Analyse production performance with Stack-driver Profiler

Sanjam singh[1]

[1]Student of Information and Technology Engineering, Chandigarh University, Gharuan, Punjab, India

ssanjam20@gmail.com[1]

Abstract

Google Stackdriver could be a freemium, mastercard needed, cloud computing systems management service offered by Google. It provides performance and medicine information to public cloud users. Stackdriver could be a hybrid cloud answer, providing support for each Google Cloud and AWS cloud environments. This analysis paper Analyses the assembly within the performance with the assistance of Stack-driver Profiler and check out to boost the code's performance, equivalent techniques that haven't been nearly as accessible or well adopted by people who work on backend services. whereas shopper app and frontend internet developers usually use tools just like the automaton Studio central processing unit Profiler or the identification tools enclosed in Chrome. Stackdriver Profiler brings these same capabilities to service developers, notwithstanding whether or not their code is running on the Google Cloud Platform.

Keywords: Stack-driver Profiler, Google Cloud Platform, cloud infrastructure, Profiler interface, environment setup.

I. Introduction

Understanding the performance of production systems is notoriously tough. making an attempt to live performance in check environments typically fails to duplicate the pressures on a production system. small benchmarking components of your application is typically possible, however it additionally usually fails to duplicate the employment and performance of a production system. Continuous identification of production systems is an efficient thanks to discover wherever resources like electronic equipment cycles and memory ar consumed as a service operates in its operating atmosphere. however identification adds an extra load on the assembly system: so as to be a suitable thanks to discover patterns of resource consumption, the extra load of identification should be tiny. Stackdriver Profiler may be a applied math, low-overhead profiler that endlessly gathers electronic equipment usage and memory-allocation data from your production applications. It attributes that data to the ASCII text file that generated it, serving to you determine the components of your application that ar overwhelming the foremost resources, and otherwise illuminating your applications performance characteristics.

Stackdriver the corporate was created in 2012 by founders Dan Belcher and Izzy ethnic group. The company's goal was to supply consistent watching across cloud computing's multiple service layers, employing a single SaaS resolution. Stackdriver secured $5 million funding from Bain Capital Ventures in Gregorian calendar month 2012. A beta version of the merchandise became publically offered on April thirty, 2013. In might 2014, the Stackdriver company was nonheritable by Google. Associate in Nursing dilated version of the merchandise was rebranded as Google Stackdriver and was launched to general handiness in Oct, 2016.

Example of Stackdriver: Site24x7 offers a centralized approach to observe your virtual infrastructure, on-premises IT atmosphere, and cloud infrastructure–all on one platform:

1. Monitor usage and performance for infrastructure and PaaS running in your Amazon internet Service atmosphere.
2. Get a comprehesive read into the health of your EC2 server instances.
3. Monitor resource utilization and operational health of 100+ Azure services.

4. Analyze virtual machine workloads and host resource utilization on your vSphere atmosphere.
5. Check standing of your jack containers.

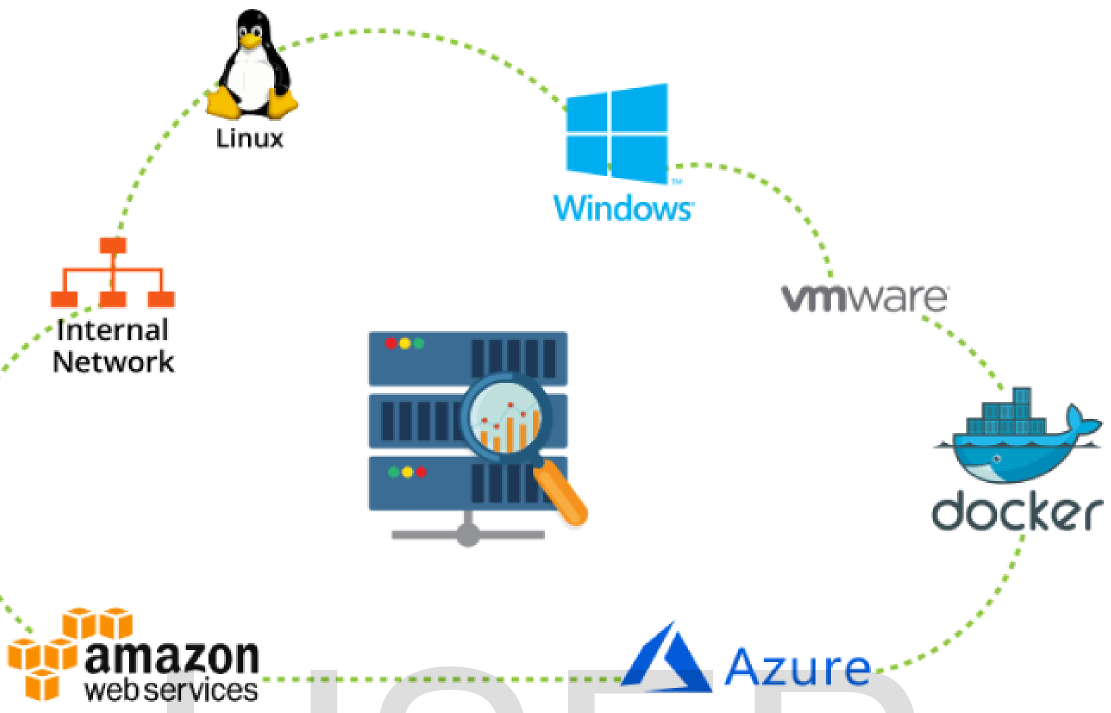
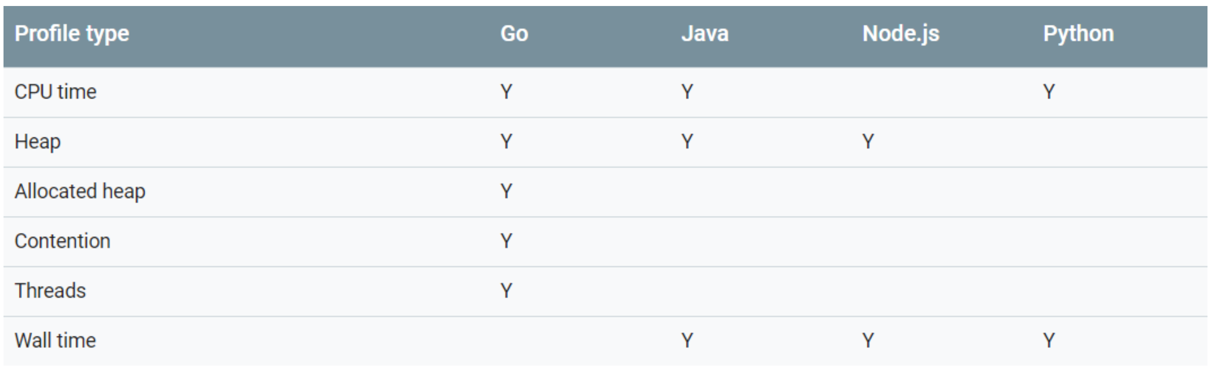


