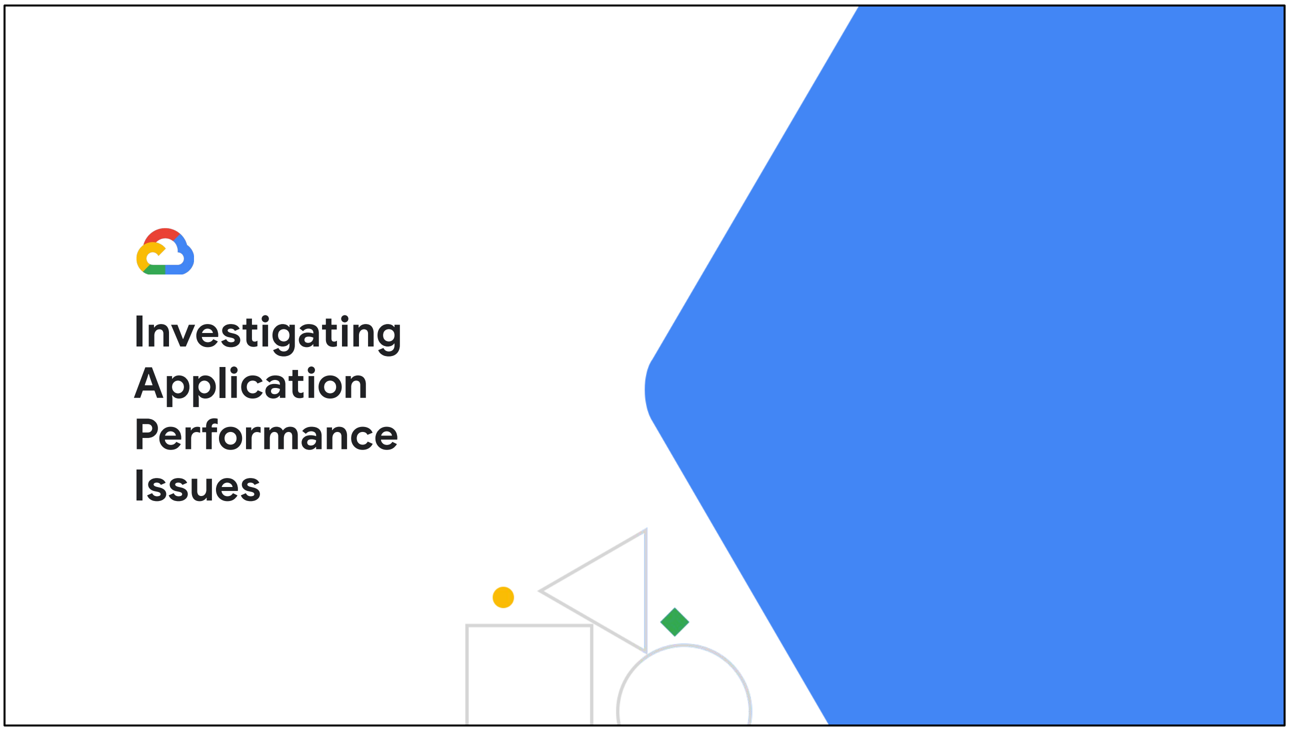


Investigating Application Performance Issues

When deploying applications to Google Cloud, the Application Performance Management products (Cloud Trace and Cloud Profiler) provide a suite of tools to give insight into how your code and services are functioning, and to help troubleshoot where needed.

Agenda

Trace

Profiler

In this module, you will learn to:

- Trace latency through layers of service interaction to eliminate performance bottlenecks.
- Profile and identify resource-intensive functions in an application.

Agenda

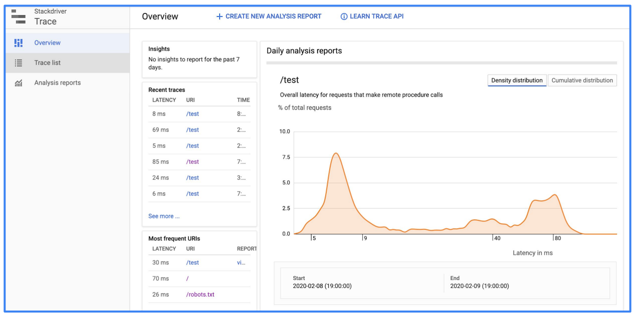
Trace

Profiler

Now, let’s have a look at Trace.

Cloud Trace Tracks Application Latency

- Track request latencies as they propagate
- Detailed, near real-time performance insights
- In-depth reports
- Support for several mainstream languages



Insights

No insights to report for the past 7 days.

Recent traces

LATENCY	URI	TIME
8 ms	/test	8:...
69 ms	/test	2:...
5 ms	/test	2:...
85 ms	/test	7:...
24 ms	/test	3:...
6 ms	/test	7:...

See more ...

Most frequent URIs

LATENCY	URI	REPORT
30 ms	/test	vi...
70 ms	/	
26 ms	/robots.txt	

Daily analysis reports

/test

Overall latency for requests that make remote procedure calls

Start 2020-02-08 (19:00:00)

End 2020-02-09 (19:00:00)

Cloud Trace is a distributed tracing system that collects latency data from your applications and displays it in the Google Cloud Console.

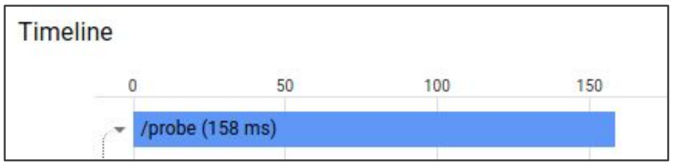
You can track how requests propagate through your application and receive detailed near real-time performance insights.

Cloud Trace automatically analyzes all of your application's traces to generate in-depth latency reports to surface performance degradations, and can capture traces from all of your VMs, containers, or App Engine projects.

Cloud Trace agent supports a specific set of mainstream programming languages, but you can also trace using OpenCensus and/or OpenTelemetry instrumentation.

Trace Terminology

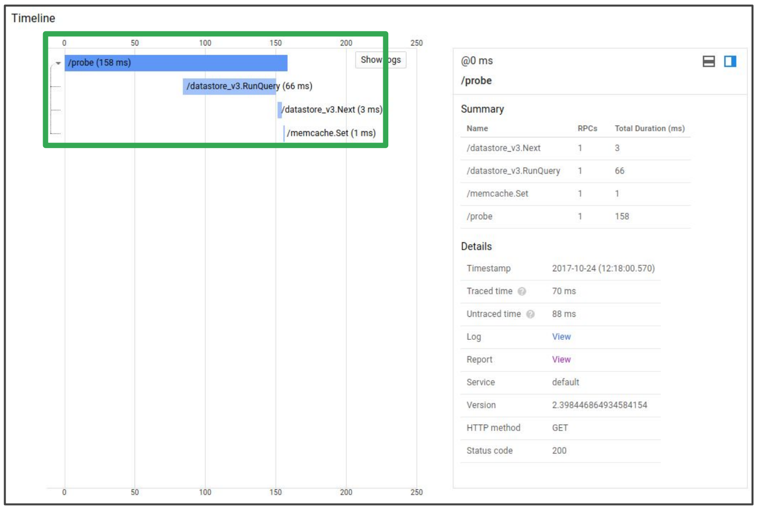
- Each Trace is a collection of Spans
- A Span wraps Metrics about an application unit of work
  - A context, timing, and other metrics



A trace is a collection of spans.

A Span is an object that wraps metrics and other contextual information about a unit of work in your application.

Viewing Trace Detail



@0 ms

/probe

Summary

Name	RPCs	Total Duration (ms)
/datastore_v3.Next	1	3
/datastore_v3.RunQuery	1	66
/memcache.Set	1	1
/probe	1	158

Details

Timestamp	2017-10-24 (12:18:00.570)
Traced time	70 ms
Untraced time	88 ms
Log	View
Report	View
Service	default
Version	2.398446864934584154
HTTP method	GET
Status code	200

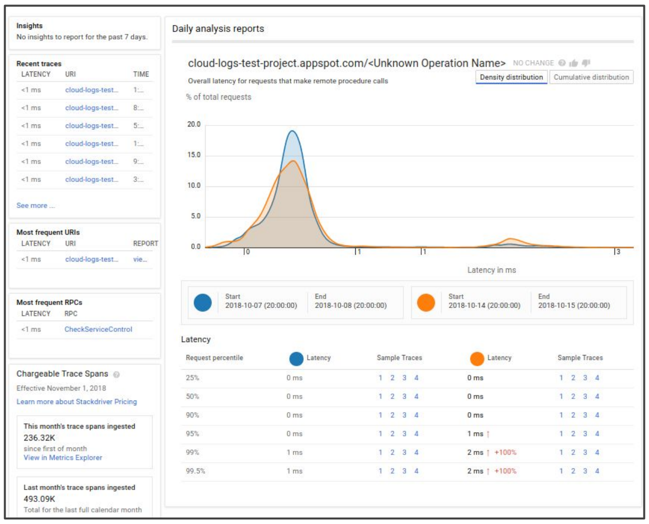
Here, we see an example trace created by a call to */probe*. It took the */probe* span a total of 158ms to handle the request and return a response.

