



[The blog](#) from the team @ [Scout](#), Application Monitoring for the long tail of performance.

English ▼

[Start your free trial](#)

Understanding page faults and memory swap-in/outs: when should you worry?

BY [Doug Breaker](#)

September 13, 2019



Updated version of an article first published on April 10th, 2015.

Imagine this: your library is trying to step up its game and compete in the Internet age. Rather than you browsing the shelves, trying to remember how the Dewey Decimal works, you'll enter your book selections from your phone. A librarian will then bring your books to the front desk.

You place your book order on a busy weekend morning. Rather than getting all of your books, the librarian just brings one back. Sometimes the librarian even asks for your book back, tells you to walk out the door to make

26
Shares

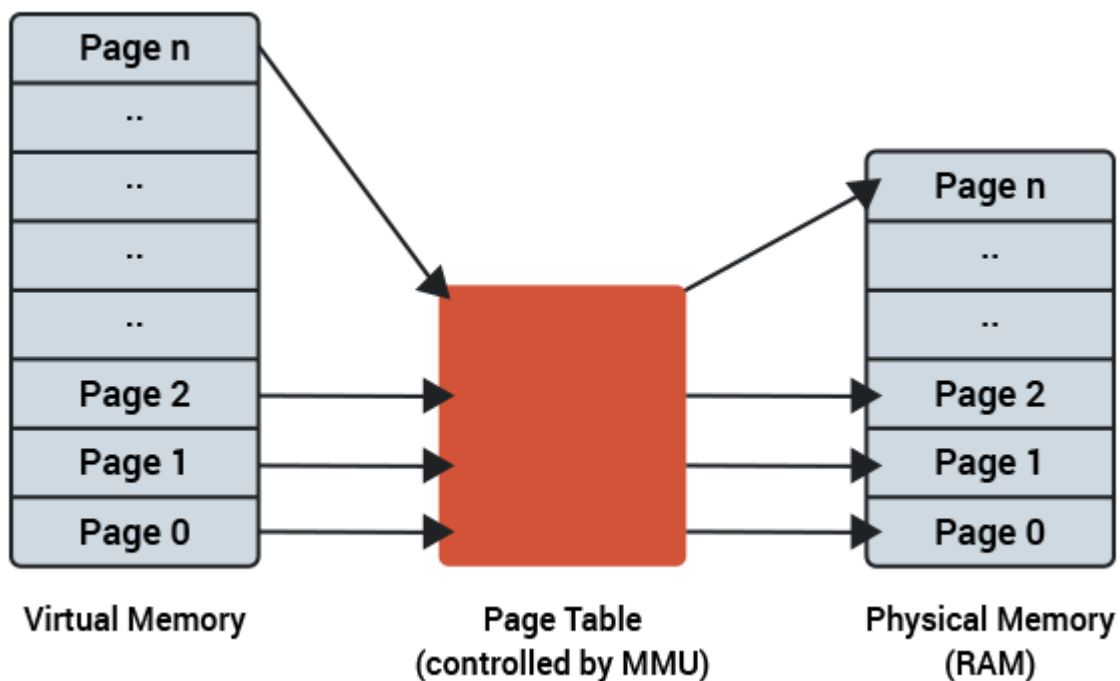
What's going on? Is the librarian insane?

This is the life of the Linux's memory management unit (librarian) and processes (you and the other book readers). A page fault happens when the librarian needs to fetch a book.

How can you tell if page faults are slowing you down, and - above all - how can you avoid being shuffled in-and-out of the library?

More about pages

Linux allocates memory to processes by dividing the physical memory into pages, and then mapping those physical pages to the virtual memory needed by a process. It does this in conjunction with the Memory Management Unit (MMU) in the CPU. Typically a page will represent 4KB of physical memory. Statistics and flags are kept about each page to tell Linux the status of that chunk of memory.