Figure1, Example of Site24x7

II. Supporting Stackdriver

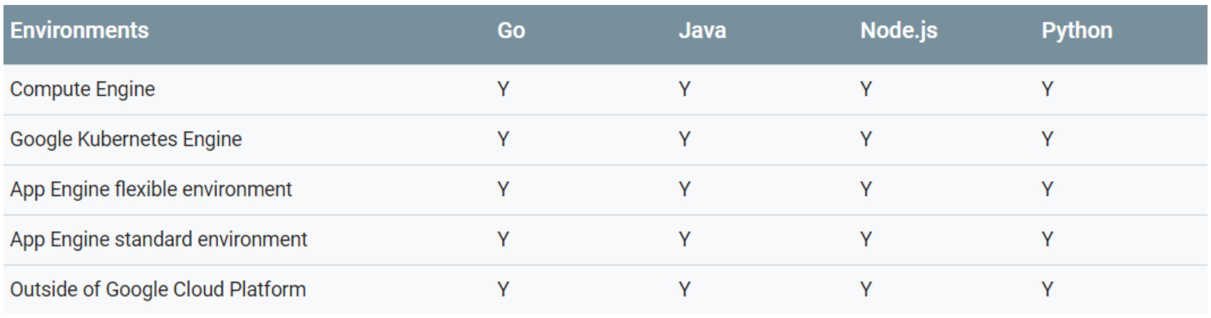
Stackdriver Profiler supports differing types of identification supported the language during which a program is written. the subsequent table summarizes the supported profile varieties by language:



Profile type	Go	Java	Node.js	Python
CPU time	Y	Y		Y
Heap	Y	Y	Y	
Allocated heap	Y			
Contention	Y			
Threads	Y			
Wall time		Y	Y	Y

Table1, supporting profile type by different languages

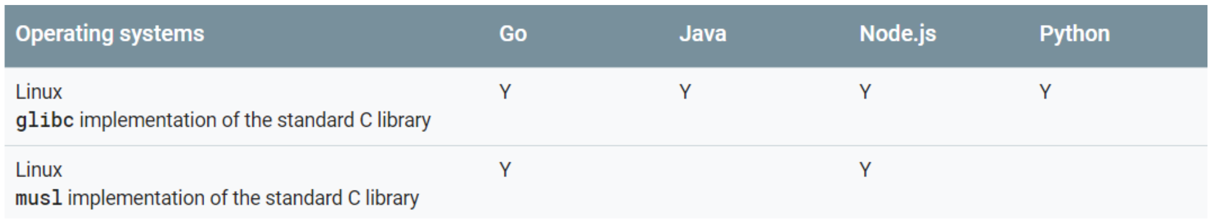
When you instrument your application to capture profile knowledge, you embrace a language-specific identification agent. the subsequent table summarizes the supported environments:



Environments	Go	Java	Node.js	Python
Compute Engine	Y	Y	Y	Y
Google Kubernetes Engine	Y	Y	Y	Y
App Engine flexible environment	Y	Y	Y	Y
App Engine standard environment	Y	Y	Y	Y
Outside of Google Cloud Platform	Y	Y	Y	Y

Table2, supporting environments by different languages

The following table summarizes the supported **operative** systems:



Operating systems	Go	Java	Node.js	Python
Linux `glibc` implementation of the standard C library	Y	Y	Y	Y
Linux `musl` implementation of the standard C library	Y		Y	

Table3, supporting operating systems by different languages

III. Google Stackdriver's main features

1) Stackdriver observation measures the health of cloud resources and applications by providing visibility into metrics like central processor usage, disk I/O, memory, network traffic and period of time. it's supported collectd, associate open supply daemon that collects system and application performance metrics. Users will receive customizable alerts once Stackdriver observation discovers performance problems. it's wont to monitor Google reason Engine and Amazon EC2 VMs.

2) Stackdriver Error reportage identifies and analyzes cloud application errors. A centralized error management interface provides IT groups with time period visibility into production errors with cloud applications, moreover because the ability to kind and filter content supported the quantity of error occurrences, once the error was initial and last seen, and wherever the error is found.

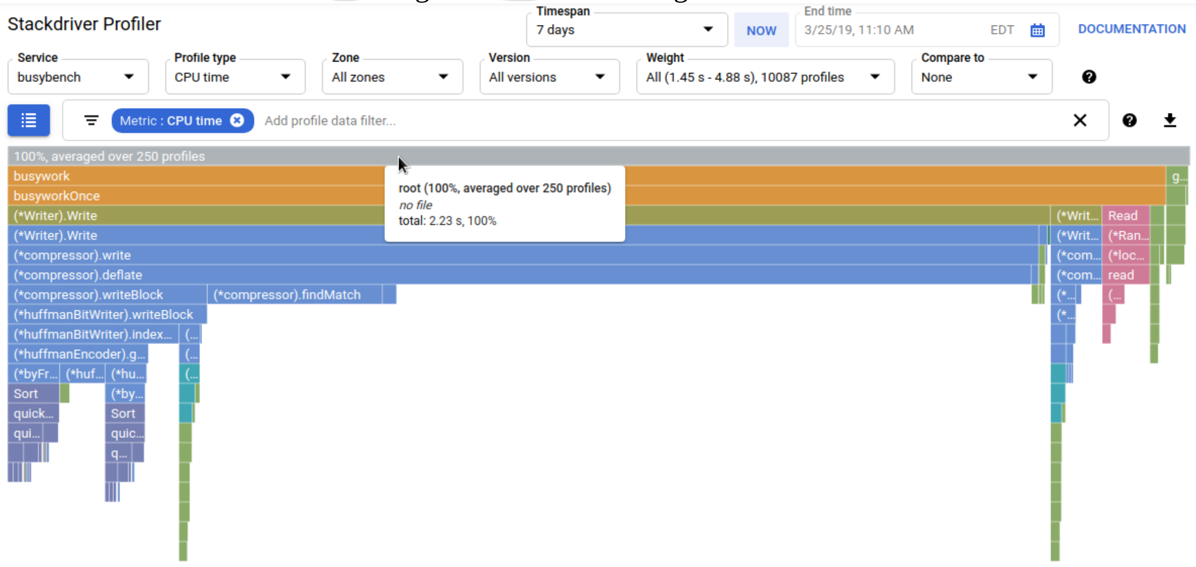
3) Stackdriver computer programme inspects the state of associate application, deployed in Google App Engine or Google reason Engine, exploitation production information and ASCII text file. throughout production, snapshots may be taken of associate application's state and connected back to a selected line location within the ASCII text file, while not having to feature work statements. This examination will occur while not moving the performance of the assembly application.