But */probe* didn't do all the work itself. */probe* called a method to RunQuery in Datastore, which took 66ms. Then */probe* took that result and called next on it, which took 3ms.

Finally, */probe* called set, which took 1ms.

So, the above four spans all worked to create the single trace we are examining.

Trace overview



Insights
No insights to report for the past 7 days.

Recent traces

LATENCY	URI	TIME
<1 ms	cloud-logs-test...	1:...
<1 ms	cloud-logs-test...	8:...
<1 ms	cloud-logs-test...	5:...
<1 ms	cloud-logs-test...	1:...
<1 ms	cloud-logs-test...	9:...
<1 ms	cloud-logs-test...	3:...

See more ...

Most frequent URIs

LATENCY	URI	REPORT
<1 ms	cloud-logs-test...	vie...

Most frequent RPCs

LATENCY	RPC
<1 ms	CheckServiceControl

Chargeable Trace Spans
Effective November 1, 2018
Learn more about Stackdriver Pricing

This month's trace spans ingested
236.32K
since first of month
View in Metrics Explorer

Last month's trace spans ingested
493.09K
Total for the last full calendar month

Daily analysis reports

cloud-logs-test-project.appspot.com/<Unknown Operation Name>
Overall latency for requests that make remote procedure calls

	Start	End
●	2018-10-07 (20:00:00)	2018-10-08 (20:00:00)
●	2018-10-14 (20:00:00)	2018-10-15 (20:00:00)

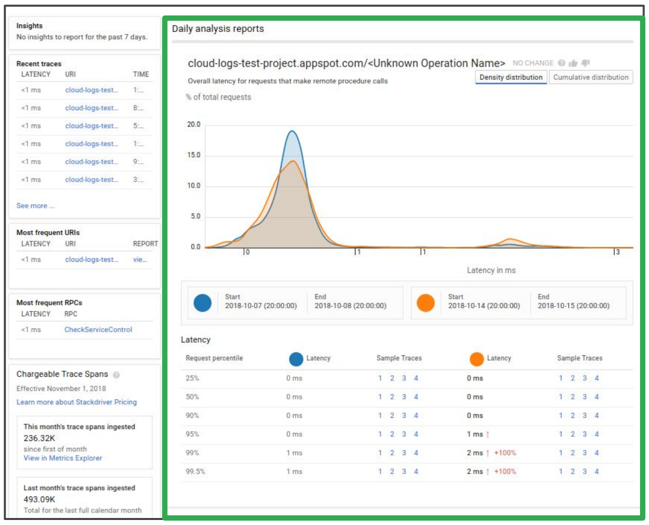
Latency

Request percentile	● Latency	Sample Traces	● Latency	Sample Traces
25%	0 ms	1 2 3 4	0 ms	1 2 3 4
50%	0 ms	1 2 3 4	0 ms	1 2 3 4
90%	0 ms	1 2 3 4	0 ms	1 2 3 4
95%	0 ms	1 2 3 4	1 ms ↑	1 2 3 4
99%	1 ms	1 2 3 4	2 ms ↑ +100%	1 2 3 4
99.5%	1 ms	1 2 3 4	2 ms ↑ +100%	1 2 3 4

Google Trace Overview is designed to show you a lot of high level tracing information. This information includes:

- auto-generated performance Insights, (An example of an Insight might be that your application is making too many calls to your datastore, which could affect performance.)
- lists of recent traces,
- most frequent URIs and RPC calls,
- information on chargeable spans, and
- a daily analysis report.

Trace Overview



Insights
No insights to report for the past 7 days.

Recent traces

LATENCY	URI	TIME
<1 ms	cloud-logs-test...	1:...
<1 ms	cloud-logs-test...	8:...
<1 ms	cloud-logs-test...	5:...
<1 ms	cloud-logs-test...	1:...
<1 ms	cloud-logs-test...	9:...
<1 ms	cloud-logs-test...	3:...

See more ...

Most frequent URIs

LATENCY	URI	REPORT
<1 ms	cloud-logs-test...	vie...

Most frequent RPCs

LATENCY	RPC
<1 ms	CheckServiceControl

Chargeable Trace Spans
Effective November 1, 2018
Learn more about Stackdriver Pricing

This month's trace spans ingested
236.32K
since first of month
View in Metrics Explorer

Last month's trace spans ingested
493.09K
Total for the last full calendar month

Daily analysis reports

cloud-logs-test-project.appspot.com/<Unknown Operation Name> NO CHANGE
Overall latency for requests that make remote procedure calls

🔵	Start 2018-10-07 (20:00:00)	End 2018-10-08 (20:00:00)	
🟠	Start 2018-10-14 (20:00:00)	End 2018-10-15 (20:00:00)	

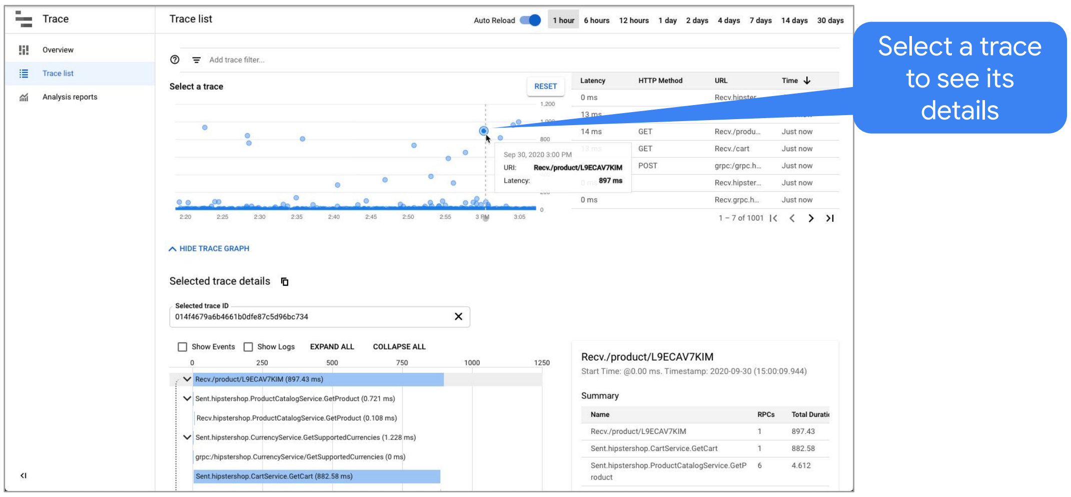
Latency

Request percentile	🔵 Latency	Sample Traces	🟠 Latency	Sample Traces
25%	0 ms	1 2 3 4	0 ms	1 2 3 4
50%	0 ms	1 2 3 4	0 ms	1 2 3 4
90%	0 ms	1 2 3 4	0 ms	1 2 3 4
95%	0 ms	1 2 3 4	1 ms ↑	1 2 3 4
99%	1 ms	1 2 3 4	2 ms ↑ +100%	1 2 3 4
99.5%	1 ms	1 2 3 4	2 ms ↑ +100%	1 2 3 4

The daily analysis displays up to three auto-generated reports. Each report displays the latency data for the previous day for a single RPC endpoint. If data for an endpoint is available from seven days earlier, that earlier data is included in the graph for comparison purposes. A report is generated for a RPC endpoint only if it is one of the three most frequent RPC endpoints, and only if there are at least 100 traces available.

If enough data isn't available to create at least one auto-generated report, Trace prompts you to create a custom analysis report.

