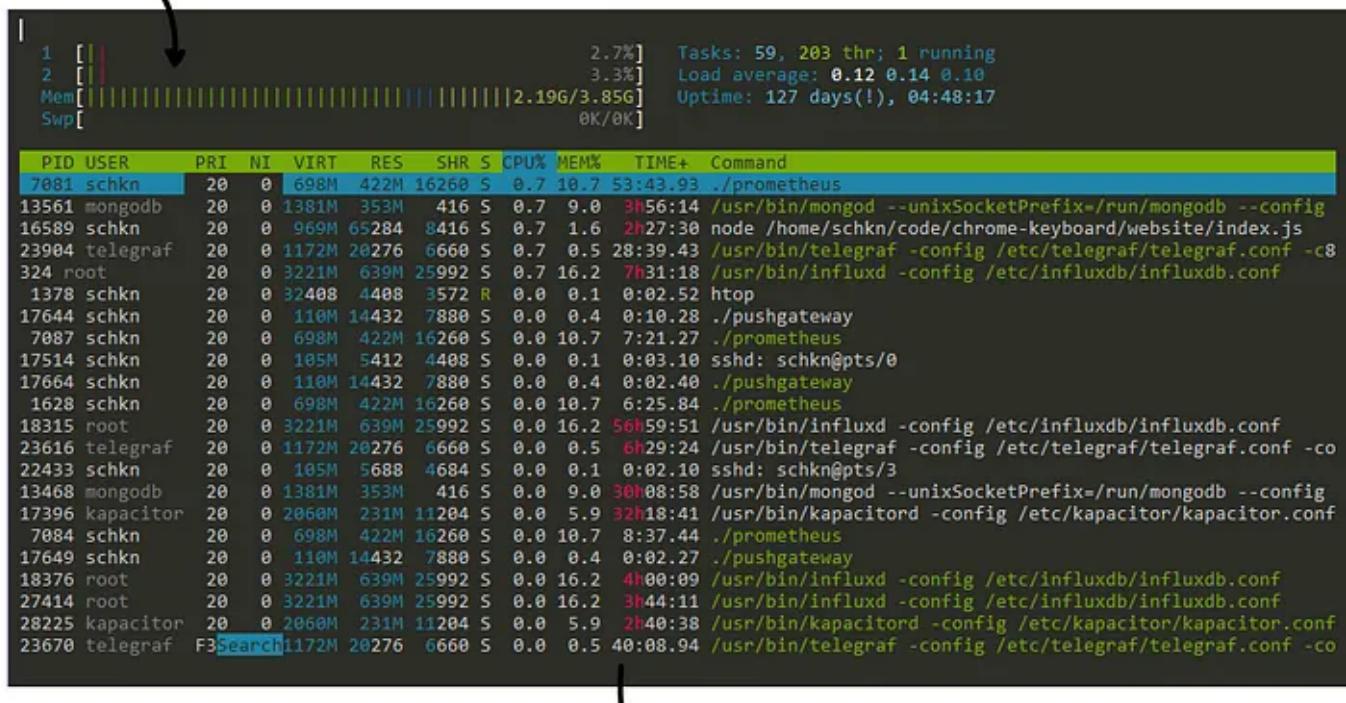
The top command is already pretty readable, but there is a command that makes everything even more readable than that: **htop**.

Htop provides the same set of functionalities (CPU, memory, uptime..) as top but in a colorful and pleasant way.

Htop also provides gauges that reflects current system usage.

Gauges

Unix htop command



Invidivual process details

Knowing that those two commands exist, why would we want to build yet another way to monitor processes?

The main reason would be **system availability**: in case of a system overload, you may have no physical or remote access to your instance.

By externalizing process monitoring, you can analyze what's causing the outage without accessing the machine.

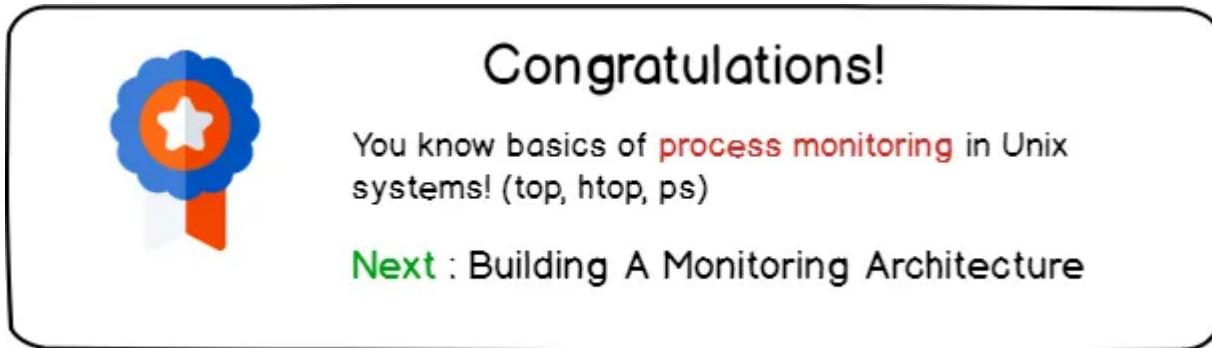
Another reason is that **processes get created and killed all the time**, often by the kernel itself.

In this case, running a top command would give you zero information as it would be too late for you to catch who's causing performance issues on your system.

You would have to dig into kernel logs to see what has been killed.