4) Stackdriver Trace collects network latency information from applications deployed in Google App Engine. Trace information is gathered, analyzed and wont to produce performance reports to spot network bottlenecks. Trace API and Trace SDK may be wont to trace, analyze and optimize custom workloads, as well.

5) Stackdriver work provides time period log management and analysis for cloud applications. Log information may be unbroken for extended periods of your time by archiving it with Google Cloud Storage. The service works with each Google and AWS, and may gather logs from Google reason Engine, Google App Engine and Amazon EC2**.**

IV. Profiler interface

Client app and frontend internet developers normally use tools just like the mechanical man Studio mainframe Profiler or the identification tools enclosed in Chrome to enhance their code's performance, equivalent techniques haven't been nearly as accessible or well adopted by people who work on backend services. Stackdriver Profiler brings these same capabilities to service developers, despite whether or not their code is running on the Google Cloud Platform or elsewhere.



Stackdriver Profiler

Timespan: 7 days | NOW | End time: 3/25/19, 11:10 AM EDT | DOCUMENTATION

Service: busybench | Profile type: CPU time | Zone: All zones | Version: All versions | Weight: All (1.45 s - 4.88 s), 10087 profiles | Compare to: None

Metric : CPU time | Add profile data filter...

root (100%, averaged over 250 profiles)
no file
total: 2.23 s, 100%

Figure2, Profiler interface

The tool gathers hardware usage and memory-allocation data from your production applications. It attributes that data to the application's computer code document, serving to you identify the weather of the appliance overwhelming the foremost resources, and otherwise illuminating the performance characteristics of the code. Low overhead of the gathering techniques utilised by the tool makes it applicable for continuous use in production environments. throughout this codev research laboratory, we'll resolve the way to line up Stackdriver Profiler for a Go program and may get at home with what moderately insights regarding application

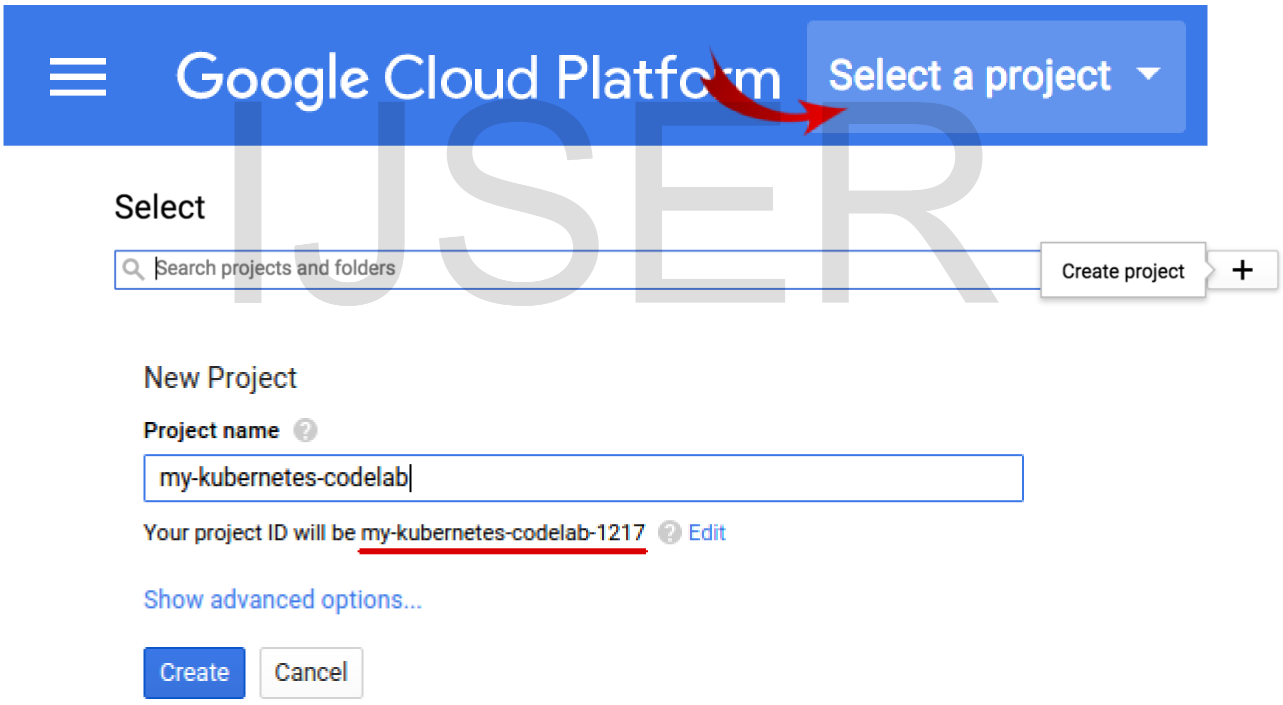
4.1.What we learn in Stackdriver Profiler

(1) How to configure a Go program for profiling with Stackdriver Profiler.
(2) How to collect, view and analyze the performance data with Stackdriver Profiler.

4.2.What we need

i) A Google Cloud Platform project.
ii) A browser, such Chrome or Firefox.
iii) Familiarity with standard Linux text editors such as Vim, EMACs or Nano.
iv) Setup and Requirements.
v) Self-paced environment setup.