Trace List Windows Shows Traces and Latencies



Sep 30, 2020 3:00 PM
URI: Recv./product/L9ECAV7KIM
Latency: 897 ms

Latency	HTTP Method	URL	Time ↓
0 ms		Recv.hipster...	[illegible]
13 ms	[illegible]	[illegible]	[illegible]
14 ms	GET	Recv./produ...	Just now
13 ms	GET	Recv./cart	Just now
[illegible]	POST	grpc:/grpc.h...	Just now
0 ms		Recv.hipster...	Just now
0 ms		Recv.grpc.h...	Just now

1 – 7 of 1001

Selected trace ID
014f4679a6b4661b0dfe87c5d96bc734

Recv./product/L9ECAV7KIM (897.43 ms)
Sent.hipstershop.ProductCatalogService.GetProduct (0.721 ms)
Recv.hipstershop.ProductCatalogService.GetProduct (0.108 ms)
Sent.hipstershop.CurrencyService.GetSupportedCurrencies (1.228 ms)
grpc:/hipstershop.CurrencyService/GetSupportedCurrencies (0 ms)
Sent.hipstershop.CartService.GetCart (882.58 ms)

Recv./product/L9ECAV7KIM
Start Time: @0.00 ms. Timestamp: 2020-09-30 (15:00:09.944)

Summary

Name	RPCs	Total Duratio
Recv./product/L9ECAV7KIM	1	897.43
Sent.hipstershop.CartService.GetCart	1	882.58
Sent.hipstershop.ProductCatalogService.GetProduct	6	4.612

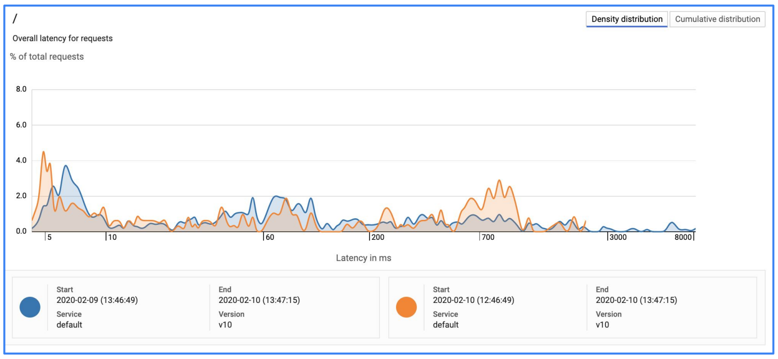
The **Trace list** window lets you find and examine individual traces in detail. Select a time interval, and you can view and inspect all spans for a trace, view summary information for a request, and view detailed information for each span in the trace from this window.

To restrict the traces being investigated, you add filters. For example, you can add a filter to display only traces whose latency exceeds 1 second.

Each dot in the latency graph corresponds to a specific request. The *(x,y)* coordinates for a request correspond to the request's time and latency. Clicking a dot will display the trace details view we saw a couple of slides ago.

The recent-request table displays the 5 most recent requests and contains the last 1,000 traces.

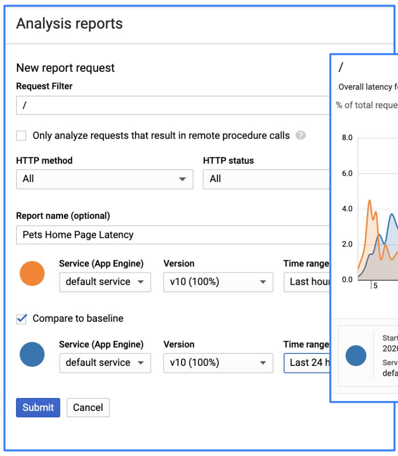
Analysis Reports Show Requests Over Time and Compares Versions and Timespans



	Start	End
●	2020-02-09 (13:46:49)	2020-02-10 (13:47:15)
	Service: default	Version: v10
●	2020-02-10 (12:46:49)	2020-02-10 (13:47:15)
	Service: default	Version: v10

Analysis reports in Cloud Trace show you an overall view of the latency for all requests, or a subset of requests, to your application. This will be similar to the daily report viewed on the Trace Overview main page.

Analysis Reports Show Requests Over Time and Compares Versions and Timespans



Analysis reports

New report request

Request Filter: /

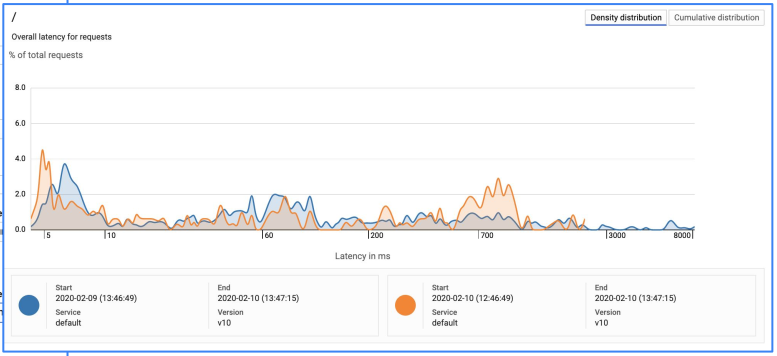
☐ Only analyze requests that result in remote procedure calls

HTTP method	All
HTTP status	All
Report name (optional)	Pets Home Page Latency

● Service (App Engine): default service — Version: v10 (100%) — Time range: Last hour

☑ Compare to baseline

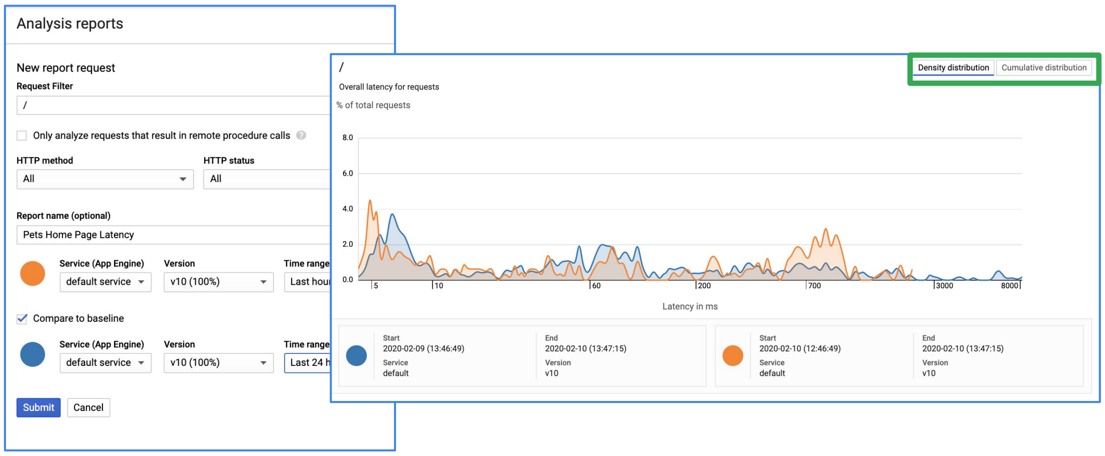
● Service (App Engine): default service — Version: v10 (100%) — Time range: Last 24 h



	Start	End
●	2020-02-09 (13:46:49)	2020-02-10 (13:47:15)
	Service: default	Version: v10
●	2020-02-10 (12:46:49)	2020-02-10 (13:47:15)
	Service: default	Version: v10

Custom reports can be created for particular request URIs and other search qualifiers.

Analysis Reports Show Requests Over Time and Compares Versions and Timespans



Analysis reports

New report request

Request Filter: /

☐ Only analyze requests that result in remote procedure calls

HTTP method: All

HTTP status: All

Report name (optional): Pets Home Page Latency

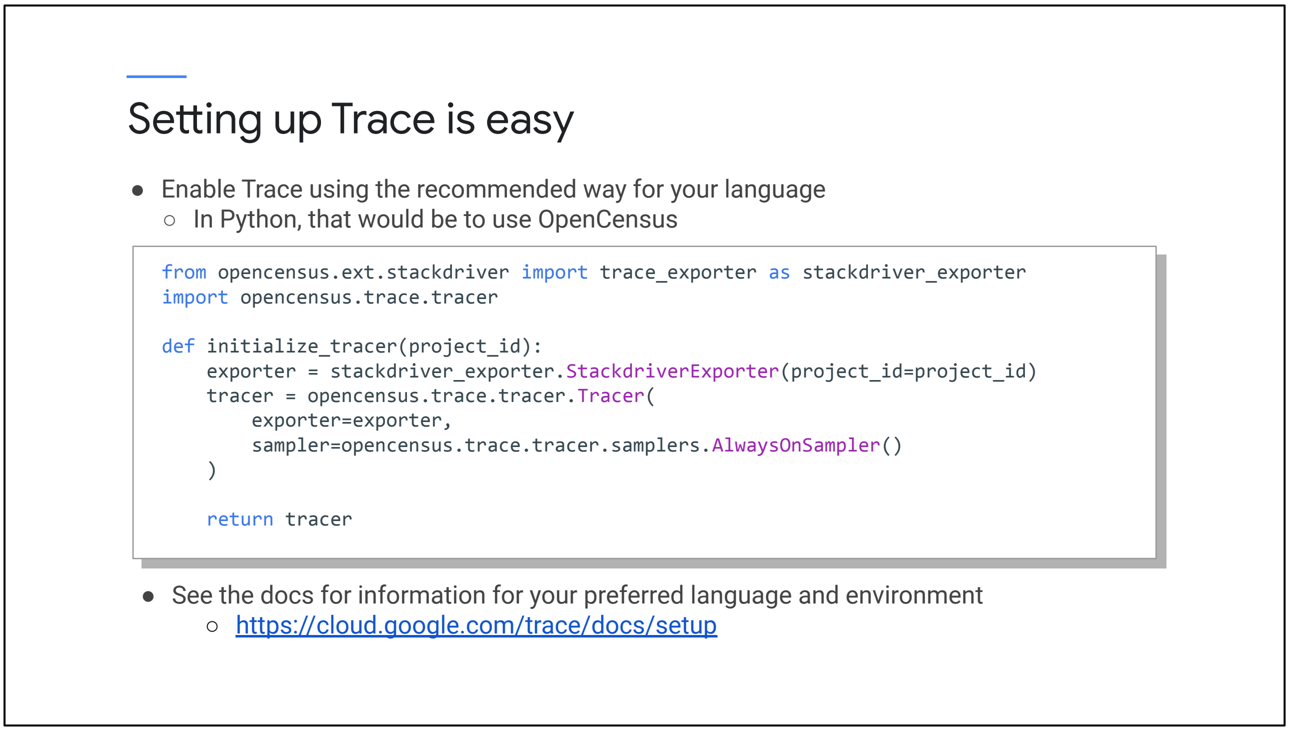
● Service (App Engine): default service — Version: v10 (100%) — Time range: Last hour