With a monitoring dashboard, you can simply go back in time and see which process was causing the issue.

Now that you know why we want to build this dashboard, let's have a look at the architecture put in place in order to build it.



Detailing Our Monitoring Architecture

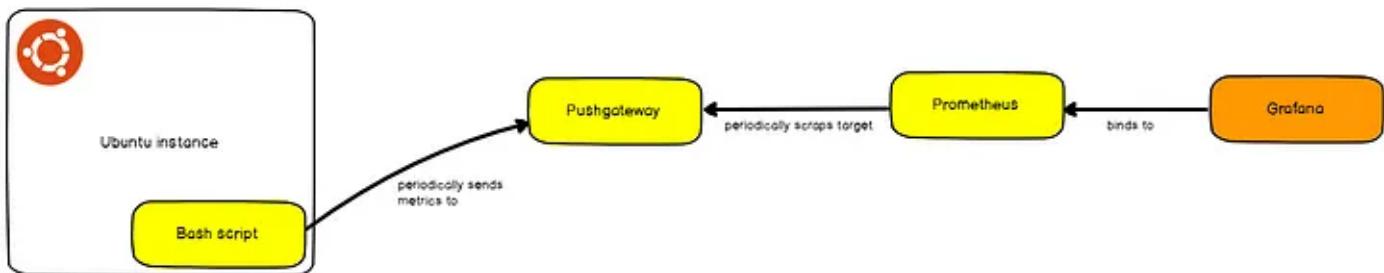
Before having a look at the architecture that we are going to use, we want to use a solution that is:

- **Resource cheap** : i.e not consuming many resources on our host;
- **Simple to put in place** : a solution that doesn't require a lot of time to instantiate;
- **Scalable** : if we were to monitor another host, we can do it quickly and efficiently.

Those are the points we will keep in mind throughout this tutorial.

The detailed architecture we are going to use today is this one:

Process Monitoring Architecture



Our architecture makes use of four different components:

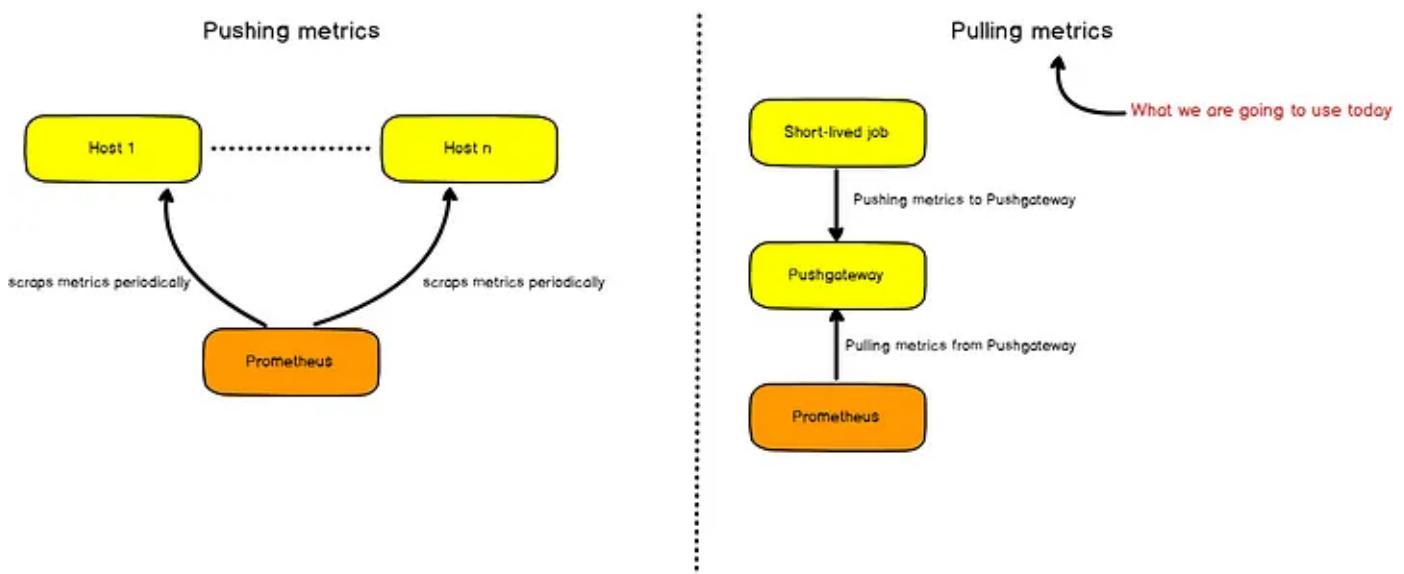
- A bash script used to send periodically metrics to the Pushgateway;
- **Pushgateway**: a metrics cache used by individual scripts as a target;
- **Prometheus**: that instantiates a time series database used to store metrics. Prometheus will scrape Pushgateway as a target in order to retrieve and store metrics;
- **Grafana**: a dashboard monitoring tool that retrieves data from Prometheus via PromQL queries and plot them.

For those who are quite familiar with Prometheus, you already know that Prometheus scraps metrics exposed by HTTP instances and stores them.

In our case, the bash script has a very tiny lifespan and it doesn't expose any HTTP instance for Prometheus.

This is why we have to use the Pushgateway ; designed for **short-lived jobs**, Pushgateway will cache metrics received from the script and expose them to Prometheus.

Gathering metrics with Prometheus





Congratulations!

