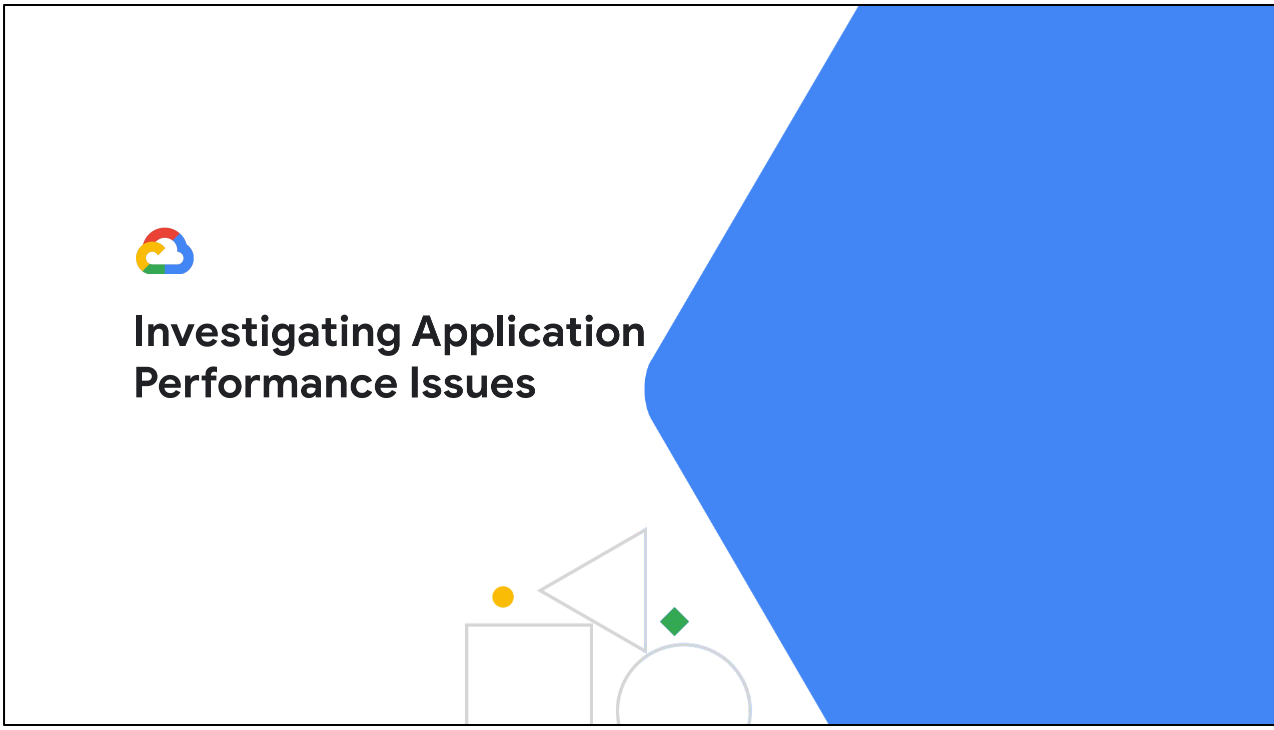


When deploying applications to Google Cloud, the Application Performance Management products (Cloud Trace, Cloud Debugger, and Cloud Profiler) provide a suite of tools to give insight into how your code and services are functioning, and to help troubleshoot where needed.

Agenda

Debugger

Trace

Profiler

In this module, you will learn to:

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

Agenda

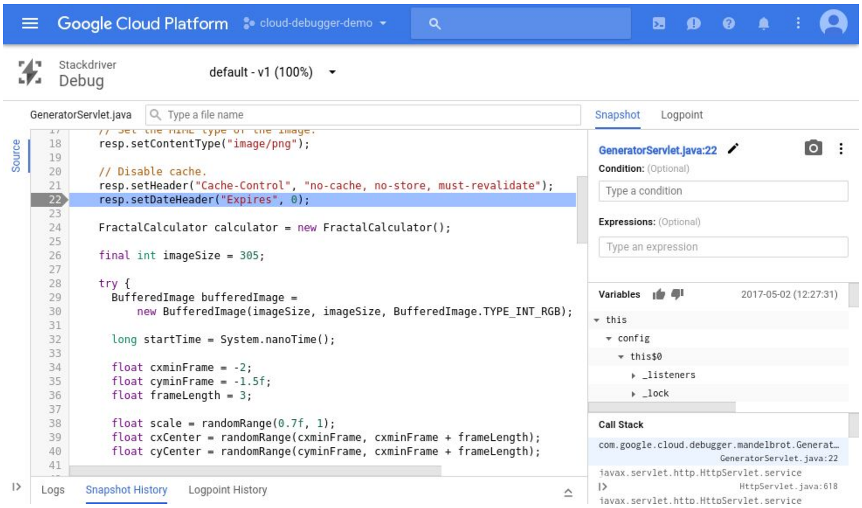
Debugger

Trace

Profiler

Let’s start with the debugger.

Google Cloud Debugger



Cloud Debugger lets you inspect the state of a running application in real-time, without stopping or slowing it down. Your users are not impacted while you capture the call stack and variables at any location in your source code. You can use it to understand the behavior of your code in production, as well as analyze its state to find those hard-to-find bugs.

Debug Running Apps with Cloud Debugger

Debug code written in:

- Java
- Python
- Go
- Node.js
- Ruby
- PHP
- .NET

Debug code running on:

- App Engine
- Compute Engine
- Kubernetes Engine
- Cloud Run
- Elsewhere

Debug code stored in:

- Local files
- Cloud Source Repositories
- GitHub
- Bitbucket
- GitLab
- App Engine

You can run Debugger against code written in a number of modern languages, including Java, Python, Go, Node.js, Ruby, PHP, and .NET.