☑ Compare to baseline

● Service (App Engine): default service — Version: v10 (100%) — Time range: Last 24 h

●	Start 2020-02-09 (13:46:49)	End 2020-02-10 (13:47:15)
	Service default	Version v10
●	Start 2020-02-10 (12:46:49)	End 2020-02-10 (13:47:15)
	Service default	Version v10

The report can show results as both a density distribution, as in the screenshot, or as a cumulative distribution.



Setting up Trace is easy

- Enable Trace using the recommended way for your language
  - In Python, that would be to use OpenCensus

```
from opencensus.ext.stackdriver import trace_exporter as stackdriver_exporter
import opencensus.trace.tracer

def initialize_tracer(project_id):
    exporter = stackdriver_exporter.StackdriverExporter(project_id=project_id)
    tracer = opencensus.trace.tracer.Tracer(
        exporter=exporter,
        sampler=opencensus.trace.tracer.samplers.AlwaysOnSampler()
    )

    return tracer
```

- See the docs for information for your preferred language and environment
  - https://cloud.google.com/trace/docs/setup

Setting up your code to use Trace is easy.

Depending on your language, there are three ways to implement tracing for your applications:

- Use OpenTelemetry and the associated Cloud Trace client library. This is the recommended way to instrument your applications.
- Use OpenCensus if an OpenTelemetry client library is not available for your language.
- Use the Cloud Trace API and write custom methods to send tracing data to Cloud Trace.

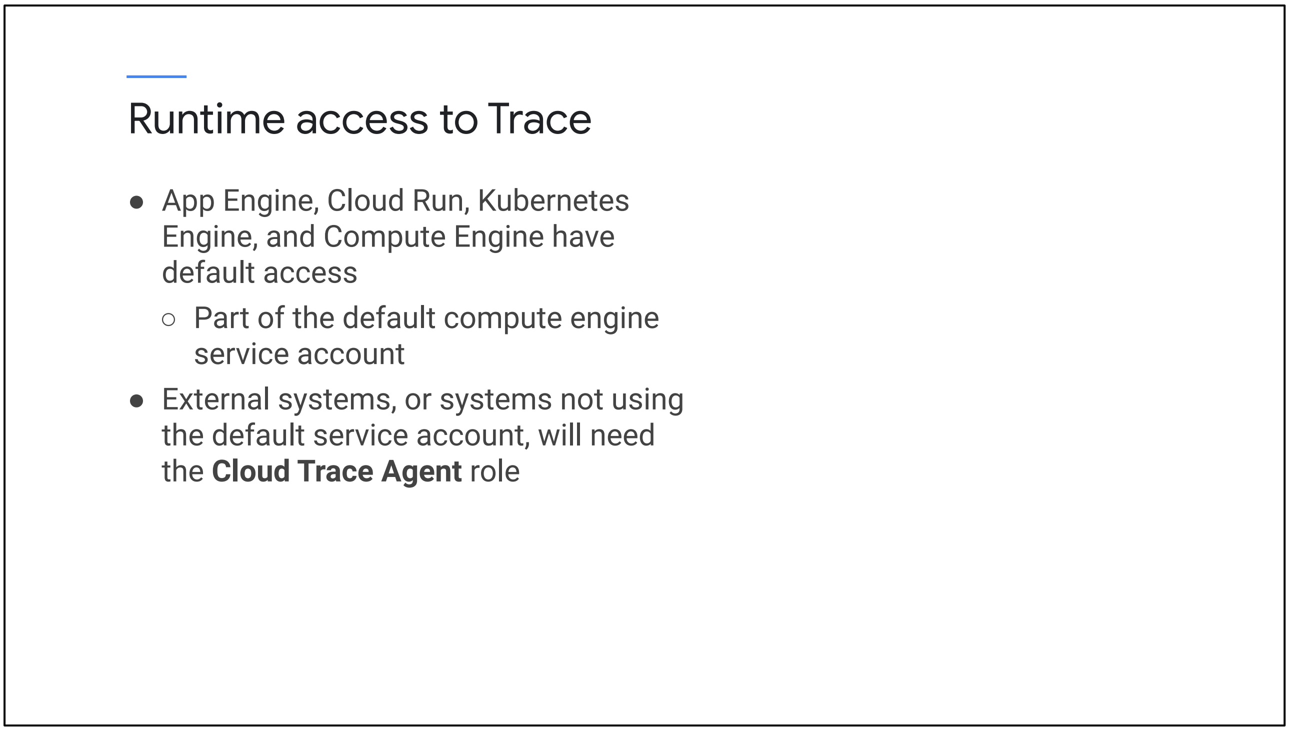
Make sure to check the documentation for language-specific recommendations.

The example on this slide, and used in the next lab, is written in Python. At the time of this writing, Google recommended using OpenCensus with Python.

In Python, first you would import the trace exporter and tracer.

Then, in some initialization section, you would create the exporter and tracer objects to be used later in code.

At this point, RPC spans would be created for your code automatically.



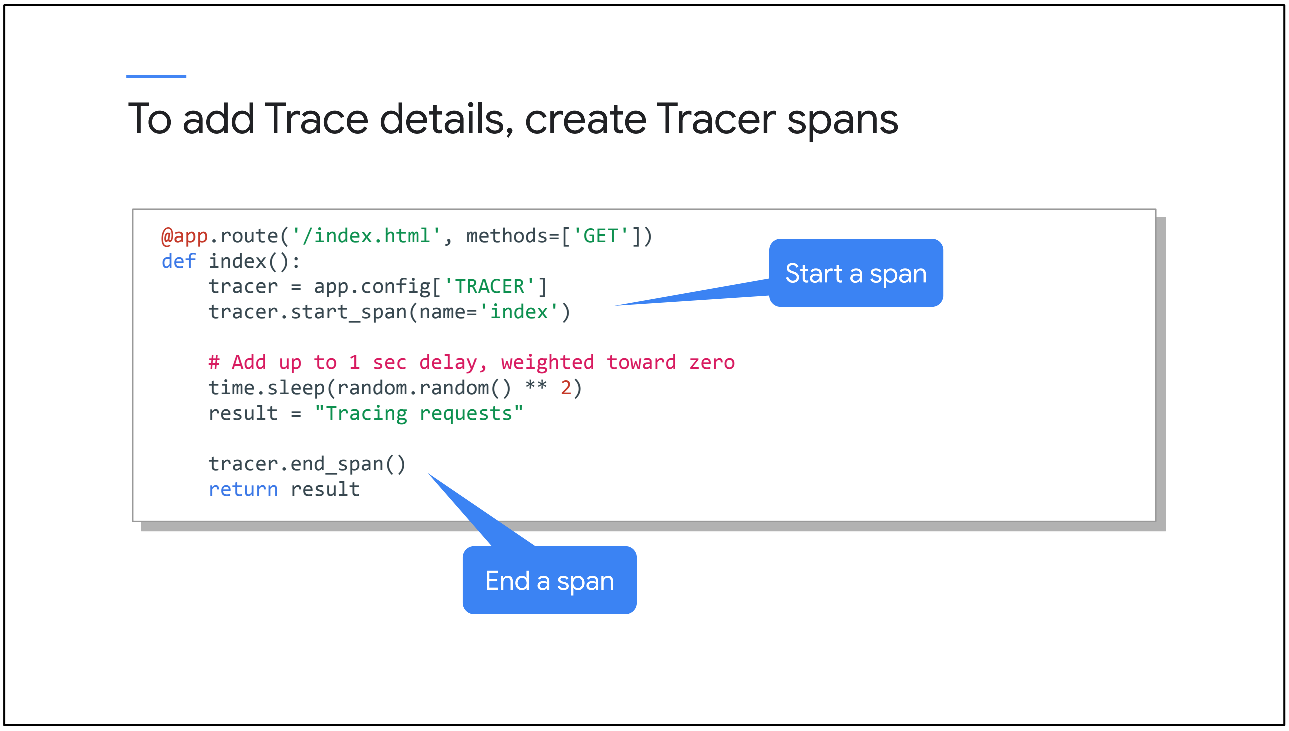
Runtime access to Trace

- App Engine, Cloud Run, Kubernetes Engine, and Compute Engine have default access
  - Part of the default compute engine service account
- External systems, or systems not using the default service account, will need the **Cloud Trace Agent** role

Like with using Debugger, Trace will need to offload tracing metrics to Google cloud.