You know have a complete overview of a classic
Prometheus - Pushgateway - Grafana
architecture

Next : Installing all the tools!

Installing The Different Tools

Now that you have a better idea of what's going on in our application, let's install the different tools needed.

a — Installing Pushgateway

In order to install Pushgateway, run a simple wget command to get the latest binaries available.

```
wget
https://github.com/prometheus/pushgateway/releases/download/v0.8.0/pushgateway-0.8.0.linux-amd64.tar.gz
```

Now that you have the archive, extract it, and run the executable available in the pushgateway folder.

```
> tar xvzf pushgateway-0.8.0.linux-amd64.tar.gz
> cd pushgateway-0.8.0.linux-amd64/
> ./pushgateway &
```

As a result, your Pushgateway should start as background process.

```
me@schkn-ubuntu:~/softs/pushgateway/pushgateway-0.8.0.linux-amd64$  
./pushgateway &  
[1] 22806  
me@schkn-ubuntu:~/softs/pushgateway/pushgateway-0.8.0.linux-amd64$  
INFO[0000] Starting pushgateway (version=0.8.0, branch=HEAD,  
revision=d90bf3239c5ca08d72ccc9e2e2ff3a62b99a122e)  
source="main.go:65"INFO[0000] Build context (go=go1.11.8,  
user=root@00855c3ed64f, date=20190413-11:29:19)  
source="main.go:66"INFO[0000] Listening on :9091. source="main.go:108"
```

Nice!

From there, Pushgateway is listening to incoming metrics on port 9091.

b — Installing Prometheus

As described in the ‘Getting Started’ section of Prometheus’s website, head over to <https://prometheus.io/download/> and run a simple wget command in order to get the Prometheus archive for your OS.

```
wget  
https://github.com/prometheus/prometheus/releases/download/v2.9.2/prometheus-2.9.2.linux-amd64.tar.gz
```

Now that you have the archive, extract it, and navigate into the main folder:

```
> tar xvzf prometheus-2.9.2.linux-amd64.tar.gz  
> cd prometheus-2.9.2.linux-amd64/
```

As stated before, Prometheus scraps ‘targets’ periodically to gather metrics from them. Targets (Pushgateway in our case) need to be configured via Prometheus’s configuration file.

```
> vi prometheus.yml
```

In the ‘global’ section, modify the ‘*scrape_interval*’ property down to one second.

```
global:  
  scrape_interval:      1s # Set the scrape interval to every 1 second.
```

In the ‘*scrape_configs*’ section, add an entry to the targets property under the static_configs section.

```
static_configs:  
  - targets: ['localhost:9090', 'localhost:9091']
```

Exit vi, and finally run the prometheus executable in the folder.

Prometheus should start when launching the final prometheus command. To ensure that everything went correctly, you can head over to <http://localhost:9090/graph>.

If you have access to Prometheus’s web console, it means that everything went just fine.

You can also verify that Pushgateway is correctly configured as a target in ‘*Status*’ > ‘*Targets*’ in the Web UI.

Prometheus Web Console

The screenshot shows the Prometheus Web Console interface. At the top, there is a navigation bar with links for Prometheus, Alerts, Graph, Status, and Help. Below the navigation bar is a section with a checkbox labeled "Enable query history". A text input field is present with the placeholder "Expression (press Shift+Enter for newlines)". Below the input field are two buttons: "Execute" (blue) and "- insert metric at cursor -" (grey). Underneath these buttons are two tabs: "Graph" (selected) and "Console". In the "Graph" tab, there is a search bar containing the text "Moment" with navigation arrows on either side. Below the search bar is a section titled "Element" with the message "no data". At the bottom left of the interface is a blue button labeled "Add Graph".

c — Installing Grafana

Last not but least, we are going to install Grafana v6.2. Head over to <https://grafana.com/grafana/download/beta>.

As done before, run a simple wget command to get it.

```
> wget https://dl.grafana.com/oss/release/grafana_6.2.0-beta1_amd64.deb
> sudo dpkg -i grafana_6.2.0-beta1_amd64.deb
```

Now that you have extracted the deb file, grafana should run as a service on your instance.

You can verify it by running the following command:

```
> sudo systemctl status grafana-server
● grafana-server.service - Grafana instance
  Loaded: loaded (/usr/lib/systemd/system/grafana-server.service;
disabled; vendor preset: enabled)
  Active: active (running) since Thu 2019-05-09 10:44:49 UTC; 5 days
ago
    Docs: http://docs.grafana.org
```

You can also check <http://localhost:3000> that is the default address for Grafana Web UI.

Now that you have Grafana on your instance, we have to configure **Prometheus** as a **datasource**.

You can configure your datasource this way :

The screenshot shows the 'Data Sources / Prometheus' configuration page. At the top, there's a red circular icon with a flame symbol. The title 'Data Sources / Prometheus' is displayed, followed by 'Type: Prometheus'. Below the title, there are two tabs: 'Settings' (which is selected) and 'Dashboards'. A search bar contains the text 'Prometheus'. To the right of the search bar are 'Default' and a toggle switch.

HTTP

URL	http://localhost:9090	i
Access	Server (Default)	Help ▾
Whitelisted Cookies	Add Name	i

Auth

Basic Auth	<input type="checkbox"/>	With Credentials	<input type="checkbox"/>
TLS Client Auth	<input type="checkbox"/>	With CA Cert	<input type="checkbox"/>
Skip TLS Verify	<input type="checkbox"/>		
Forward OAuth Identity	i	<input type="checkbox"/>	

Scrape interval: 1s [i](#)

Query timeout: 60s [i](#)

HTTP Method: GET [i](#)

Action Buttons: Save & Test (green), Delete (red), Back

That's it!