The code can be running in any of Google's compute technologies, including App Engine, Compute Engine, Kubernetes Engine, Cloud Run, and elsewhere.

Debugger will need access to your application source code. It can pull it from the local filesystem, Google's Cloud Source Repositories, App Engine, or several supported third-party source repositories, including GitHub, Bitbucket, and GitLab.

Note, not all features are available in every language and runtime combination.

Debugger Must Be Enabled

- Details are language and environment specific; for example, in Python on App Engine, you...
  - Add `google-python-cloud-debugger` to your requirements.txt file
  - Import `googleclouddebugger`
  - Add the following code to your program

```
try:
  import googleclouddebugger
  googleclouddebugger.enable()
except ImportError:
  pass
```

To use Debugger, first enable the Cloud Debugger API in your project. Debugger then needs to be enabled in your code. Details are both language-specific, and can be found in the documentation.

Here you see an example of setting up Debugger in a Python application running on App Engine standard.

Start by adding the *google-python-cloud-debugger* dependency into your requirements.txt file.

Then, as early as possible in your code (main.py?), import *googleclouddebugger,* and use it to enable the debugger.

Runtime Access to Debugger

- App Engine and Cloud Run are already configured
- For Compute Engine, GKE, and external system access:
  - Service Account with the **Cloud Debugger Agent** role
  - Or add the following scopes to your VMs or cluster nodes
    - `https://www.googleapis.com/auth/cloud-platform`
    - `https://www.googleapis.com/auth/cloud_debugger`
  - Compute Engine Debugger enablement code slightly different, check the documentation

The Google Cloud compute resource where you're running your code will need permissions to upload telemetry.

App Engine and the managed version of Cloud Run will already have the access they need.

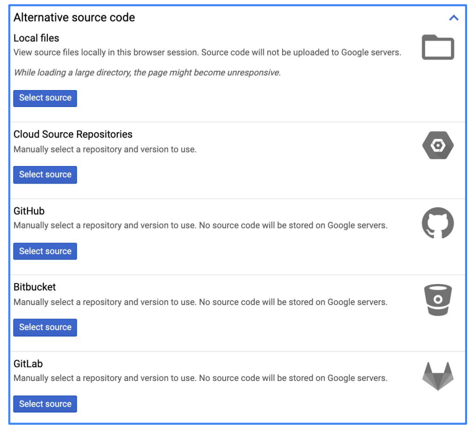
For Compute Engine VMs, Google Kubernetes Engine nodes, and external systems, create a service account with the Cloud Debugger Agent role, plus any other access the code will need, and make sure the code runs as that account.

VMs and Nodes running under the default compute engine service account (not a best practice) will need the two scopes mentioned on the current slide.

Please note that the way you enable the Debugger for code running in Compute Engine or external systems is slightly different from the example on the last slide, so please check the documentation.

Source Code Location

- App Engine standard will select code automatically
- Automatic selection in GAE Flex, GCE, GKE, and Cloud Run requires a *source-context.json* in application root folder
  - Can generate with *gcloud debug source gen-repo-info-file*
- Or, use **Alternative source code** and select manually



Debugger will also need access to your application's source code. App Engine standard will select the source automatically.

App Engine flex, Compute Engine, Google Kubernetes Engine, and Cloud Run can also select the source code automatically, but you'll have to provide a source context file in the application root to support the feature. You can generate this *source-context.json* file using the gcloud command on the slide.

To manually select the source, use the **Alternative source code** page seen on the right.

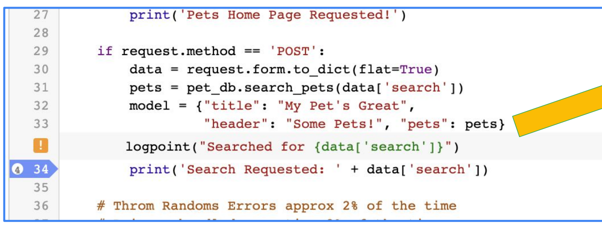
Set Breakpoints to Debug from Snapshots

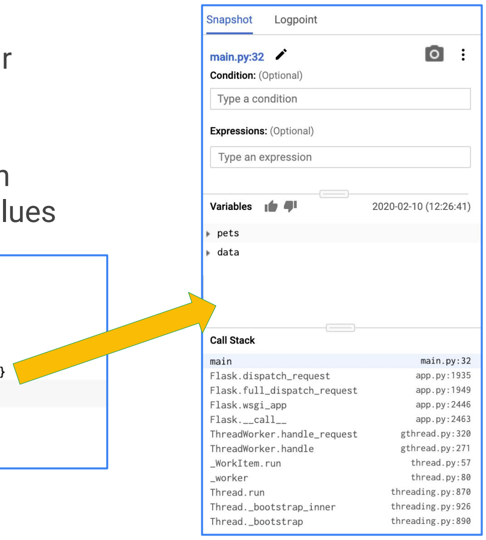
- Setting a breakpoint doesn't stop your code like a traditional debugger
- When the breakpoint is reached, a snapshot of your running app is taken
  - You can then inspect variable values

Unlike traditional debuggers, Cloud Debugger breakpoints do not stop code execution.
When the flow of execution passes the breakpoint, a nonintrusive snapshot is taken.
The snapshot captures local variables and the call stack state at that line, and you can inspect these at your leisure.