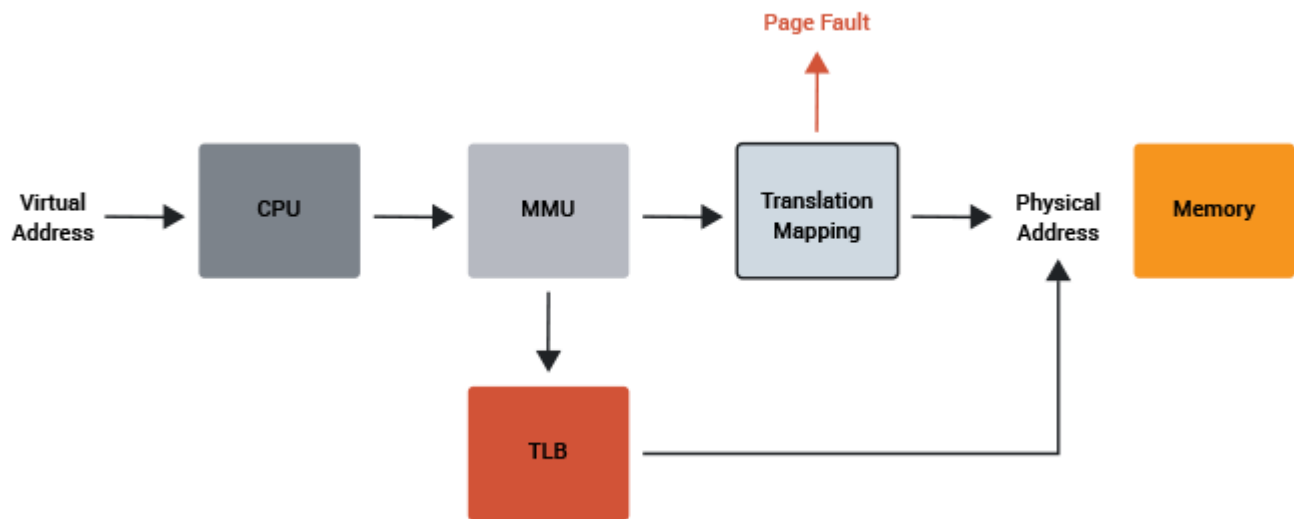
These pages can be in different states. Some will be free (unused), some will be used to hold executable code, and some will be allocated as data for a program. There are lots of clever algorithms that manage this list of pages and control how they are cached, freed and loaded.

What's a page fault? An example.

Imagine a large running program on a Linux system. The program executable size could be measured in megabytes, but not all that code will run at once. Some of the code will only be run during initialization or when a special condition occurs. Over time Linux can discard the pages of memory which hold executable code, if it thinks that they are no longer needed or will be used rarely. As a result not all of the machine code will be held in memory even when the program is running.

A program is executed by the CPU as it steps its way through the machine code. Each instruction is stored in

memory. The MMU knows that **the page for that code isn't available (because Linux told it) and so the CPU will raise a page fault.**



The name sounds more serious than it really is. It isn't an error, but rather a known event where the CPU is telling the operating system that it needs physical access to some more of the code.

Linux will respond by allocating more pages to the process, filling those pages with the code from the binary file, configuring the MMU, and telling the CPU to continue.

A page fault is you requesting the next book in the *Lord of the Rings* Trilogy from the librarian, the librarian retrieving the book from the shelves, and notifying you that the book is now at the front desk.

Minor page faults?

There is also a **special case scenario called a minor page fault which occurs when the code (or data) needed is actually already in memory, but it isn't allocated to that process.** For example, if a user is running a web browser then the memory pages with the browser executable code can be shared across multiple users (since the binary is read-only and can't change). If a second user starts the same web browser then Linux won't load all the binary again from disk, it will map the shareable pages from the first user and give the second process access to them. In other words, a minor page fault occurs only when the page list is updated (and the MMU configured) without actually needing to access the disk.

A minor page fault is your friend requesting to read your checked out copy of *The Two Towers* and you saying "hey, lets just make a copy of mine!" OR you returning a book, but then immediately checking it out again before the book was even returned to a shelf.

Copy on Write?

A similar thing happens for data memory used by a program. An executable can ask Linux for some memory, say 8 megabytes, so that it can perform some task or other. Linux doesn't actually give the process 8 megabytes of physical memory. Instead it allocates 8 megabytes of virtual memory and marks those pages as **"copy on write."** This means that while they are unused there is no need to actually physically allocate them, but the moment the process writes to that page, a real physical page is allocated and the page assigned to the process.