Click on 'Save and Test' and make sure that your datasource is working properly.



Building a bash script to retrieve metrics

Your next task is to build a simple bash script that retrieve metrics such as the CPU usage and the memory usage for individual processes.

Your script can be defined as a cron task that will run every second later on.

To perform this task, you have multiple candidates.

You could run top commands every second, parse it using sed and send the metrics to Pushgateway.

The hard part with top is that it runs on multiple iterations, providing a metrics average over time. This is not really what we are looking for.

Instead, we are going to use the `ps` command and more precisely the `ps aux` command.

CPU usage Memory usage

A terminal window showing the output of the `ps aux` command. The output lists various processes with columns for USER, PID, %CPU, %MEM, VSZ, RSS, TTY, STAT, START, TIME, and COMMAND. Annotations with arrows point from the text "CPU usage" to the %CPU column and from "Memory usage" to the %MEM column.

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	225544	6640	?	Ss	Jan05	3:40	/lib/systemd/system --system --deserialize 41
root	2	0.0	0.0	0	0	?	S	Jan05	0:00	[kthreadd]
root	4	0.0	0.0	0	0	?	I<	Jan05	0:00	[kworker/0:0H]
root	6	0.0	0.0	0	0	?	I<	Jan05	0:00	[mm_percpu_wq]
root	7	0.0	0.0	0	0	?	S	Jan05	2:23	[ksoftirqd/0]
root	8	0.0	0.0	0	0	?	I	Jan05	72:10	[rcu_sched]
root	9	0.0	0.0	0	0	?	I	Jan05	0:00	[rcu_bh]
root	10	0.0	0.0	0	0	?	S	Jan05	0:00	[migration/0]

This command exposes individual CPU and memory usages as well as the exact command behind it.

This is exactly what we are looking for.

But before going any further, let's have a look at what **Pushgateway** is expecting as an input.

Pushgateway, pretty much like Prometheus, works with **key value pairs**: the key describes the **metric monitored** and the value is self explanatory.

Here are some examples:

simplest metric

cpu_usage
~~~~~  
key                  14.04  
~~~~~  
value

metric with labels

cpu_usage{process="java", pid="42"} 14.04
~~~~~  
labeled key                  value  
~~~~~

As you can tell, the first form simply describes the CPU usage, but the second one describes the CPU usage for the java process.

Adding labels is a way of specifying what your metric describes more precisely.

Now that we have this information, we can build our final script.

As a reminder, our script will perform a ps aux command, parse the result, transform it and send it to the Pushgateway via the syntax we described before.

Create a script file, give it some rights and navigate to it.

```
> touch better-top  
> chmod u+x better-top  
> vi better-top
```

Here's the script:

If you want the same script for memory usage, simply change the 'cpu_usage' label to 'memory_usage' and the \$3z to \$4z

```
#!/bin/bash  
z=$(ps aux)  
while read -r z  
do  
    var=$var$(awk '{print "cpu_usage{process=\"\"$11\"\", pid=\"\"$2\"\"}",  
$3z}');  
done <<< "$z"  
curl -X POST -H "Content-Type: text/plain" --data "$var"  
" http://localhost:9091/metrics/job/top/instance/machine
```

So what does this script do?

First, it performs the **ps aux** command we described before.

As you can tell, this script gathers all metrics for our processes but it only runs one iteration.

For now, we are simply going to execute it every one second using a sleep command.

Later on, you are free to create a service to execute it every second with a timer (at least with systemd).