Set Breakpoints to Debug from Snapshots

- Setting a breakpoint doesn't stop your code like a traditional debugger
- When the breakpoint is reached, a snapshot of your running app is taken
  - You can then inspect variable values

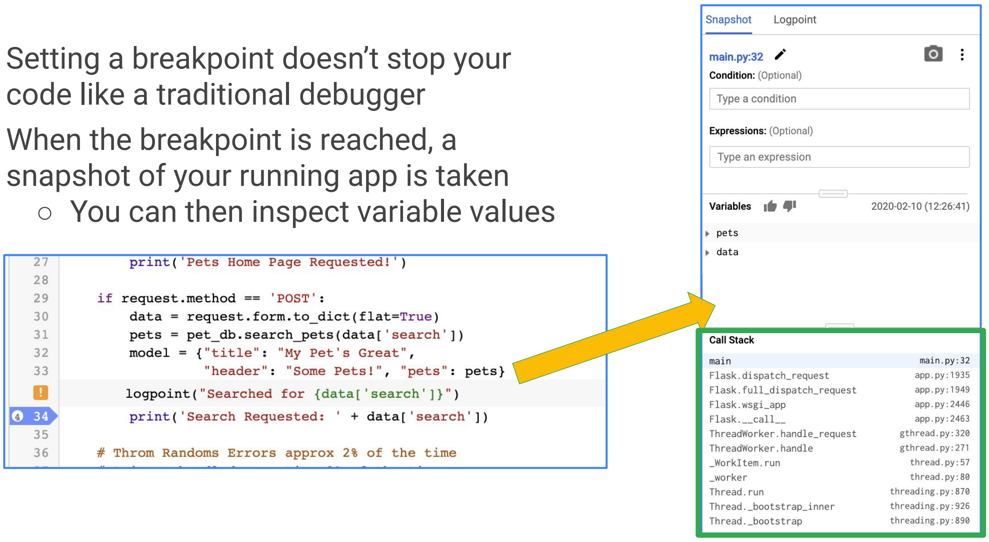




In the example, a breakpoint was set at line 34.
When the execution flow passed the breakpoint, a snapshot was taken.
You can see the details in the snapshot to the right.

Set Breakpoints to Debug from Snapshots

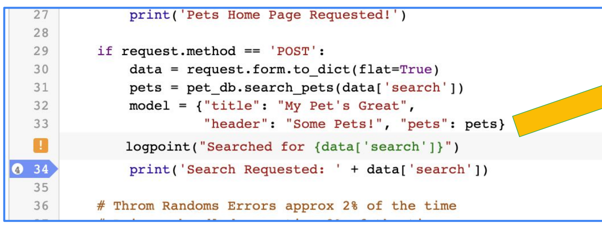
- Setting a breakpoint doesn't stop your code like a traditional debugger
- When the breakpoint is reached, a snapshot of your running app is taken
  - You can then inspect variable values

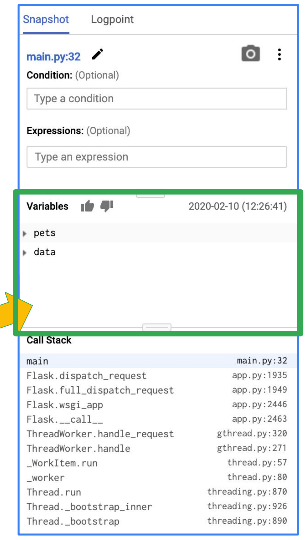


At the bottom, you see the call stack.

Set Breakpoints to Debug from Snapshots

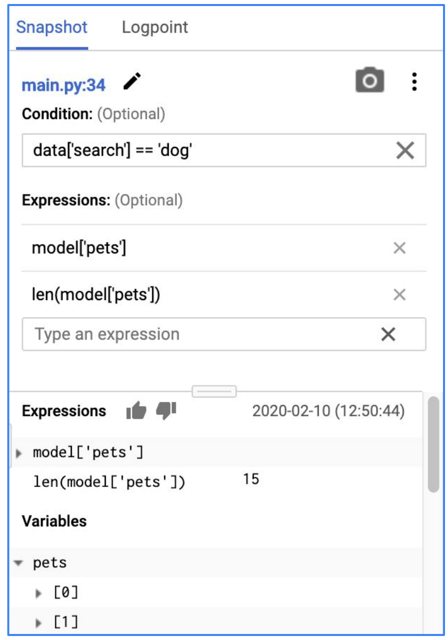
- Setting a breakpoint doesn't stop your code like a traditional debugger
- When the breakpoint is reached, a snapshot of your running app is taken
  - You can then inspect variable values