This happens all the time on a multi-user, multitasking system. The physical memory is used in the most efficient way possible to hold the parts of memory that are actually needed for processes to run.

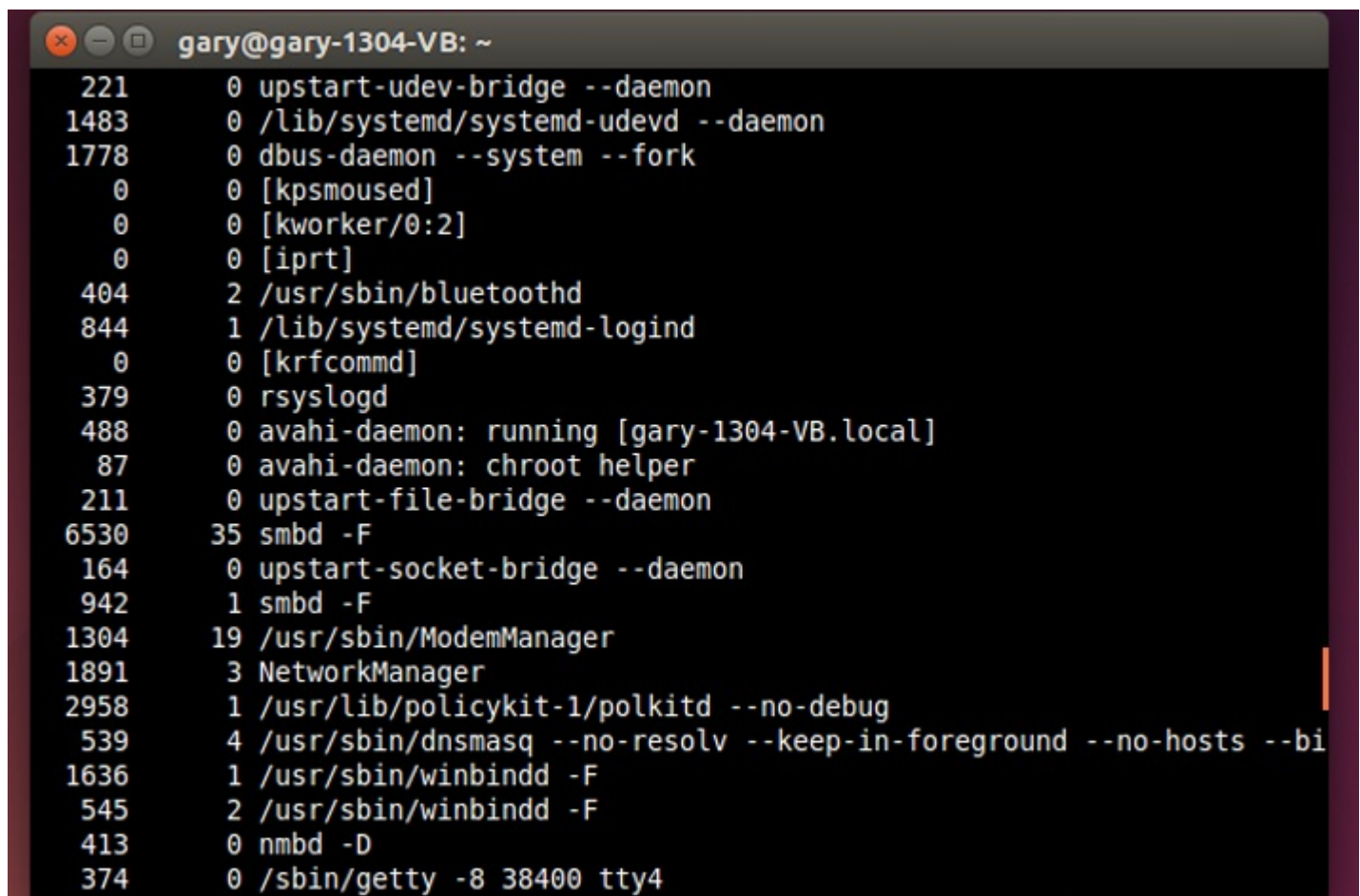
Copy on Write is you telling the librarian you'll be there in 15 minutes and you want your *The Return of the King* book when you get there. The librarian notes where the book is so they can quickly find it when you arrive.

How frequent are page faults?