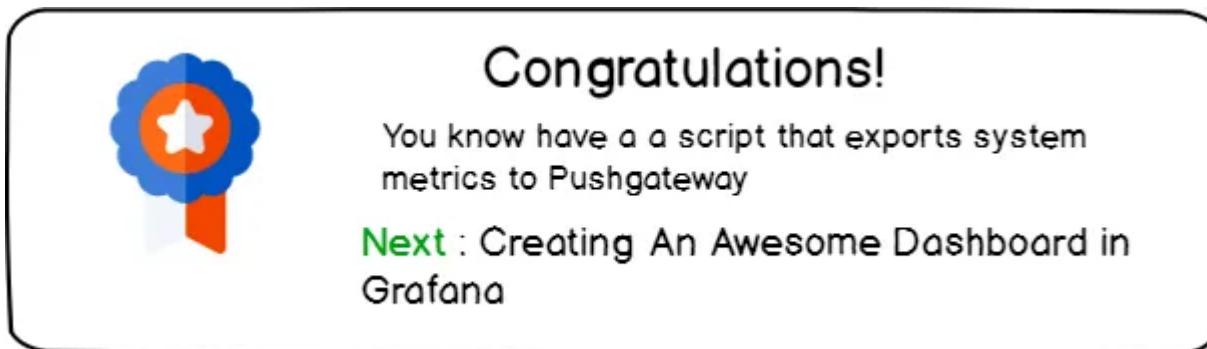
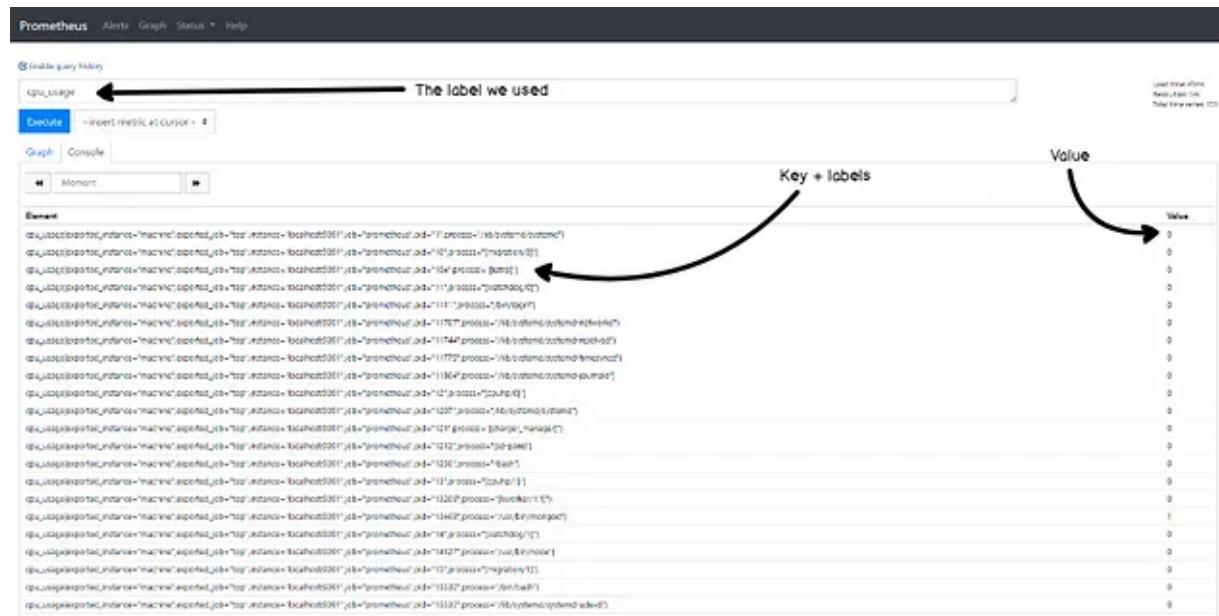
Interested in systemd? [I made a complete tutorial about monitoring them with Chronograf](#)

```
> while sleep 1; do ./better-top; done;
```

Now that our metrics are sent to the Pushgateway, let's see if we can explore them in **Prometheus Web Console**.

Head over to <http://localhost:9090>. In the ‘Expression’ field, simply type ‘cpu_usage’. You should now see all metrics in your browser.

Congratulations! Your CPU metrics are now stored in Prometheus TSDB.