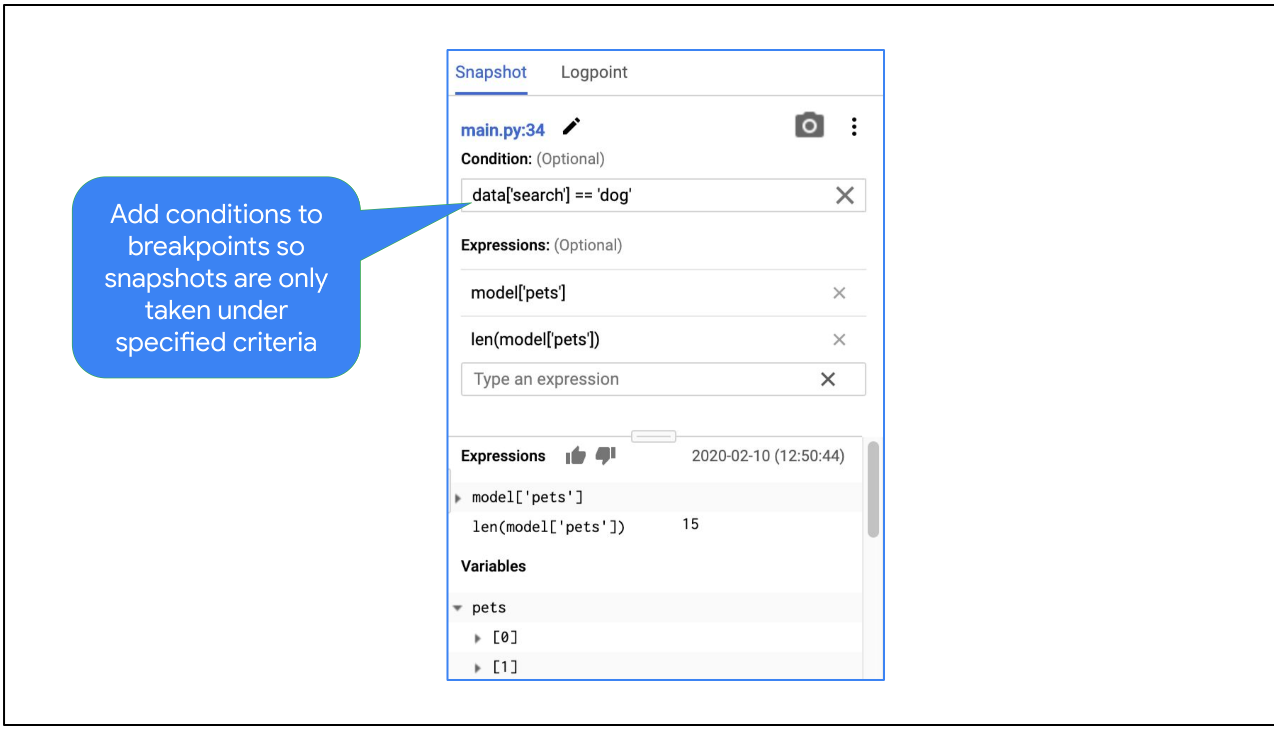


Above that, you could expand *pets* or *data* to see what those variables contained at that moment in time.

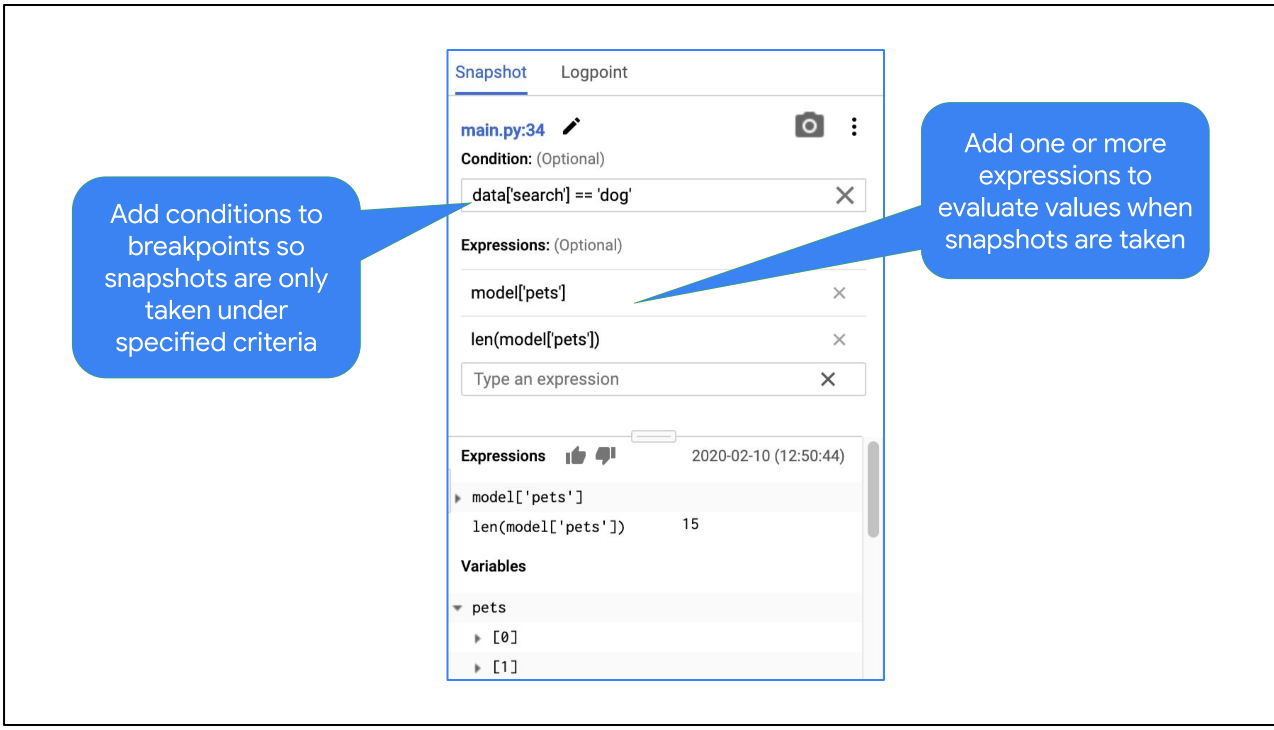
Note, it may take as long as 40 seconds for a new breakpoint to take effect.