One of the easiest ways to see the number of major and minor page faults on a Linux system is with the `ps` command. Try the following:

```
ps -eo minflt,majflt,cmd
```

This will list the current running processes on the system along with the number of minor and major page faults that each process has generated.



```
gary@gary-1304-VB: ~
221      0 upstart-udev-bridge --daemon
1483     0 /lib/systemd/systemd-udev --daemon
1778     0 dbus-daemon --system --fork
   0      0 [kpsmoused]
   0      0 [kworker/0:2]
   0      0 [iprt]
  404     2 /usr/sbin/bluetoothd
  844     1 /lib/systemd/systemd-logind
   0      0 [krfcommd]
  379     0 rsyslogd
  488     0 avahi-daemon: running [gary-1304-VB.local]
   87     0 avahi-daemon: chroot helper
  211     0 upstart-file-bridge --daemon
6530    35 smbd -F
  164     0 upstart-socket-bridge --daemon
  942     1 smbd -F
1304    19 /usr/sbin/ModemManager
1891     3 NetworkManager
2958     1 /usr/lib/policykit-1/polkitd --no-debug
  539     4 /usr/sbin/dnsmasq --no-resolv --keep-in-foreground --no-hosts --bi
1636     1 /usr/sbin/winbindd -F
  545     2 /usr/sbin/winbindd -F
  413     0 nmbd -D
  374     0 /sbin/getty -8 38400 tty4
```

The way to see the page faults that are generated by an executable is to use the `/usr/bin/time` command with the `-v` option. Note: It is important to specify `/usr/bin/time` rather than just typing `time` because your shell likely has a `time` command, which although similar won't do exactly the same thing.

Try this:

```
/usr/bin/time -v firefox
```

browser you will likely see a number of major page faults. On my test machine it was around 40. However the minor page faults is quite large, around 30000 on my test setup.

```
Average resident set size (kbytes): 0  
Major (requiring I/O) page faults: 33  
Minor (reclaiming a frame) page faults: 101353  
Voluntary context switches: 9718
```

Now if you run the command again you will see that the number of major faults has dropped to zero, but the minor page faults remains high. This is because on the second go around there were no page faults generated which required the kernel to fetch the executable code from the disk, as they were still somewhere in memory from the first go around. However the number of minor page faults remained the same as the kernel found the pages of memory needed for various shareable libraries etc., and quickly made them available to the process.

Swapping

Under normal operation, the kernel is managing pages of memory so that the virtual address space is mapped onto the physical memory and every process has access to the data and code that it needs. But what happens when the kernel doesn't have any more physical memory left? Assuming that we would like the system to keep running then the kernel has a trick it can use. The kernel will start to write to disk some of the pages which it is holding in memory, and use the newly freed pages to satisfy the current page faults.

Writing pages out to disk is a relatively slow process (compared to the speed of the CPU and the main memory), however it is a better option than just crashing or killing off processes.

The process of writing pages out to disk to free memory is called swapping-out. If later a page fault is raised because the page is on disk, in the swap area rather than in memory, then the kernel will read back in the page from the disk and satisfy the page fault. This is swapping-in.