If you don't already have a Google Account (Gmail or Google Apps), we must create one. Sign-in to Google Cloud Platform console (console.cloud.google.com) and create a new project:



my-kubernetes-codelab

Your project ID will be my-kubernetes-codelab-1217

Figure3, how to create Google Cloud projects

4.3.Steps of using Google Cloud projects

i) Remember the project ID, a unique name across all Google Cloud projects (the name above has already been taken and will not work for you, sorry!). It will be referred to later in this codelab as PROJECT_ID.
ii) Next, you'll need to enable billing in the Cloud Console in order to use Google Cloud resources.
iii) Running through this codelab shouldn't cost you more than a few dollars, but it could be more if you decide to use more resources or if you leave them running (see "cleanup" section at the end of this document).

iv) New users of Google Cloud Platform are eligible for a $300 free trial.
v) Google Cloud Shell
vi) While Google Cloud can be operated remotely from your laptop, to make the setup simpler in this codelab we will be using Google Cloud Shell, a command line environment running in the Cloud.
vii) Activate Google Cloud Shell

From the GCP Console click the Cloud Shell icon on the top right toolbar:

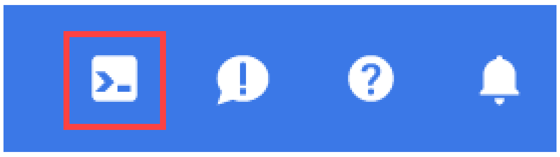
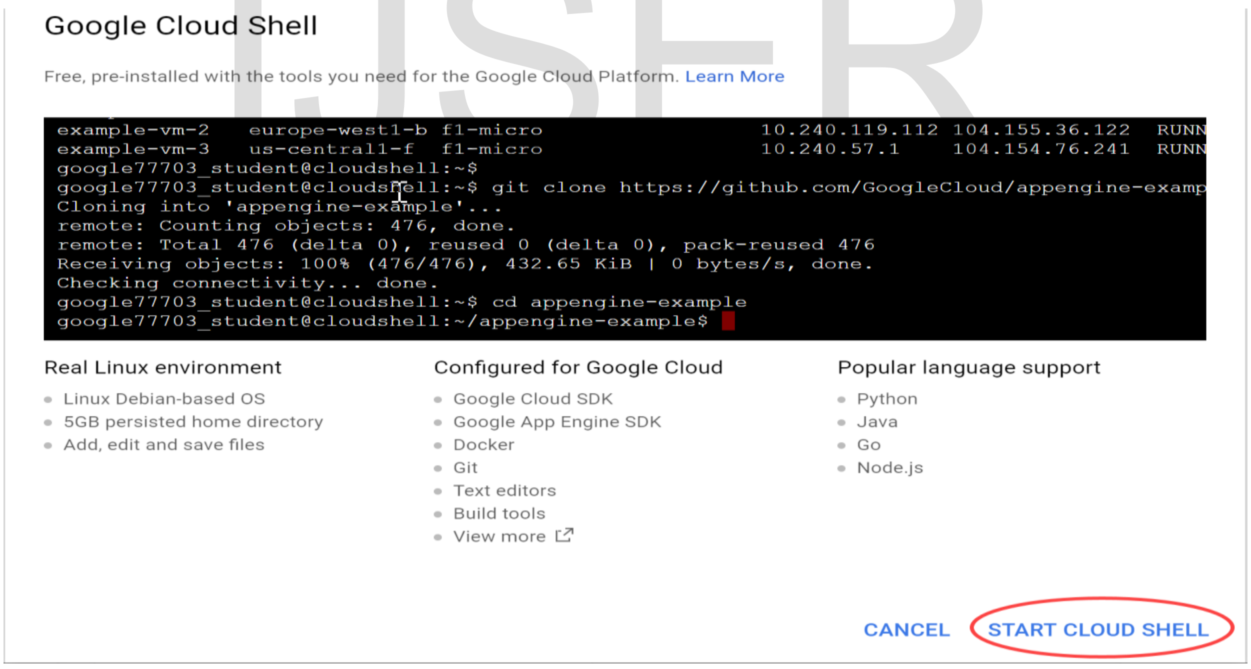


Figure4, click "Start Cloud Shell"

It should only take a few moments to provision and connect to the environment:

i) This virtual machine is loaded with all the development tools you'll need. It offers a persistent 5GB home directory, and runs on the Google Cloud, greatly enhancing network performance and authentication. Much, if not all, of your work in this lab can be done with simply a browser or your Google Chromebook.
ii) Once connected to the cloud shell, you should see that you are already authenticated and that the project is already set to your *PROJECT_ID*.



Google Cloud Shell

Free, pre-installed with the tools you need for the Google Cloud Platform. Learn More

```
example-vm-2   europe-west1-b f1-micro              10.240.119.112 104.155.36.122  RUNN
example-vm-3   us-central1-f  f1-micro              10.240.57.1    104.154.76.241  RUNN
google77703_student@cloudshell:~$
google77703_student@cloudshell:~$ git clone https://github.com/GoogleCloud/appengine-examp
Cloning into 'appengine-example'...
remote: Counting objects: 476, done.
remote: Total 476 (delta 0), reused 0 (delta 0), pack-reused 476
Receiving objects: 100% (476/476), 432.65 KiB | 0 bytes/s, done.
Checking connectivity... done.
google77703_student@cloudshell:~$ cd appengine-example
google77703_student@cloudshell:~/appengine-example$
```

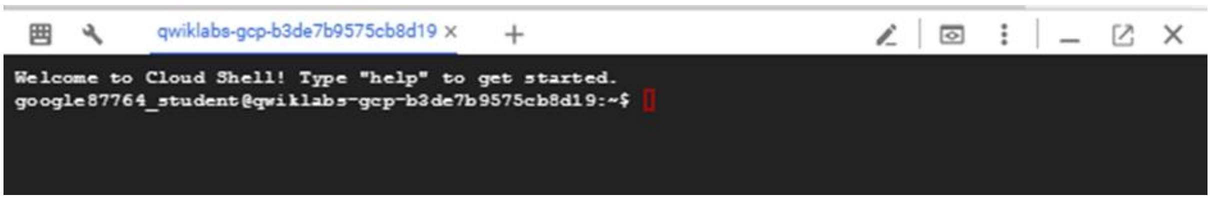
Real Linux environment
- Linux Debian-based OS
- 5GB persisted home directory
- Add, edit and save files