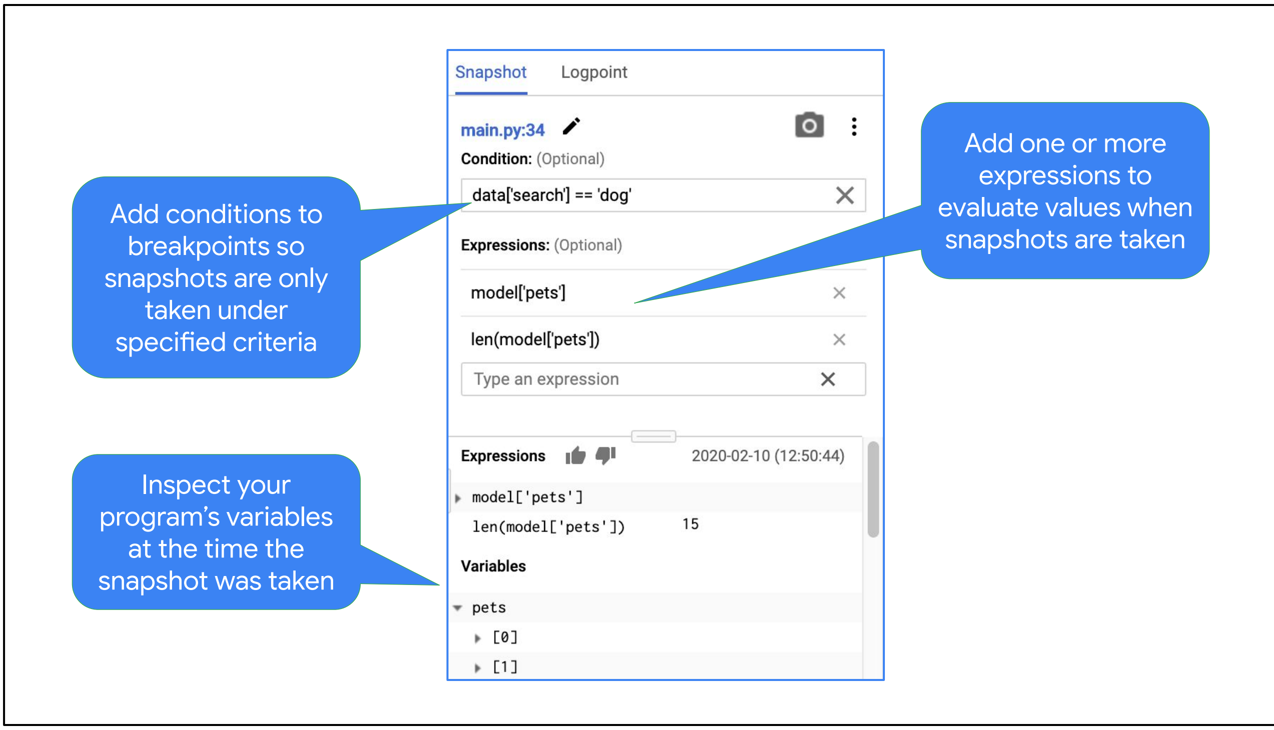
Here you see a closeup of the previous slide's line 34 snapshot.



Towards the top, note the condition that has been set for the breakpoint. We are telling debugger, "Only take a snapshot if the value in data-search is equal to dog."

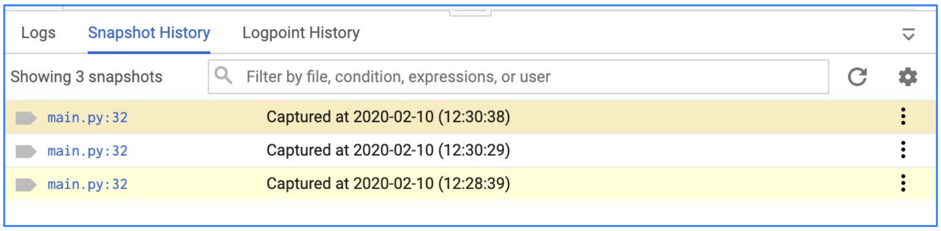


Below that, notice the expressions that have been created. Expressions will be evaluated when the snapshot is taken, and their values appear below. You might use an expression to access a variable that's out of the local set (global or static), or to simplify deep navigation.



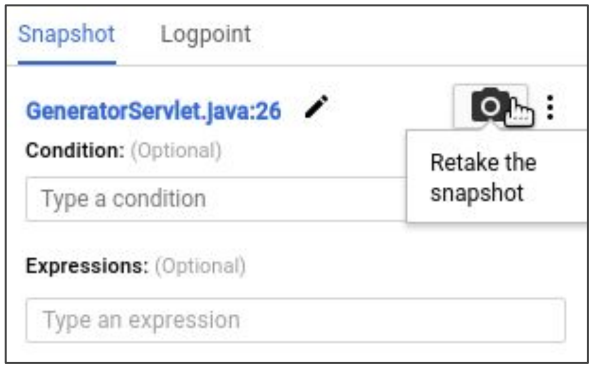
At the bottom are any local variables. You can use these to inspect your program's variable values at the time the snapshot was taken.

Snapshots History is Available



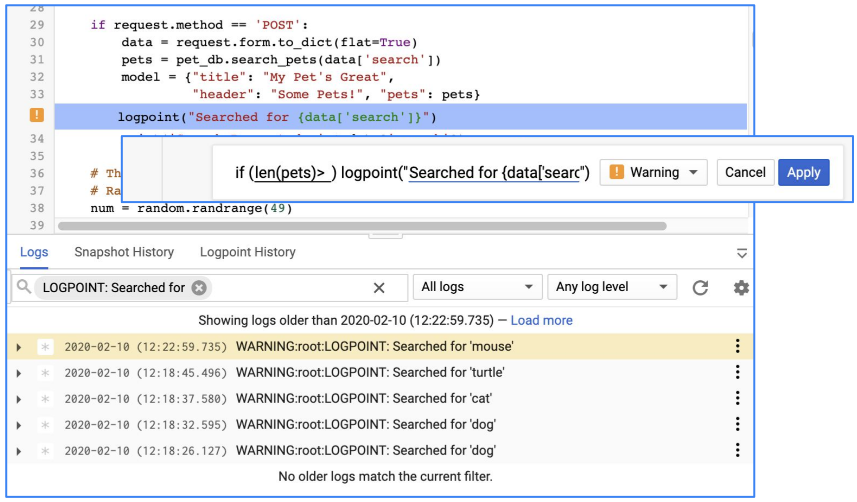
At the bottom of the Debugger interface, you'll find a searchable snapshot history panel. Use it when you've set multiple snapshot locations on a file, or to view snapshots that you've already taken.

Retake a Snapshot



A snapshot is only taken once. If you want to capture another snapshot of your app's data at the same location, you can manually retake it.