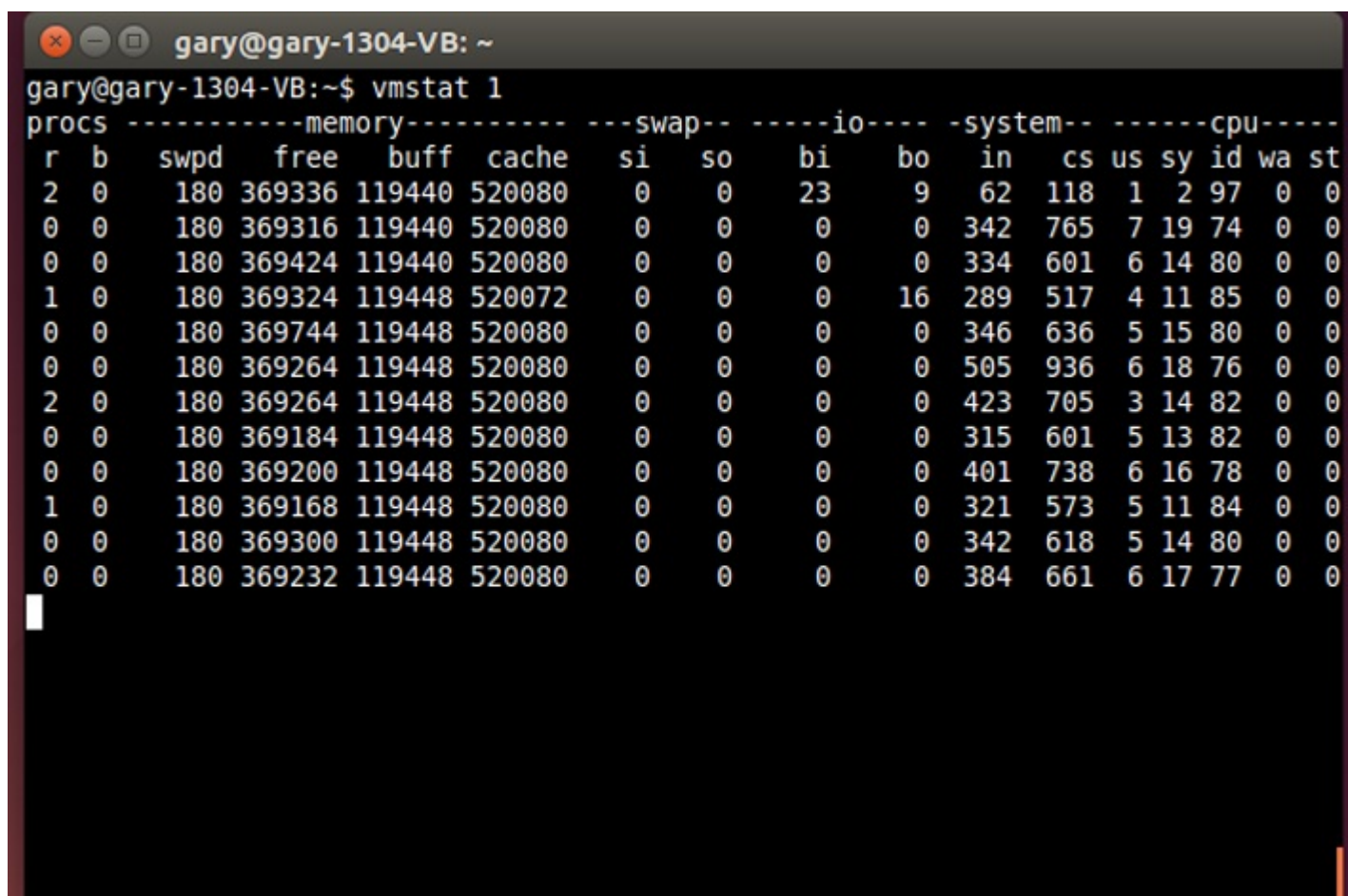
If a system is heavily loaded then an undesirable situation can occur when the latest page fault requires a page to be swapped-in but there still isn't enough free memory. So to satisfy the swap-in the kernel must first swap-out. At this stage there is a danger that the system performance will degrade. If this is only a temporary situation and more free system memory becomes available, then this isn't a problem.

However, there is a worse scenario. Imagine a situation where the kernel must first swap-out some pages in order to free some memory for a swap-in. But then the pages which were just swapped-out are needed again (because of a new page fault) and so must be swapped-in again. To satisfy this swap-in the previous pages that were just swapped-in are now swapped-out. And so on. This is known as thrashing. When a computer system starts thrashing it spends more time trying to satisfy major page faults than it does in actually running processes.

You can use the top command to see how much swap space is being used on your system and the vmstat command to see the current numbers of swap-in si and swap-out so operations.