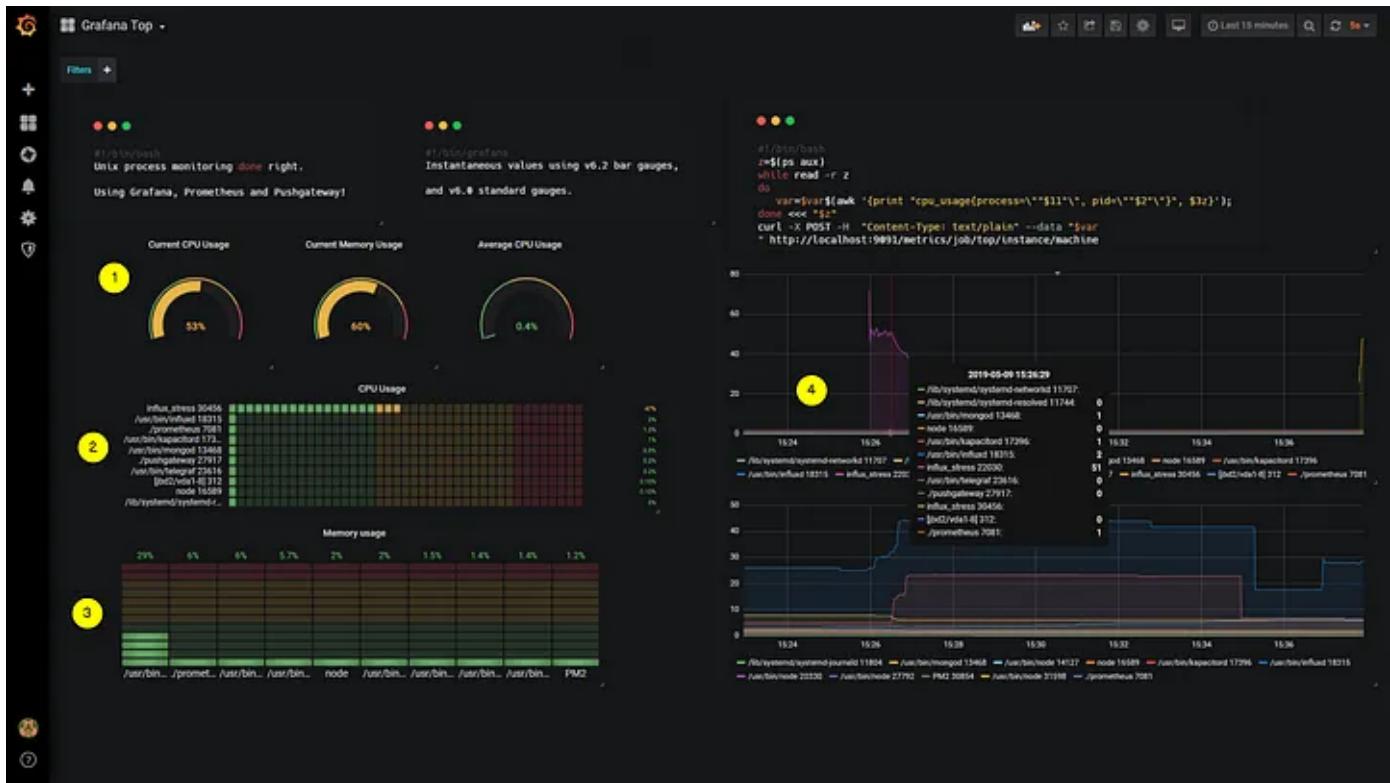


Building An Awesome Dashboard With Grafana

Now that our metrics are stored in Prometheus, we simply have to build a **Grafana** dashboard in order to visualize them.

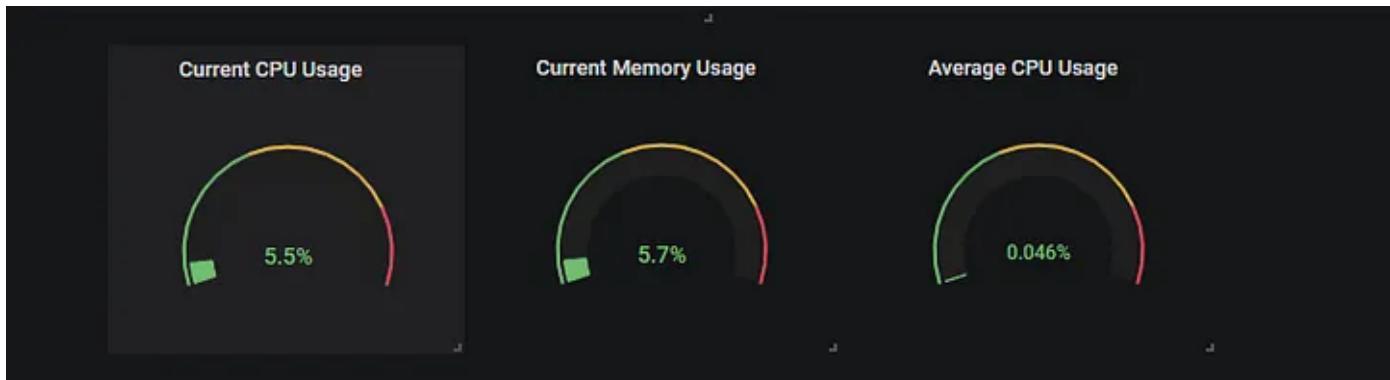
For your comfort, I have annotated the final dashboard with numbers from 1 to 4.

They will match the different subsections of this chapter. If you're only interested in a certain panel, head over directly to the corresponding subsection.



1—Building Rounded Gauges

Here's a closer view of what rounded gauges in our panel.



For now, we are going to focus on the CPU usage of our processes as it can be easily mirrored for memory usage.

With those panels, we are going to track two metrics : the current CPU usage of all our processes and the average CPU usage.

In order to retrieve those metrics, we are going to perform PromQL queries on our Prometheus instance.

So.. what's PromQL?

PromQL is the query language designed for Prometheus.

Similarly what you found find on InfluxDB instances with InfluxQL (or IFQL), PromQL queries can aggregate data using functions such as the sum, the average and the standard deviation.

The syntax is very easy to use as we are going to demonstrate it with our panels.

a — Retrieving the current overall CPU usage

In order to retrieve the current overall CPU usage, we are going to use **PromQL sum function**.

At a given moment in time, our overall CPU usage is simply the sum of individual usages.

Here's the cheat sheet:

💡 Panel needed



💡 PromQL query

```
sum(cpu_usage)
```

💡 Grafana Query Configuration

The screenshot shows the Grafana Query configuration interface. A yellow arrow points from the text "you need instant values" to the "Instant" checkbox, which is highlighted with a red circle. The text "check this!" is also present above the arrow.

💡 Grafana Visualization Configuration

The screenshot shows the Grafana Visualization configuration interface. It includes sections for "Field" (with settings like Title: Auto, Unit: percent (0-100), Min: 0, Max: 100, Decimals: auto) and "Thresholds" (with three color-coded markers: red at 80, orange at 40, and green at Base).

b — Retrieving the average CPU usage

Not much work to do for average CPU usage, you are simply going to use the `avg` function of PromQL. You can find the cheat sheet below.

💡 Panel needed



💡 PromQL query

```
avg(cpu_usage)
```

💡 Grafana Query Configuration

Query: avg(cpu_usage)

Legend: ([process])

Min step: 0

Resolution: 1/1

Format: Time series

Time series: Instant

Min time interval: 0

Relative time: 1h

Time shift: 0h

check this!
you need instant values

💡 Grafana Visualization Configuration

Visualization: Gauge

Show: Calculation

Calc: Mean

Labels:

Markers:

Field

Title:	Auto
Unit:	percent (0-100)
Min:	0
Max:	100
Decimals:	auto

Thresholds

- Base
- 80
- 40

2 — Building Horizontal Gauges

Horizontal gauges are one of the latest additions of Grafana v6.2.

Our goal with this panel is to expose the top 10 most consuming processes of our system.

To do so, we are going to use the **topk** function that retrieves the top k elements for a metric.