Configured for Google Cloud
- Google Cloud SDK
- Google App Engine SDK
- Docker
- Git
- Text editors
- Build tools
- View more

Popular language support
- Python
- Java
- Go
- Node.js

Figure5, how to start

iii) Run the following command in the cloud shell to confirm that you are authenticated:
(1) gcloud auth list
(2) Command output
(3) Credentialed accounts
(4) - <myaccount>@<mydomain>.com (active)



```
Welcome to Cloud Shell! Type "help" to get started.
google87764_student@qwiklabs-gcp-b3de7b9575cb8d19:~$
```

Figure6, run of start cloud shell

(5) gcloud config list project
(6) Command output
(7) [core]
(8) project = <PROJECT_ID>

If it is not, you can set it with this command:

(9) gcloud config set project <PROJECT_ID>
(10) Command output
(11) Updated property [core/project].

V. Navigate to Stackdriver Profiler

In Cloud Console, navigate to the Profiler UI by clicking on "Profiler" in the left navigation bar:

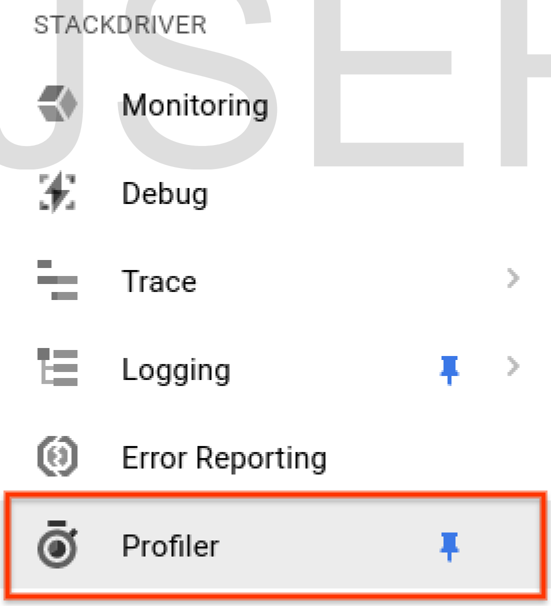
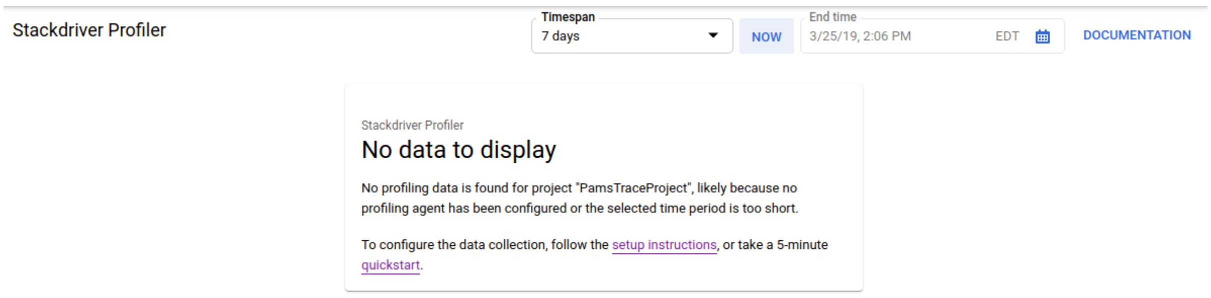


Figure7, Navigate to Stackdriver Profiler

Alternatively you'll use the Cloud Console search bar to navigate to the Profiler UI: simply sort "Stackdriver Profiler" and choose the found item. Either way, you must see the Profiler UI with the "No information to display" message like below. The project is new, thus it does not have any identification information collected nonetheless.


Stackdriver Profiler
Timespan
7 days
NOW
End time
3/25/19, 2:06 PM EDT
DOCUMENTATION

Stackdriver Profiler
No data to display

No profiling data is found for project "PamsTraceProject", likely because no profiling agent has been configured or the selected time period is too short.

To configure the data collection, follow the setup instructions, or take a 5-minute quickstart.

figure8,when no data to display

It's now time to get something profiled!

The directory contains the "main.go" file, which is a synthetic app that has the profiling agent enabled:

Code:

main.go

```
...
import (
    ...
    "cloud.google.com/go/profiler"
)
...
func main() {
    err := profiler.Start(profiler.Config{
        Service:      "hotapp-service",
        DebugLogging:   true,
        MutexProfiling: true,
    })
    if err != nil {
        log.Fatalf("failed to start the profiler: %v", err)
    }
    ...
}
```

The profiling agent collects CPU, heap and thread profiles by default. The code here enables the collection of mutex (also known as "contention") profiles.

Now, run the program:

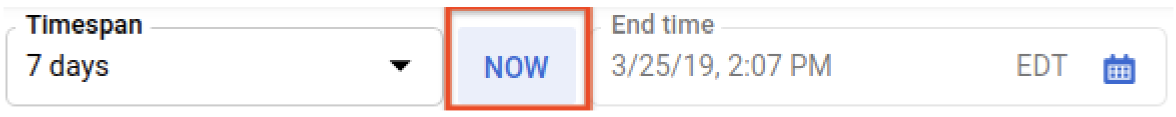
$ go run main.go

As the program runs, the identification agent can sporadically collect profiles of the 5 designed sorts. the gathering is irregular over time (with average rate of 1 profile per minute for every of the types), therefore it's going to take up to a few minutes to induce every of the kinds collected. The program tells you once it creates a profile. The messages square measure enabled by the DebugLogging flag within the configuration above; otherwise, the agent runs silently:

```
$ go run main.go
2018/03/28 15:10:24 profiler has started
2018/03/28 15:10:57 successfully created profile THREADS
2018/03/28 15:10:57 start uploading profile
2018/03/28 15:11:19 successfully created profile CONTENTION
2018/03/28 15:11:30 start uploading profile
2018/03/28 15:11:40 successfully created profile CPU
2018/03/28 15:11:51 start uploading profile
2018/03/28 15:11:53 successfully created profile CONTENTION
2018/03/28 15:12:03 start uploading profile
2018/03/28 15:12:04 successfully created profile HEAP
2018/03/28 15:12:04 start uploading profile
2018/03/28 15:12:04 successfully created profile THREADS
2018/03/28 15:12:04 start uploading profile
2018/03/28 15:12:25 successfully created profile HEAP
2018/03/28 15:12:25 start uploading profile
2018/03/28 15:12:37 successfully created profile CPU
...
```