Try:

vmstat 1



```

gary@gary-1304-VB: ~$ vmstat 1
procs -----memory----- ---swap-- -----io----- -system-- -----cpu-----
 r  b   swpd   free   buff   cache    si   so    bi    bo    in   cs  us  sy  id  wa  st
 2  0     180 369336 119440 520080     0    0    23    9    62  118   1   2  97   0   0
 0  0     180 369316 119440 520080     0    0     0    0   342  765   7  19  74   0   0
 0  0     180 369424 119440 520080     0    0     0    0   334  601   6  14  80   0   0
 1  0     180 369324 119448 520072     0    0     0   16  289  517   4  11  85   0   0
 0  0     180 369744 119448 520080     0    0     0    0   346  636   5  15  80   0   0
 0  0     180 369264 119448 520080     0    0     0    0   505  936   6  18  76   0   0
 2  0     180 369264 119448 520080     0    0     0    0   423  705   3  14  82   0   0
 0  0     180 369184 119448 520080     0    0     0    0   315  601   5  13  82   0   0
 0  0     180 369200 119448 520080     0    0     0    0   401  738   6  16  78   0   0
 1  0     180 369168 119448 520080     0    0     0    0   321  573   5  11  84   0   0
 0  0     180 369300 119448 520080     0    0     0    0   342  618   5  14  80   0   0
 0  0     180 369232 119448 520080     0    0     0    0   384  661   6  17  77   0   0

```

Swapping is you requesting a lot of books - too many to hold at the front desk. The librarian needs to keep the rest in a storage room in the basement, and it takes a long time to go back-and-forth.

When should you worry about page faults and swapping?

Most of the time, **your primary performance worry is a high rate of swap-in/out's**. This means your host doesn't have physical memory to store the needed pages and is using the disk often, which is significantly slower than physical memory.

What metrics should you monitor?

- Swap Activity (swap-ins and swap outs)
- Amount of swap space used

Swap activity is the major performance factor with memory access; simply using a moderate amount of swap space isn't necessarily an issue if the pages swapped out belong to a mostly idle process. However when you begin to use a large amount of swap space there is a greater chance of swap activity impacting your server performance.

One more thing

The kernel's aggressiveness in preemptively swapping-out pages is governed by a kernel parameter called swappiness. It can be set to a number from 0 to 100, where 0 means that more is kept in memory and 100 means that the kernel should try and swap-out as many pages as possible. The default value is 60. Kernel maintainer Andrew Morton has stated that he uses a swappiness of 100 on his desktop machines, "my point is that decreasing the tendency of the kernel to swap stuff out is wrong. You really don't want hundreds of megabytes of BloatyApp's untouched memory floating about in the machine. Get it out on the disk, use the memory for something useful."

TL;DR

- The total amount of virtual address space for all the running processes far exceeds the amount of physical memory.
- The mapping between the virtual address space and physical memory is handled by the Linux kernel and by the CPU's MMU using pages of memory.
- When the CPU needs to access a page that isn't in memory it raises a page fault.
- A major page fault is one that can only be satisfied by accessing the disk.
- A minor page fault can be satisfied by sharing pages that are already in memory.
- Swapping occurs when pages are written to the disk to free memory so that a major page fault can be satisfied.
- Swap activity is the primary performance concern when it comes to page faults.

More servers? Or faster code?

Adding servers can be a band-aid for slow code. [Scout APM](#) helps you find and fix your inefficient and costly code. We automatically identify N+1 SQL calls, memory bloat, and other code-related issues so you can spend less time debugging and more time programming.

Ready to optimize your site? [Sign up for a free trial.](#)

Also see

- [Restricting process CPU usage using nice, cpulimit, and cgroups](#)
- [Slow Server? This is the Flow Chart You're Looking For](#)
- [Understanding CPU Steal Time - when should you be worried?](#)
- [Understanding Linux CPU Load - when should you be worried?](#)

New PHP Laravel Agent.

Interested in trying our new PHP monitoring agent on your Laravel apps? Then click here to get started.

Get started

Recent Posts

- [Scout Now Partnering With API Management Leader DreamFactory.](#)
- [Scout APM for PHP exits Beta](#)

26

Shares

- [Partnering with Render and Manifold](#)

Get notified of new posts.

Once a month, we'll deliver a finely-curated selection of optimization tips to your inbox.



- [Company](#)
- [Blog](#)
- [Careers](#)

Products

- [Ruby Application Monitoring](#)
- [Elixir Application Monitoring](#)
- [Python Application Monitoring](#)
- [PHP Monitoring Beta](#)
- [DevTrace](#)

Support

- [Docs](#)
- [Status Page](#)
- support@scoutapm.com
- [Slack](#)

Legal

- [Terms of Service](#)
- [Privacy Policy](#)