Similarly to what we did before, we are going to define thresholds in order to be informed when a process is consuming too many resources.

💡 Panel needed



💡 PromQL query

```
topk(10, cpu_usage)
```

💡 Grafana Query Configuration

💡 Grafana Visualization Configuration

3 — Building Vertical Gauges

Vertical gauges are very similar to horizontal gauges, we only need to tweak the **orientation parameter** in the visualization panel of Grafana.

Also, we are going to monitor our memory usage with this panel so the query is slightly different.

💡 Panel needed



💡 PromQL query

```
topk(10, memory_usage)
```

💡 Grafana Query Configuration

💡 Grafana Visualization Configuration

Awesome! We made great progress so far, one panel to go.

4 — Building Line Graphs

Line graphs have been in Grafana for a long time and this is the panel that we are going to use to have an historical view of how our processes have evolved over time.

This graph can be particularly handy when :

- You had some outage in the past and would like to investigate which processes were active at the time.
- A certain process died but you want to have a view of its behaviour right before it happened

When it comes to troubleshooting exploration, it would honestly need a whole article (especially with the recent Grafana Loki addition).

Okay, here's the final cheat sheet!

💡 Panel needed



💡 PromQL query

```
topk(10, cpu_usage)
```

💡 Grafana Query Configuration

The screenshot shows the Grafana Query configuration screen. At the top, there's a dropdown for 'Query' set to 'Prometheus'. Below it, a query editor contains the PromQL query `topk(10, cpu_usage)`. To the right of the editor, there are several controls: 'Legend' (dropdown), 'Min step' (0), 'Resolution' (1/1), 'Format' (dropdown), 'Time series' (dropdown), and 'Instant' (checkbox). A yellow arrow points from the text 'Not checked in this case you want historical values' to the 'Instant' checkbox. Below these controls are time range sliders for 'Min time interval' (0), 'Relative time' (1h), and 'Time shift' (1h).

💡 Grafana Visualization Configuration

The screenshot shows the Grafana Visualization configuration screen. It has tabs for 'Visualization' and 'Graph' (selected). Under 'Graph', there are sections for 'Draw Modes' (Bars, Lines, Points), 'Mode Options' (Fill, Line Width, Staircase), 'Hover tooltip' (Mode: All series, Sort order: None), and 'Stacking & Null value' (Stack: off, Null value: null). A button at the bottom left says '+ Add series override'.

From there, we have all the panels that we need for our final dashboard.

You can arrange them the way you want or simply take some inspiration from the one we built.



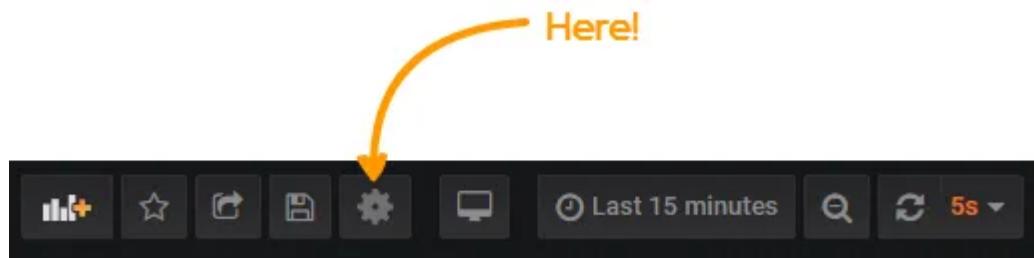
Bonus : explore data using ad hoc filters

Real time data is interesting to see — but the real value comes when you are able to explore your data.

In this bonus section, we are not going to use the ‘Explore’ function (maybe in another article?), we are going to use ad hoc filters.

With Grafana, you can define variables associated to a graph. You have many different options for variables : you can for example define a variable for your data source that would allow to dynamically switch the datasource in a query.

In our case, we are going to use simple ad hoc filters to explore our data.



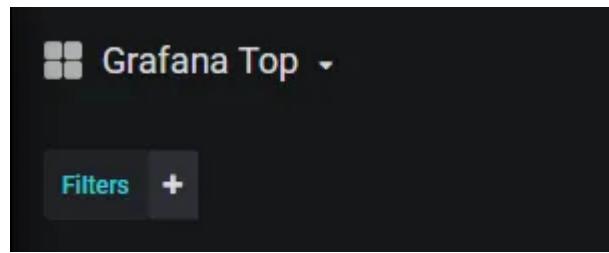
From there, simply click on ‘Variables’ in the left menu, then click on ‘New’.