App Engine, Cloud Run, Google Kubernetes Engine, and Compute Engine have default access. Compute Engine and GKE get that access through the default Compute Engine service account.

For external systems, or Compute Engine and GKE environments not running under the default service account, make sure they run under a service account with at least the Cloud Trace Agent role.



To add Trace details, create Tracer spans

```python
@app.route('/index.html', methods=['GET'])
def index():
    tracer = app.config['TRACER']
    tracer.start_span(name='index')

    # Add up to 1 sec delay, weighted toward zero
    time.sleep(random.random() ** 2)
    result = "Tracing requests"

    tracer.end_span()
    return result
```

Start a span
End a span

Trace automatically creates spans for RPC calls. When your application interacts with Firestore through the API, that would be an RPC call. But you can also create spans manually to enrich the data Trace collects.

Here, we see an example from a Python Flask web application. When a GET request comes into *index.html*, we pull a reference to the tracer we created a couple of slides ago, and we use it to create a new span named 'index.'

In the span, we manually create a 0-2 second delay, weighted towards 0. We set the result we return on the web page to "Tracing requests," and end the span.

Agenda

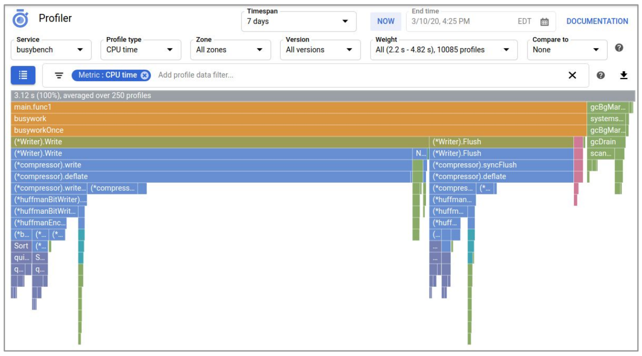
Trace

Profiler

Now that we can find logic errors and latency bottlenecks in our code, let’s profile memory and CPU usage.

Understand performance with Cloud Profiler

- Continuous profiling of production systems
- Statistical, low-overhead memory and CPU profiler
- Contextualized to your code


Profiler

Timespan: 7 days | NOW | End time: 3/10/20, 4:25 PM EDT | DOCUMENTATION

Service: busybench | Profile type: CPU time | Zone: All zones | Version: All versions | Weight: All (2.2 s - 4.82 s), 10085 profiles | Compare to: None

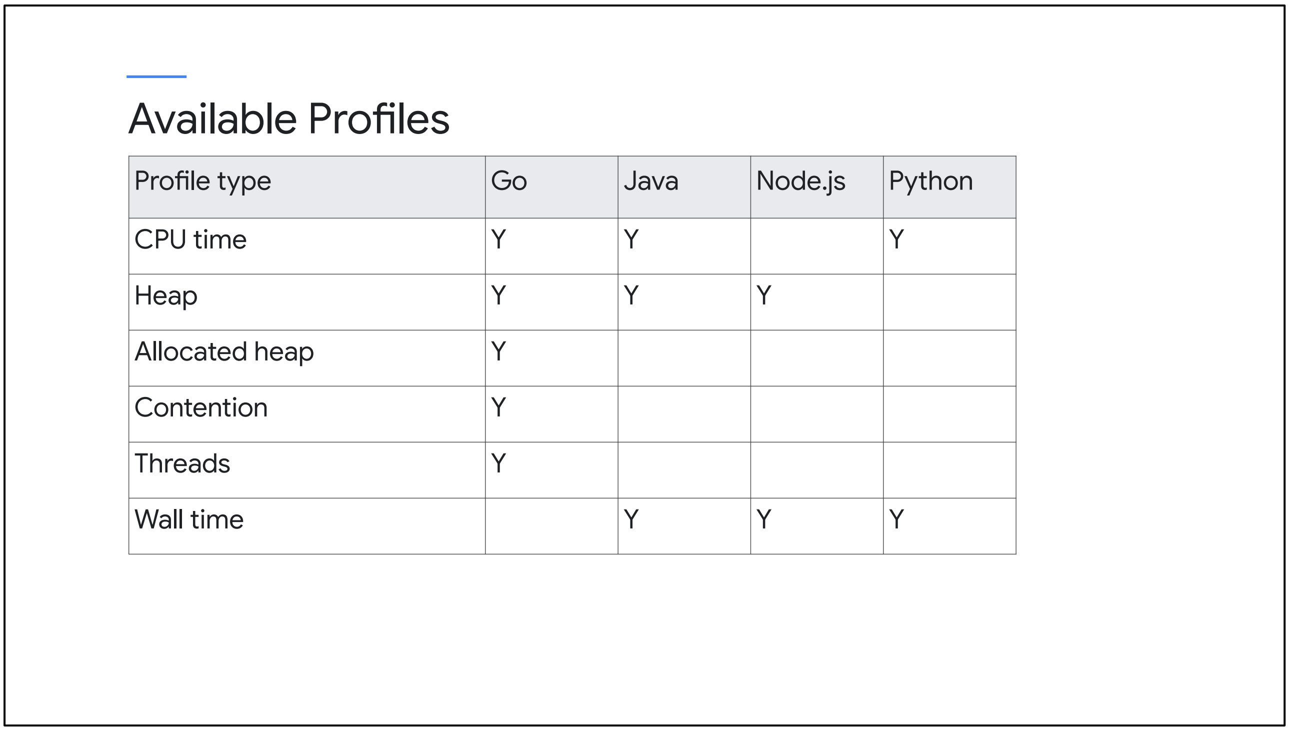
Metric : CPU time | Add profile data filter...

3.12 s (100%), averaged over 250 profiles

Understanding the performance of production systems is notoriously difficult. Attempting to measure performance in test environments usually fails to replicate the pressures on a production system.

Continuous profiling of production systems is an effective way to discover where resources like CPU cycles and memory are consumed as a service operates in its working environment.

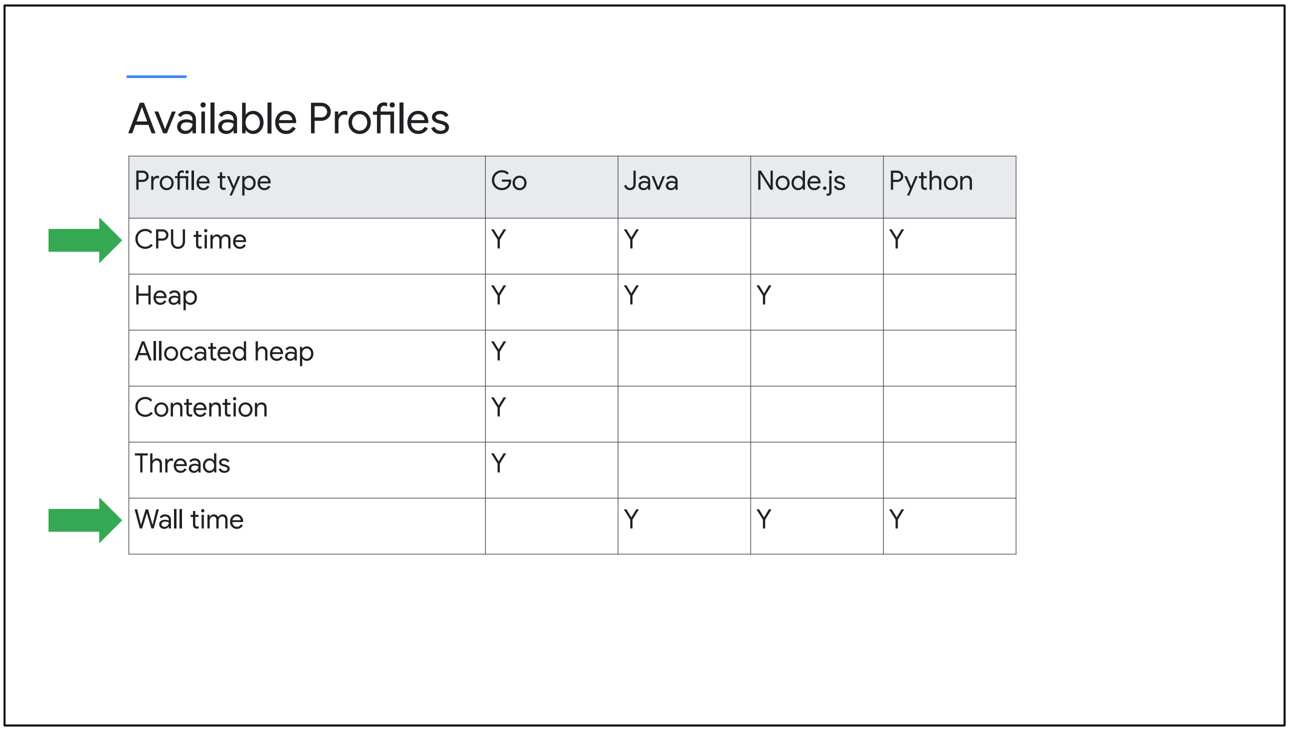
Cloud Profiler is a statistical, low-overhead profiler that continuously gathers CPU usage and memory-allocation information from your production applications. It attributes that information to the source code that generated it, helping you identify the parts of your application that are consuming the most resources, and otherwise illuminating the performance of your application characteristics.