Dynamically Add Log Messages



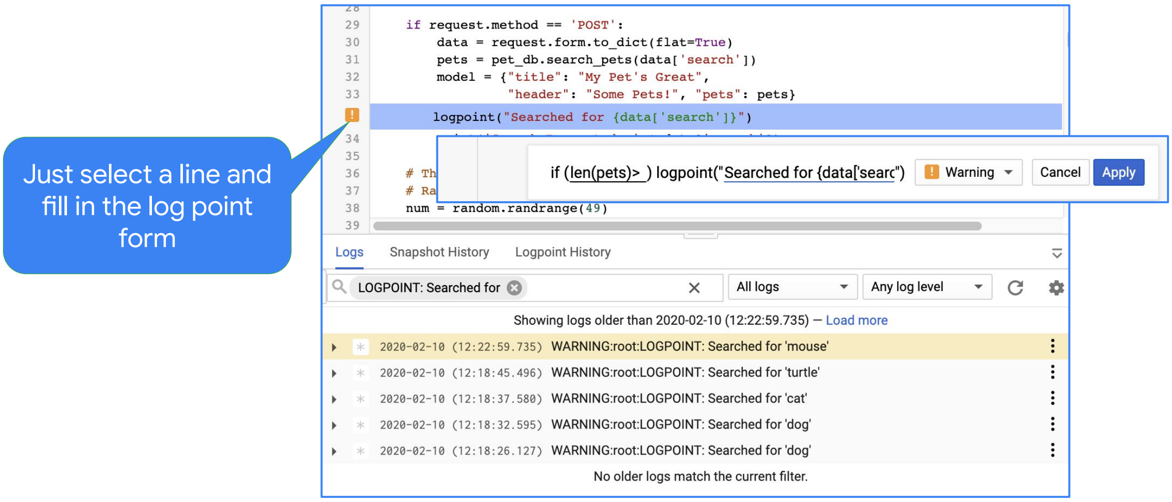
One of the classic techniques for debugging applications is adding log messages.

"Made it this far."

"Made it this far 2, and x=___"

The problem with this approach is that you have to edit the code to do it, and then re-deploy the application.

Dynamically Add Log Messages with Log Points

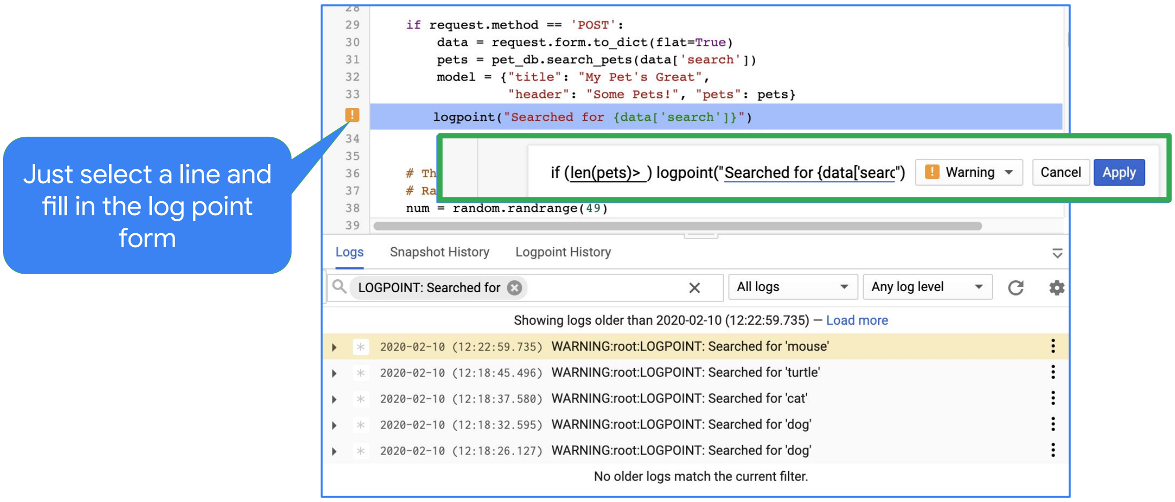


Logpoints allow you to inject logging into running services without editing, restarting, or interfering with the normal function of the service. Every time any instance executes code at the logpoint location, Cloud Debugger logs a message.

The output is sent to the appropriate log for the target's environment. On App Engine, for example, the output is sent to the request log in Cloud Logging.

Logpoints remain active for 24 hours after creation, or until they are deleted, or the service is redeployed.

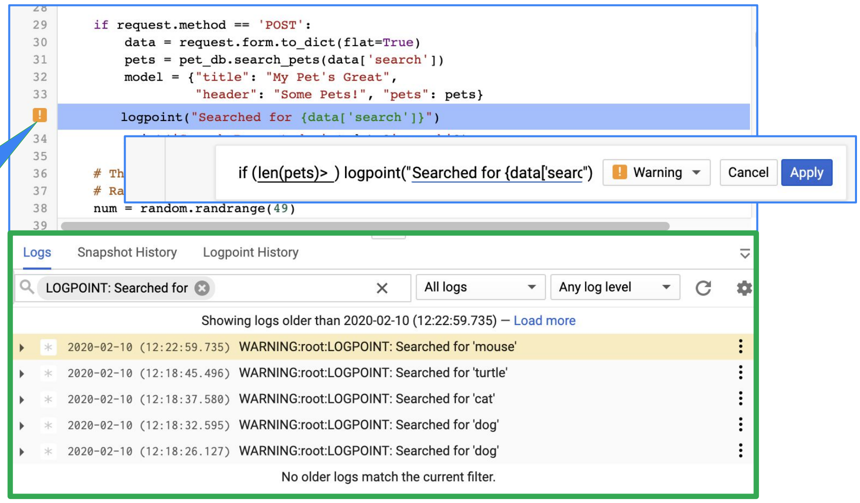
Dynamically Add Log Messages with Log Points



In this example, you see we're adding a logpoint to line 34. If the length of the pets collection is greater than 0, then we create a warning level log entry, "Searched for," followed by the term in data-search.