Figure9, DebugLogging

The UI can update itself shortly once initial of the profiles is collected. It will not auto-update then, thus to check the new information, you will need to refresh the Profiler UI manually. To do that, click the currently button within the interval picker twice:



Timespan: 7 days
End time: 3/25/19, 2:07 PM EDT

Figure10, click "now"

After the UI refreshes, you will see something like this:

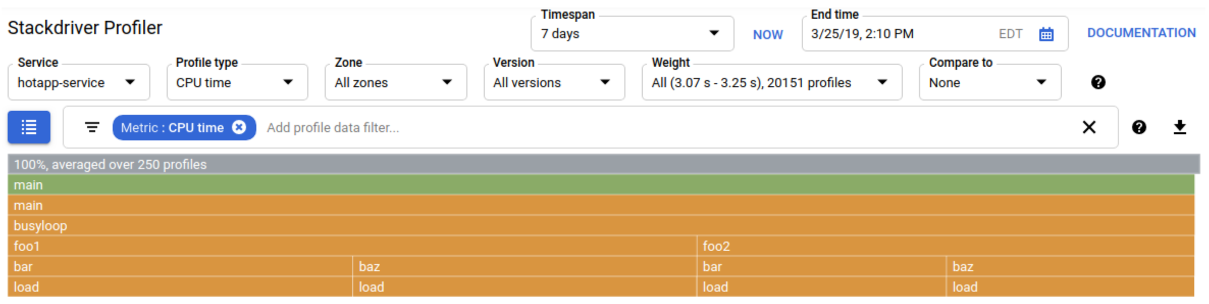


Figure11, stackdriver profile

The profile type selector shows the five profile types available:

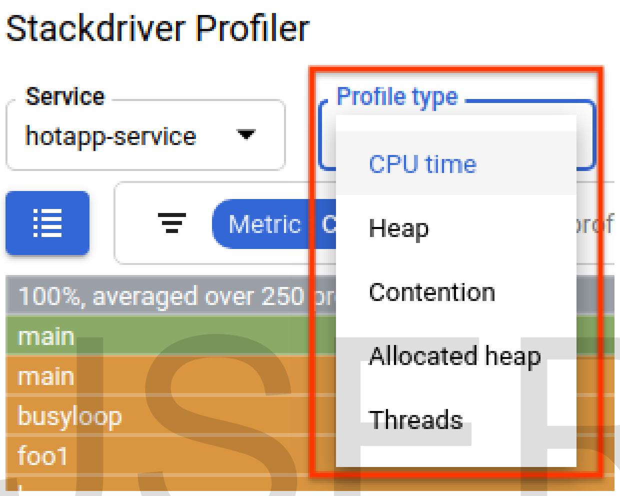


Figure12, the five profile types

Let's now review each of the profile types and some important UI capabilities, and then conduct some experiments. At this stage, you no longer need the Cloud Shell terminal, so you can exit it by pressing CTRL-C and typing "exit".

VI. Analyse the Profiler Data

a. CPU-intensive Code

Select the CPU profile type. After the UI loads it, you'll see in the flame graph the four leaf blocks for load function, which collectively account for all the CPU consumption:

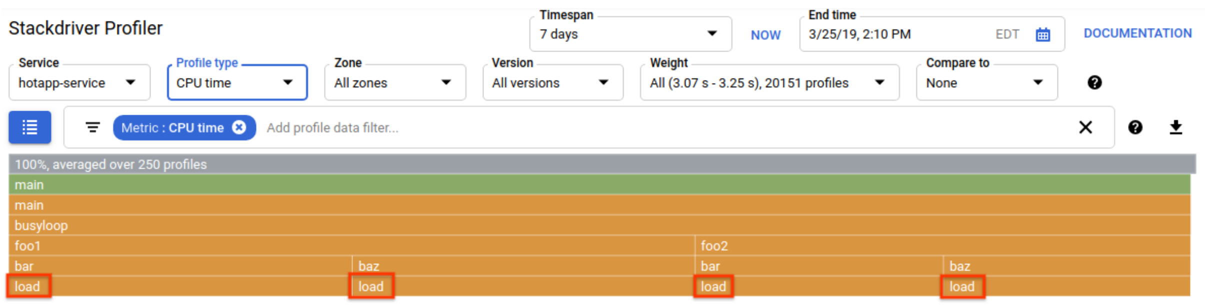


Figure12, CPU-intensive Code the five profile types

This function is specifically written to consume a lot of CPU cycles by running a tight loop:

main.go

```
func load() {
    for i := 0; i < (1 << 20); i++ {
    }
}
```

The function is called indirectly from busyloop() via four call paths: busyloop → {foo1, foo2} → {bar, baz} → load. The width of a function box represents the relative cost of the specific call path. In this case all four paths have about the same cost. In a real program, you want to focus on optimizing call paths that matter the most in terms of performance. The flame graph, which visually emphasizes the more expensive paths with larger boxes, makes these paths easy to identify.

You can use the profile data filter to further refine the display. For example, try adding a "Show stacks" filter specifying "baz" as the filter string. You should see something like the screenshot below, where only two of the four call paths to load() are displayed. These two paths are the only ones that go through a function with the string "baz" in its name. Such filtering is useful when you want to focus on a subpart of a bigger program (for example, because you only own part of it).