Available Profiles

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

The profiling types available vary by language. Check the Google Cloud Profiler documentation for the most recent options..

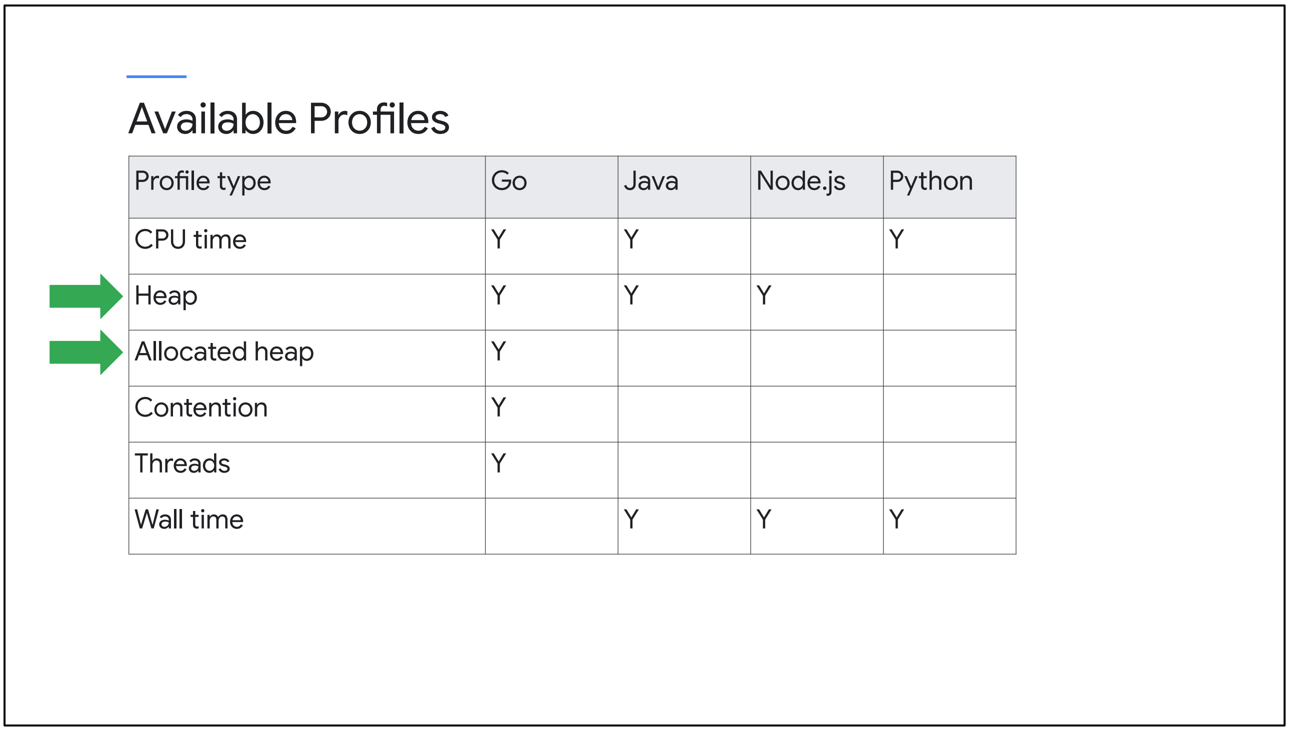


Available Profiles

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

For the CPU metrics you will find:

- **CPU time** is the time the CPU spends executing a block of code, not including the time it was waiting or processing instructions for something else.
- **Wall time** is the time it takes to run a block of code, including all wait time, including that for locks and thread synchronization. The wall time for a block of code can never be less than the CPU time.

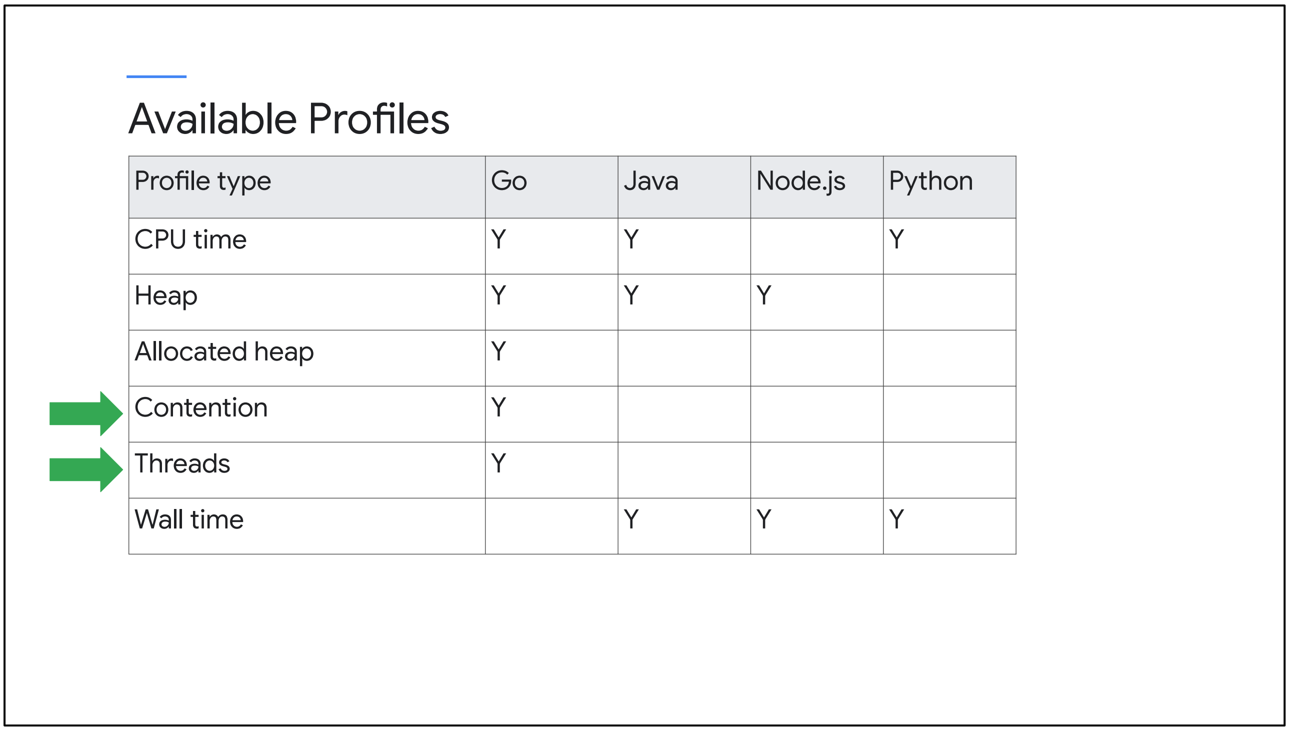


Available Profiles

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

For Heap you have:

- **Heap** is the amount of memory allocated in the program's heap when the profile is collected.
- ***Allocated heap*** is the total amount of memory that was allocated in the program's heap, including memory that has been freed and is no longer in use.