Dynamically Add Log Messages with Log Points

Just select a line and fill in the log point form



At the very bottom of the interface, you can see the actual logged messages. This is simply a subset of Logs Explorer.

Agenda

Debugger

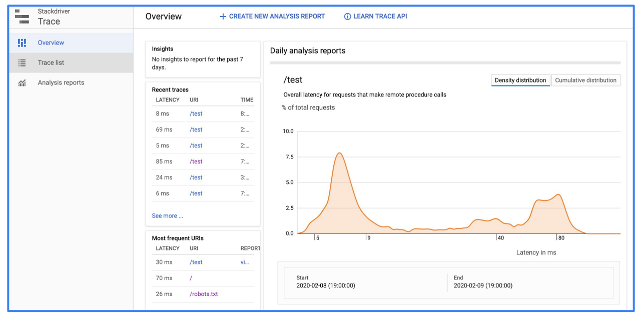
Trace

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



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.

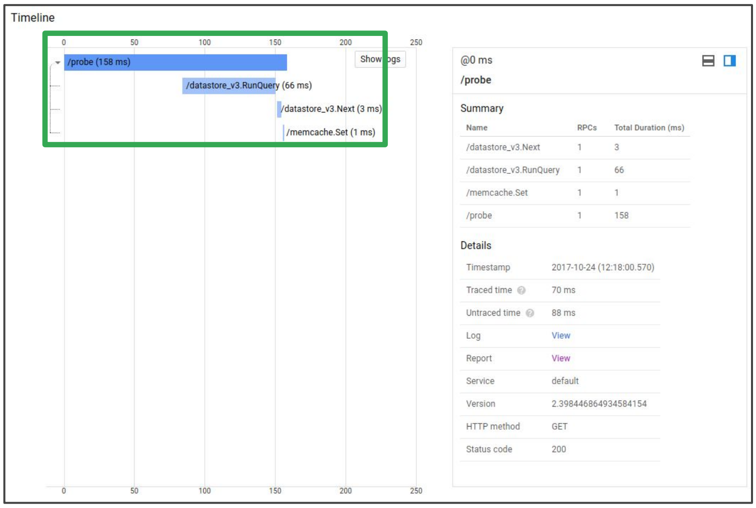
Trace Terminology

- 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



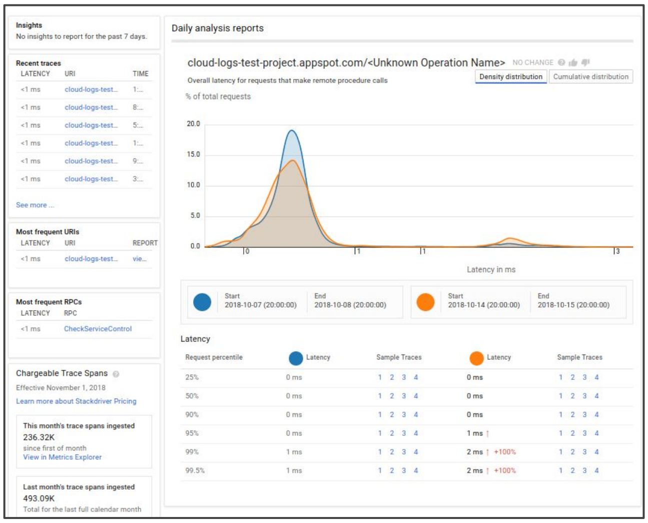
Here, we see an example trace created by a call to */probe*. It took the */probe* span a total of 158ms to handle the request and return a response.

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.

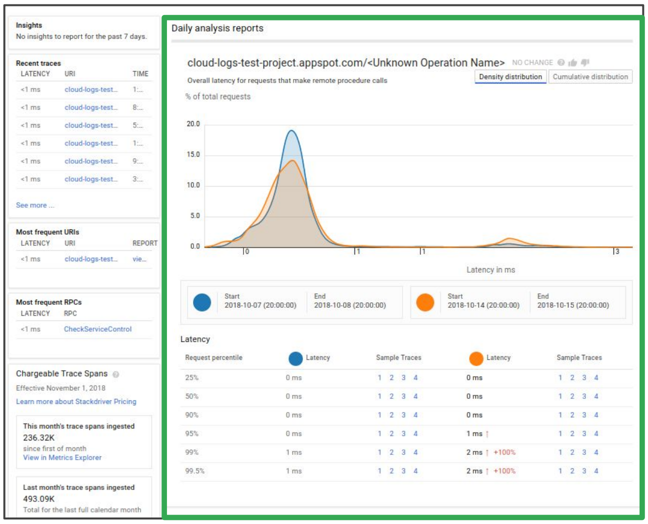
Trace overview



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.