Variable Configuration

Name	Type
AdhocFilters	Ad hoc filters

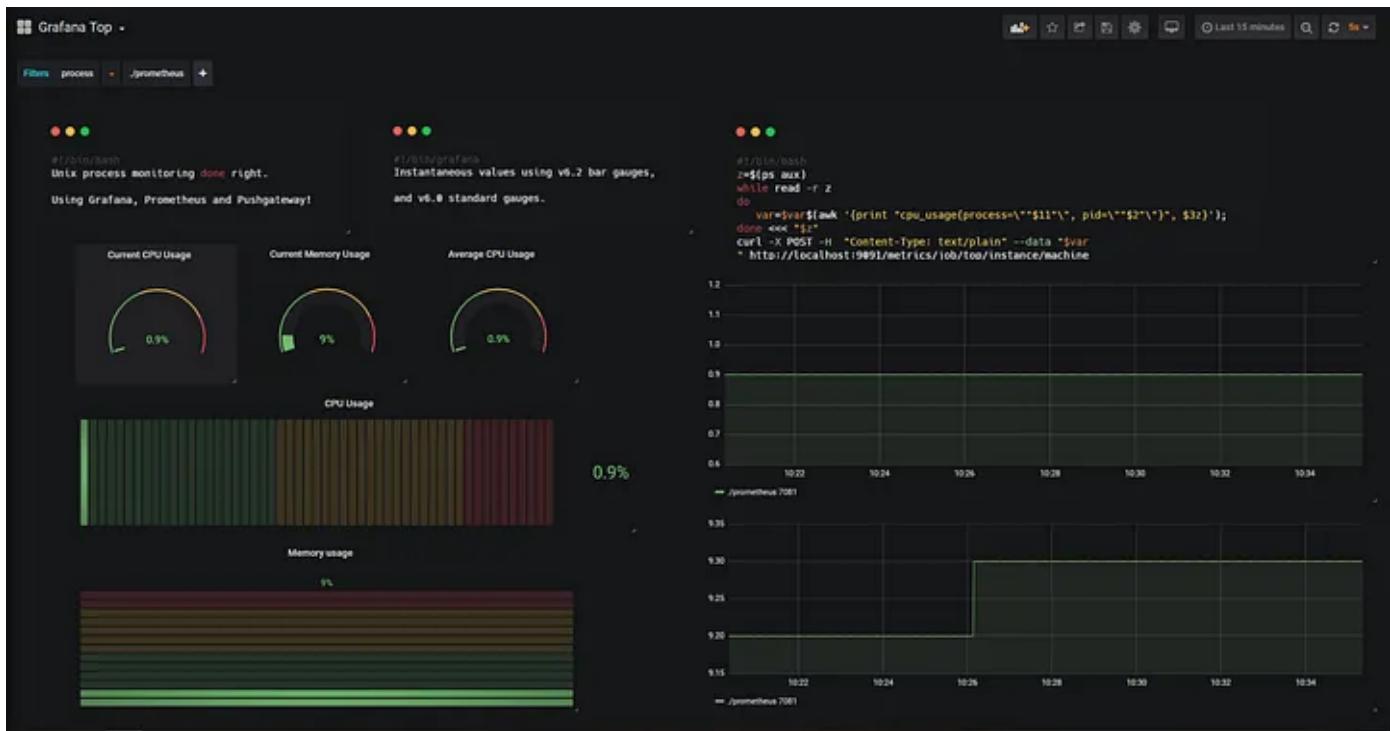
As stated, ad hoc filters are automatically applied to dashboards that target the Prometheus datasource. Back to our dashboard.

Take a look at the top left corner of the dashboard.



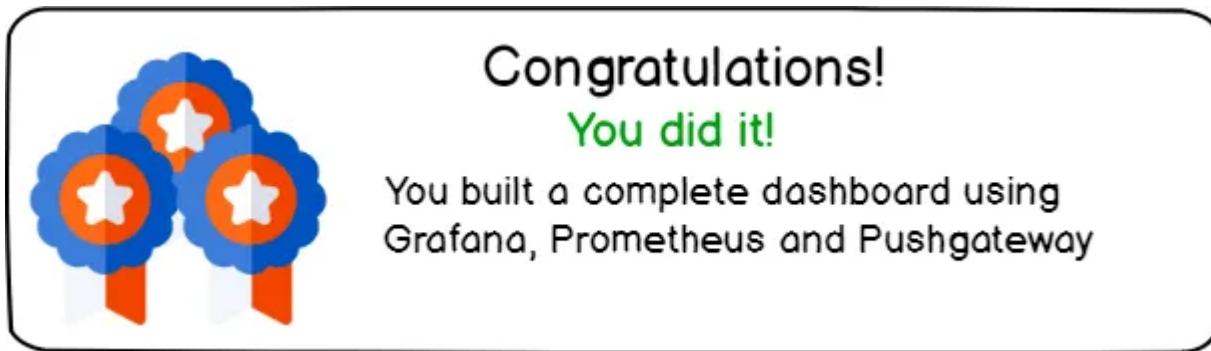
Now let's say that you want the performance of a certain process in your system : let's take Prometheus itself for example.

Simply navigate into the filters and see the dashboard updating accordingly.



Now you have a direct look at how Prometheus is behaving on your instance.

You could even go back in time and see how the process behaved, independently from its pid!



A quick word to conclude

From this tutorial, you now have a better understanding of what **Prometheus** and **Grafana** have to offer.

You know have a **complete monitoring dashboard** for one instance, but there is really a small step to make it scale and monitor an entire cluster of Unix instances.

DevOps monitoring is definitely an interesting subject — but it can turn into a nightmare if you do it wrong.

This is exactly why we write those articles and build those dashboards : to help you reach the maximum efficiency of what those tools have to offer.

We believe that great tech can be enhanced with useful showcases.

I made similar articles, so if you enjoyed this one, make sure to read the others :

Until then, have fun, as always.



Are You a Writer?

Smedian helps you find Medium publications that may want to publish your content.

Find Your Next Publisher

(<https://www.smedian.com/pubs?src=embed-button-smbrowse>)

Powered by Smedian ♥ (<https://www.smedian.com>) — Make Mine
(<https://www.smedian.com/pr/embeds>)

Originally published at <http://devconnected.com> on May 18, 2019.

About Help Terms Privacy

Get the Medium app