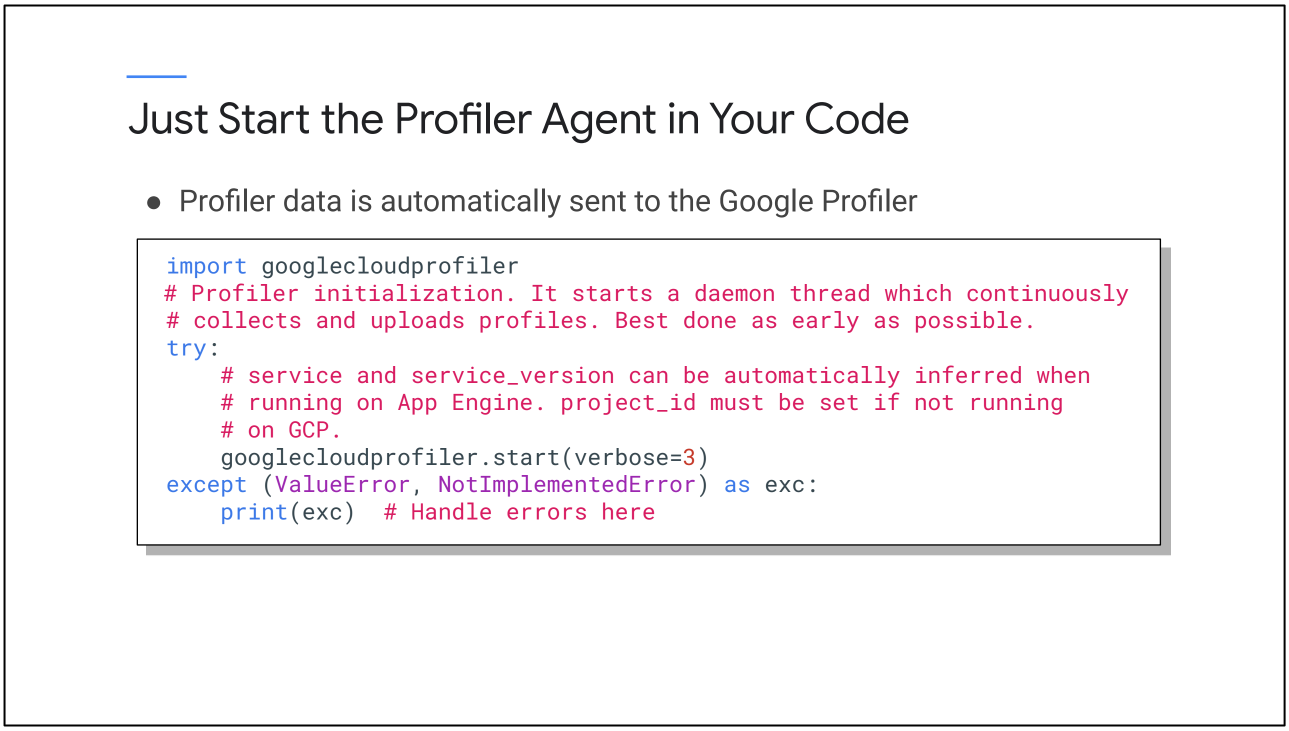


Available Profiles

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

And for Threads you have:

- **Contention** provides information about threads stuck waiting for other threads.
- **Threads** contains thread counts.



Just Start the Profiler Agent in Your Code

- Profiler data is automatically sent to the Google Profiler

```python
import googlecloudprofiler
# Profiler initialization. It starts a daemon thread which continuously
# collects and uploads profiles. Best done as early as possible.
try:
    # service and service_version can be automatically inferred when
    # running on App Engine. project_id must be set if not running
    # on GCP.
    googlecloudprofiler.start(verbose=3)
except (ValueError, NotImplementedError) as exc:
    print(exc)  # Handle errors here
```

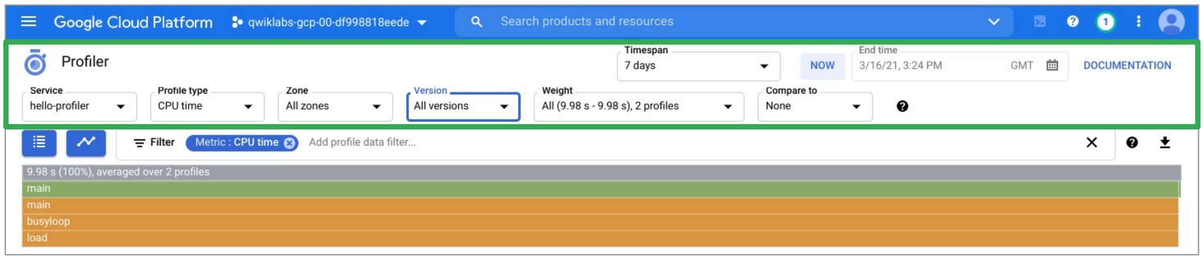
Like with Google's other application performance management products, the exact setup steps will vary by language, so check the documentation. Here, we are sticking with our Python application, which will be running on App Engine.

Before you start, make sure the **Profiler API** is enabled in your project.

Start by importing the *googlecloudprofiler* package.

Then, early as possible in your code, start the profiler. In this example, we are setting the logging level (verbose) to 3, or debug level. That will log all messages. The default would be 0 or error only.

Select Profile to be Analyzed

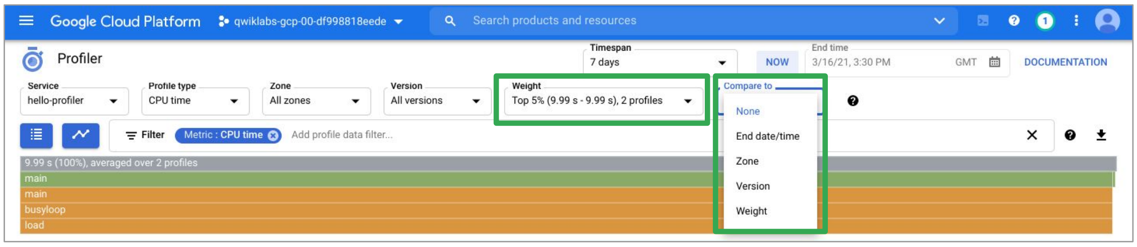

Google Cloud Platform
qwiklabs-gcp-00-df998818eede
Search products and resources
Profiler
NOW
DOCUMENTATION
Filter
Metric : CPU time
Add profile data filter...

Timespan: 7 days
End time: 3/16/21, 3:24 PM GMT
Service: hello-profiler
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: All (9.98 s - 9.98 s), 2 profiles
Compare to: None

```
9.98 s (100%), averaged over 2 profiles
main
main
busyloop
load
```

At the top of the Profiler interface, you can select the profile to be analyzed. You can select by timespan, service, profile type, zone, version, etc.

Select Profile to be Analyzed


Google Cloud Platform
qwiklabs-gcp-00-df998818eede
Search products and resources
Profiler
DOCUMENTATION
NOW
Filter
Add profile data filter...

Timespan: 7 days
End time: 3/16/21, 3:30 PM GMT

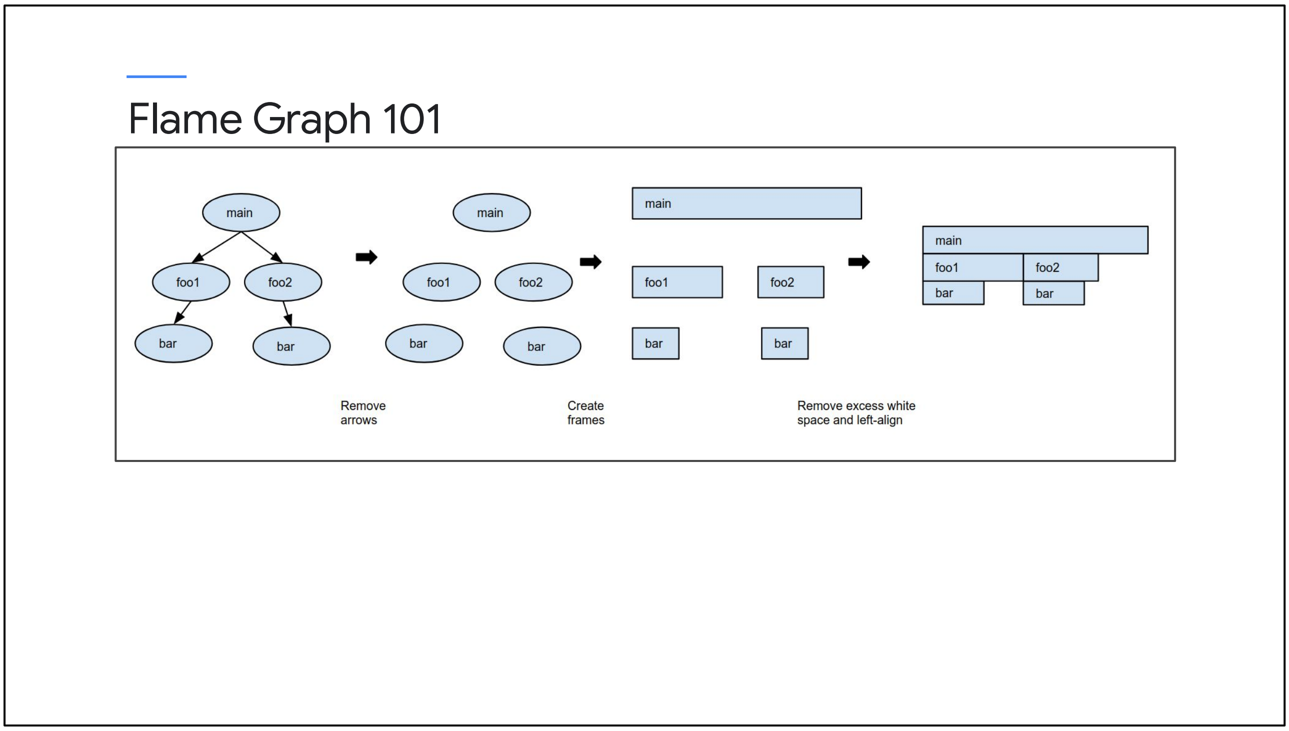
Service: hello-profiler
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: Top 5% (9.99 s - 9.99 s), 2 profiles
Compare to: None / End date/time / Zone / Version / Weight