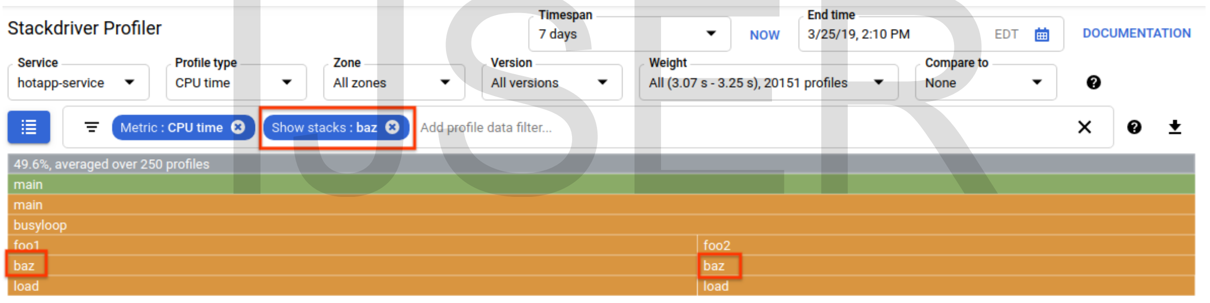
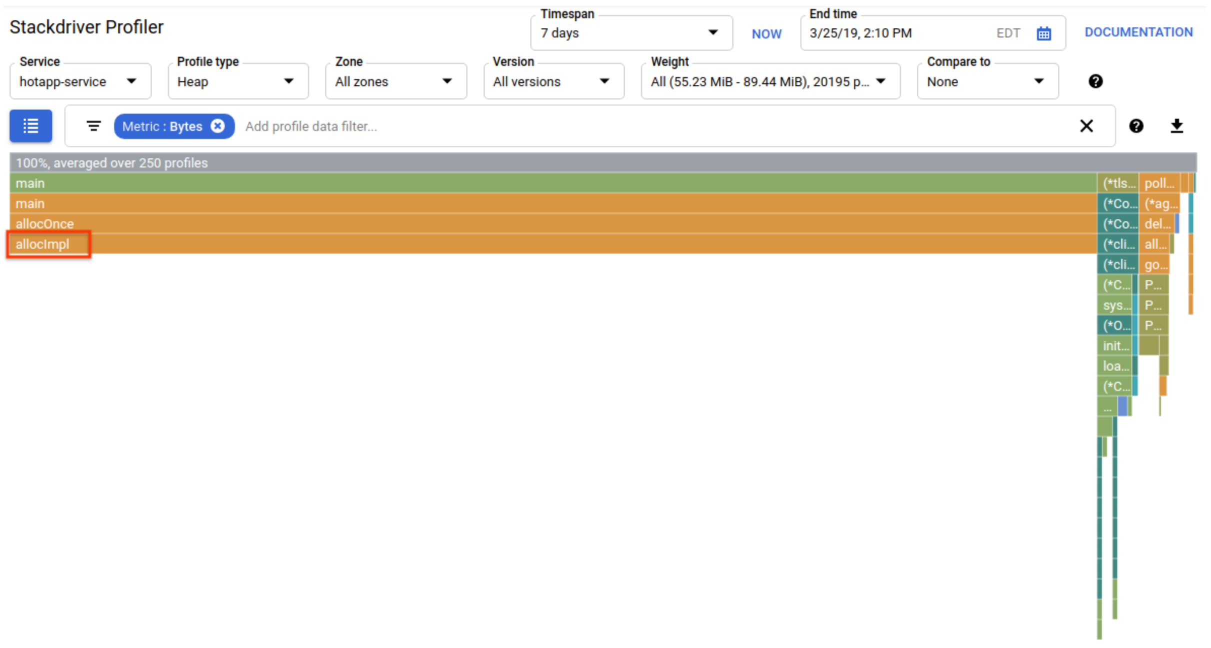

Stackdriver Profiler
Timespan: 7 days
End time: 3/25/19, 2:10 PM EDT
Service: hotapp-service
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: All (3.07 s - 3.25 s), 20151 profiles
Compare to: None
Metric : CPU time
Show stacks : baz
49.6%, averaged over 250 profiles

Figure13, click "baz"

b. Memory-intensive Code

Now switch to "Heap" profile type. Make sure to remove any filters you created in previous experiments. You should now see a flame graph where allocImpl, called by alloc, is displayed as the main consumer of memory in the app:


Stackdriver Profiler

Timespan: 7 days | NOW | End time: 3/25/19, 2:10 PM EDT | DOCUMENTATION

Service: hotapp-service | Profile type: Heap | Zone: All zones | Version: All versions | Weight: All (55.23 MiB - 89.44 MiB), 20195 p... | Compare to: None

Metric : Bytes | Add profile data filter...

100%, averaged over 250 profiles

Figure14, allocated allocImpl function

The summary table above the flame graph indicates that the total amount of used memory in the app is on average ~57.4 MiB, most of it is allocated by allocImpl function. This is not surprising, given the implementation of this function:

main.go

```
func allocImpl() {

    // Allocate 64 MiB in 64 KiB chunks

    for i := 0; i < 64*16; i++ {

        mem = append(mem, make([]byte, 64*1024))

    }

}
```

The function executes once, allocating 64 MiB in smaller chunks, then storing pointers to those chunks in a global variable to protect them from being garbage collected. Note that the amount of memory shown as used by profiler is slightly different from 64 MiB: the Go heap profiler is a statistical tool, so the measurements are low-overhead but not byte-accurate. Don't be surprised when seeing a ~10% difference like this.

c. IO-intensive Code

If you choose "Threads" in the profile type selector, the display will switch to a flame graph where most of the width is taken by wait and waitImpl functions:

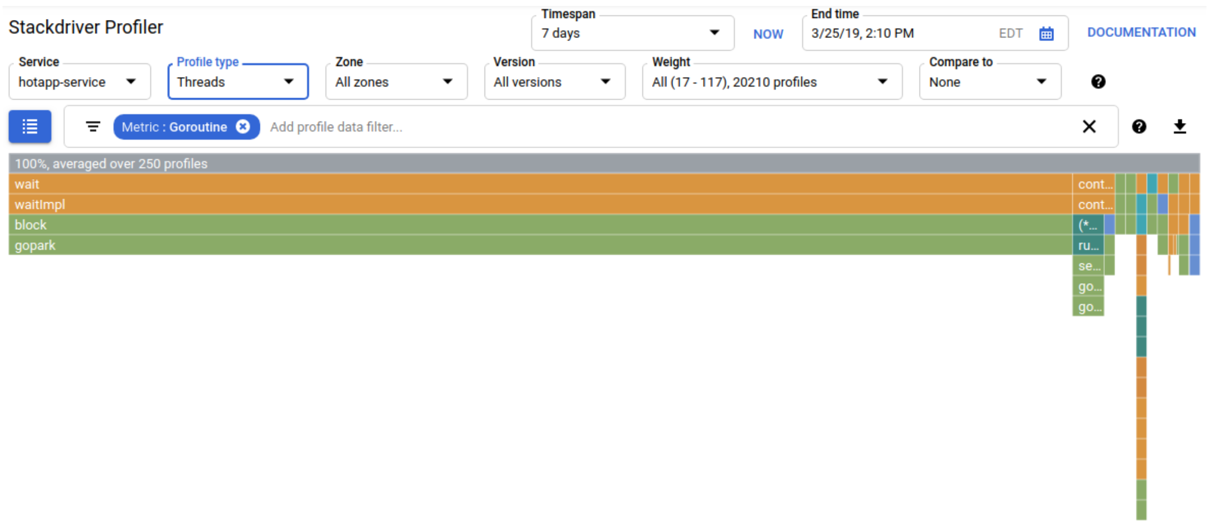


Figure15, call stack from the wait function

In the summary above flame graph, you can see that there are 100 goroutines that grow their call stack from the wait function. This is exactly right, given that the code that initiates these waits looks like this:

main.go

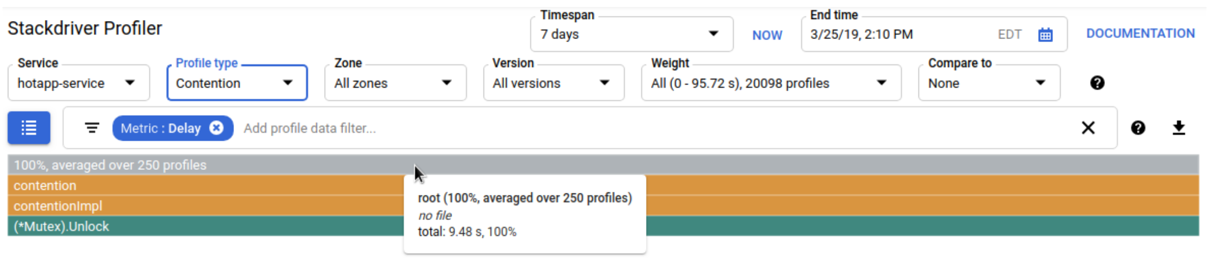
```
func main() {

        ...

        // Simulate some waiting goroutines.

        for i := 0; i < 100; i++ {

                go wait()

        }
```

This profile type is useful for understanding whether the program spends any unexpected time in waits (like I/O). Such call stacks would not be typically sampled by the CPU profiler, as they don't consume any significant portion of the CPU time. You'll often want to use "Hide stacks" filters with Threads profiles - for example, to hide all stacks ending with a call to gopark, since those are often idle goroutines and less interesting than ones that wait on I/O.

The threads profile type can also help identify points in program where threads are waiting for a mutex owned by another part of the program for a long period, but the following profile type is more useful for that.

d. **Contention-intensive Code**

The Contention profile type identifies the most "wanted" locks in the program. This profile type is available for Go programs but must be explicitly enabled by specifying "MutexProfiling: true" in the agent configuration code. The collection works by recording (under the "Contentions" metric) the number of times when a specific lock, when being unlocked by a goroutine A, had another goroutine B waiting for the lock to be unlocked. It also records (under the "Delay" metric) the time the blocked goroutine waited for the lock. In this example, there is a single contention stack and the total wait time for the lock was 11.03 seconds:


Stackdriver Profiler

Timespan: 7 days | NOW | End time: 3/25/19, 2:10 PM EDT | DOCUMENTATION

Service: hotapp-service | Profile type: Contention | Zone: All zones | Version: All versions | Weight: All (0 - 95.72 s), 20098 profiles | Compare to: None

Metric : Delay | Add profile data filter...

100%, averaged over 250 profiles
contention
contentionImpl
(*Mutex).Unlock

root (100%, averaged over 250 profiles)
no file
total: 9.48 s, 100%

figure16, single contention stack and the total wait time

The code that generates this profile consists of 4 goroutines fighting over a mutex:

main.go

```
func contention(d time.Duration) {

    contentionImpl(d)

}


func contentionImpl(d time.Duration) {

    for {

        mu.Lock()

        time.Sleep(d)

        mu.Unlock()

    }

}

...

func main() {

    ...

    for i := 0; i < 4; i++ {

        go contention(time.Duration(i) * 50 * time.Millisecond)

    }

}
```

VII Conclusion

Stackdriver aggregates metrics, logs, and events from infrastructure, giving developers and operators a fashionable set of discernible signals that speed root-cause analysis and scale back unit of time to resolution (MTTR). Stackdriver doesn’t need intensive integration or multiple “panes of glass,” and it won’t lock developers into employing a specific cloud supplier. we have a tendency to learned however a Go program are often designed to be used with Stackdriver Profiler. we have a tendency to additionally learned a way to collect, read and analyze the performance information with this tool. we are able to currently apply your new ability to the $64000 services you run on Google Cloud Platform. Native integration with Google Cloud information tools BigQuery, Cloud Pub/Sub, Cloud Storage, Cloud Datalab, and out-of-the-box integration with all of your different application parts. Stackdriver is made from the bottom up for cloud-powered applications. whether or not you’re running on Google Cloud Platform, Amazon net Services, on-premises infrastructure, or with hybrid clouds, Stackdriver combines metrics, logs, and data from all of your cloud accounts and comes into one comprehensive read of your setting, therefore you'll quickly perceive service behavior and take action.

References

[1] https://cloud.google.com/profiler/docs/about-profiler

[2] https://medium.com/@duhroach/application-performance-management-with-stackdriver-844300899d52

[3] https://cloud.google.com/blog/products/gcp/google-stackdriver-generally-available

[4] https://www.bizjournals.com/boston/blog/startups/2012/09/stackdriver-bain-capital-ventures.html

[5] https://www.site24x7.com/cloud-monitoring.html?ad_src=google_asia_africa_search&ad_grp=cloud_monitoring&gclid=Cj0KCQjwrMHsBRCIARIsAFgSeI1kSwgUkPvN0kwjcFJcU9kzmemRBPPTY1sN9Qo_oapp0-HJK-FtNsQaAmmmEALw_wcB

[6] https://en.wikipedia.org/wiki/Stackdriver

[7] https://searchcloudcomputing.techtarget.com/definition/Google-Stackdriver

[8] https://cloud.google.com/stackdriver/