Metric : CPU time

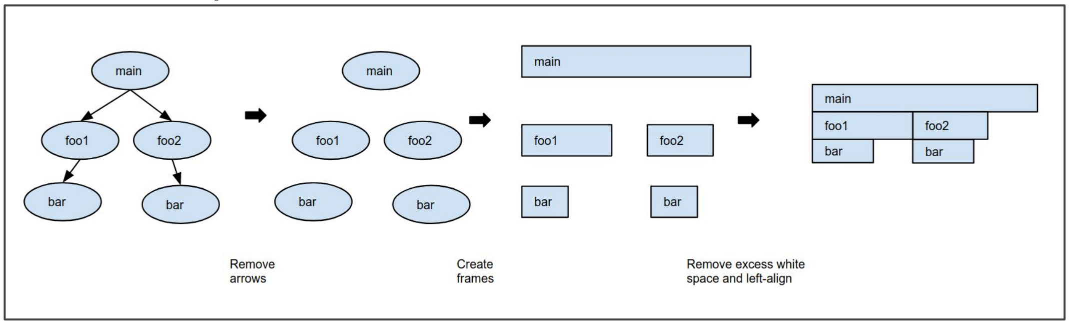
9.99 s (100%), averaged over 2 profiles
main
main
busyloop
load

The weight will limit the subsequent flame graph to particular peak consumptions. Top 5%, for example.

Compare to allows the comparison of two profiles.



Flame Graph 101

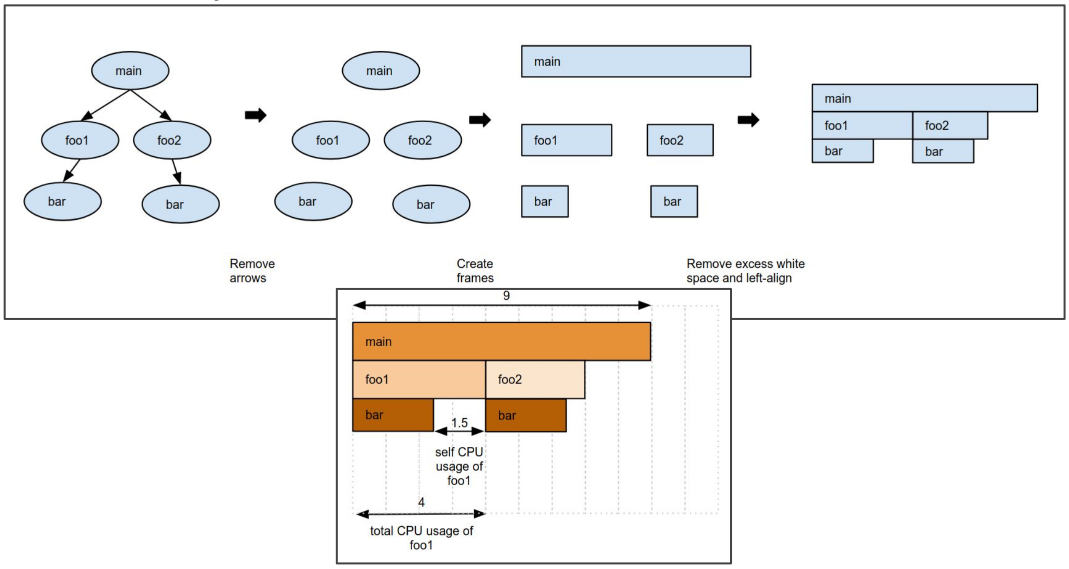


Cloud Profiler displays profiling data by using [Flame Graphs](). Unlike trees and standard graphs, flame graphs make efficient use of screen space by representing a large amount of information in a compact and readable format.

Take a look at this example. We have a basic application with a main method which calls foo1, which in turn calls bar. Then main calls foo2, which also calls bar.

As you move through the graphic left to right, you can see how the information is collapsed. First, by removing arrows, then by creating frames, and finally by removing spaces and left aligning.

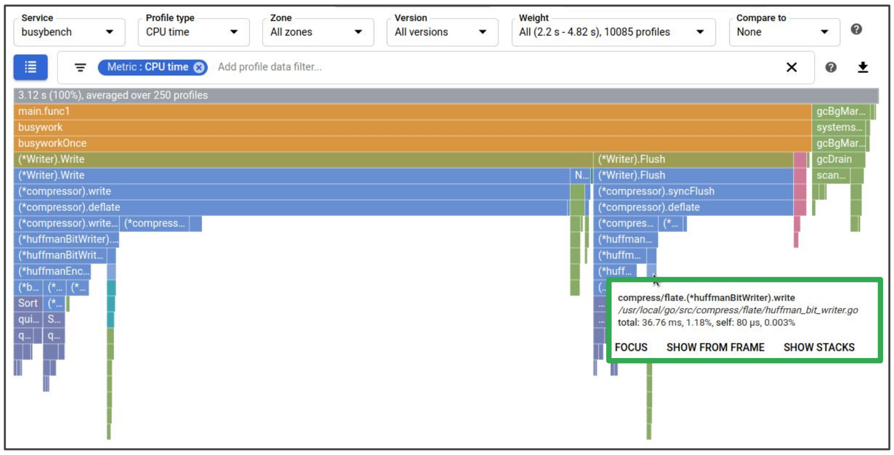
Flame Graph 101



In the bottom view, you see the result as it would appear in the Profiler. If we were looking at CPU time, then you can see the main method takes a total of 9 seconds.

Below the main bar, you can see how that CPU time was spent—some in main itself, but most in the calls to foo1 and foo2. And most of the foo time (cough) was spent in bar. So, if we could make bar faster, we could really save some time in main.

Hover Over Method For



Service: busybench
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: All (2.2 s - 4.82 s), 10085 profiles
Compare to: None

Metric : CPU time
Add profile data filter...

3.12 s (100%), averaged over 250 profiles

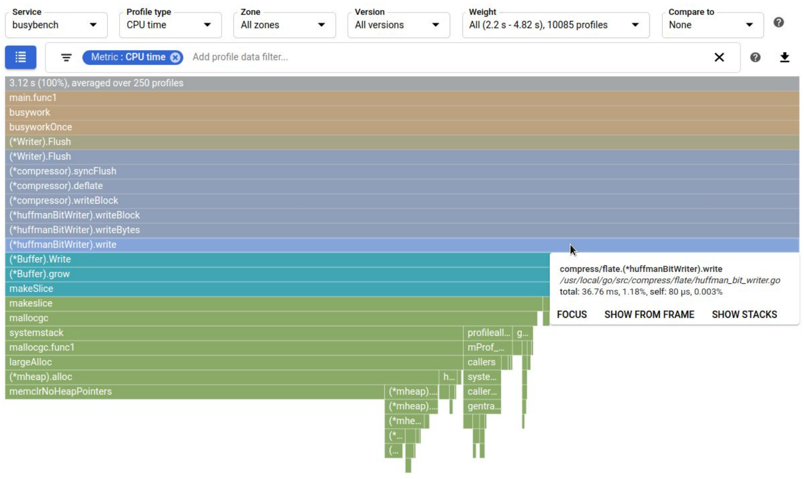
compress/flate.(*huffmanBitWriter).write
/usr/local/go/src/compress/flate/huffman_bit_writer.go
total: 36.76 ms, 1.18%, self: 80 µs, 0.003%

Here we have a full example.

When you hold the pointer over a frame, a tooltip opens and displays additional information including:

- The function name
- The source file location
- And some metric consumption information

Select a Frame to Zoom


Service: busybench
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: All (2.2 s - 4.82 s), 10085 profiles
Compare to: None

Metric : CPU time
Add profile data filter...

3.12 s (100%), averaged over 250 profiles

compress/flate.(*huffmanBitWriter).write
/usr/local/go/src/compress/flate/huffman_bit_writer.go
total: 36.76 ms, 1.18%, self: 80 µs, 0.003%

If you click a frame, the graph is redrawn, making the selected method's call stack more visible.