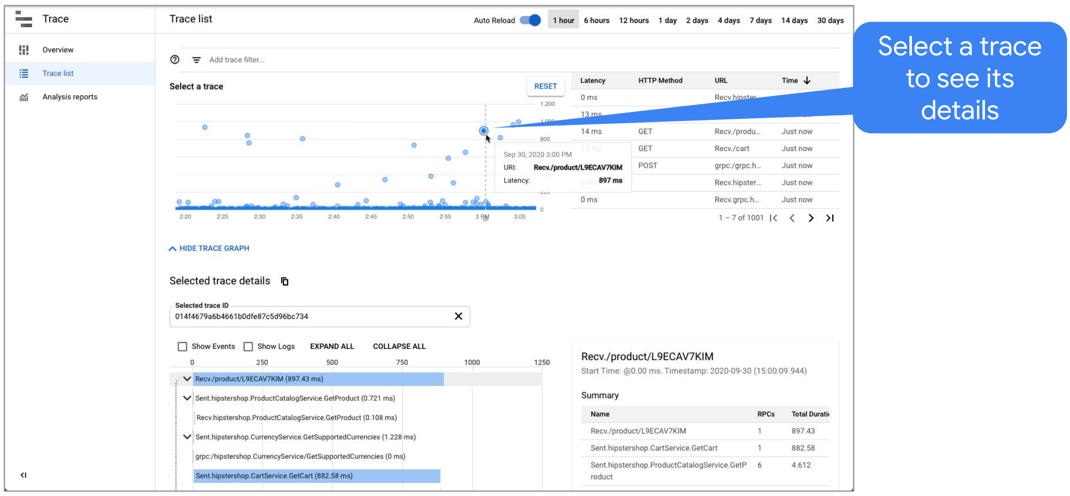
Trace Overview



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



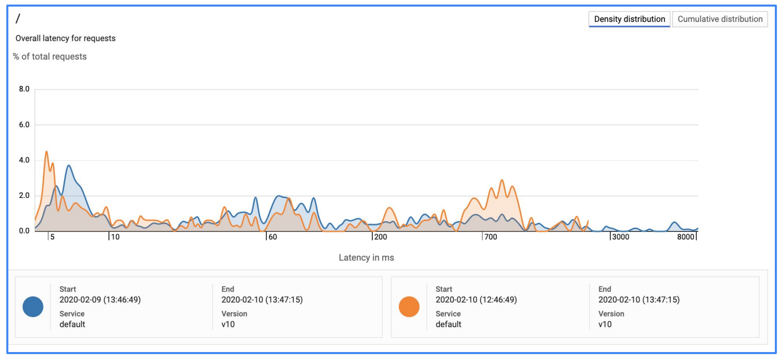
The **Trace list** window lets you find and examine individual traces in detail. Select a time interval, and you can view and inspect all spans for a trace, view summary information for a request, and view detailed information for each span in the trace from this window.

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.

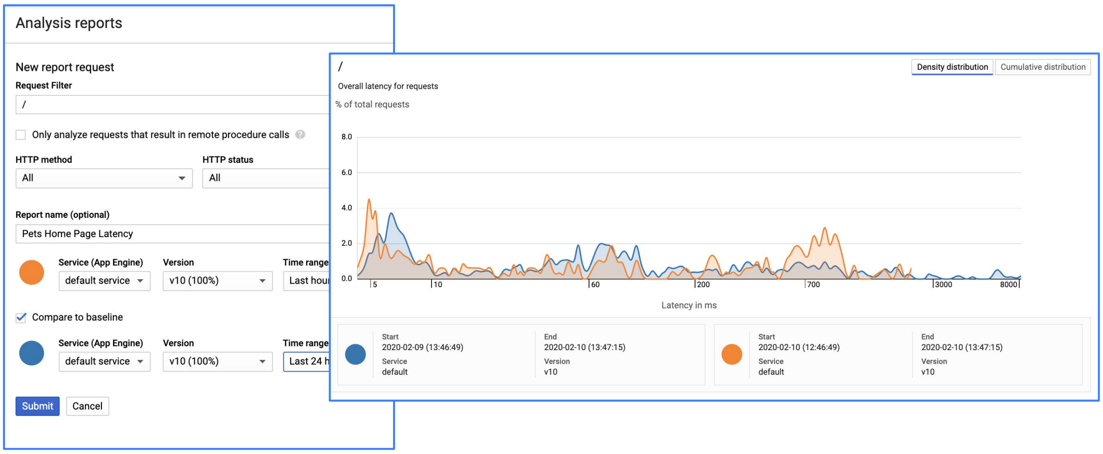
The recent-request table displays the 5 most recent requests and contains the last 1,000 traces.

Analysis Reports Show Requests Over Time and Compares Versions and Timespans



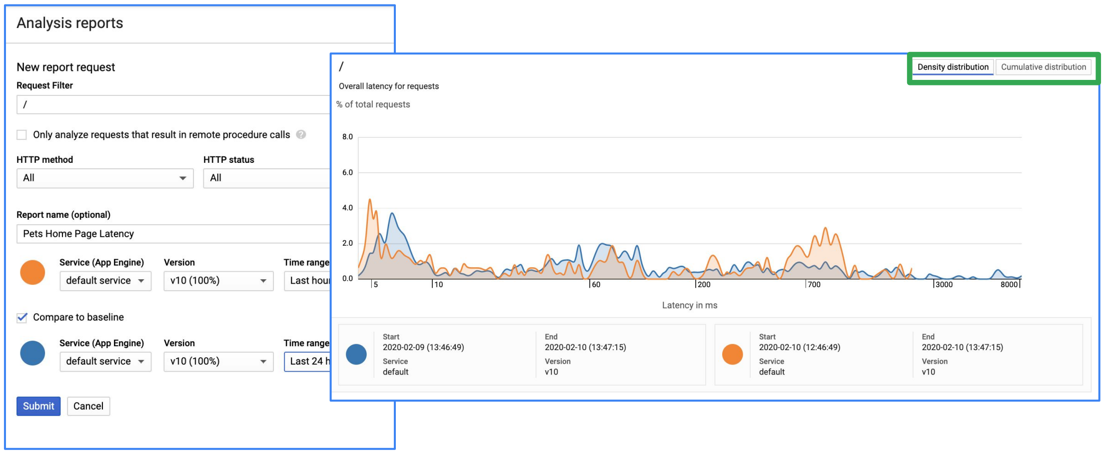
Analysis reports in Cloud Trace show you an overall view of the latency for all requests, or a subset of requests, to your application. This will be similar to the daily report viewed on the Trace Overview main page.

Analysis Reports Show Requests Over Time and Compares Versions and Timespans



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

Analysis Reports Show Requests Over Time and Compares Versions and Timespans



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

```python
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.

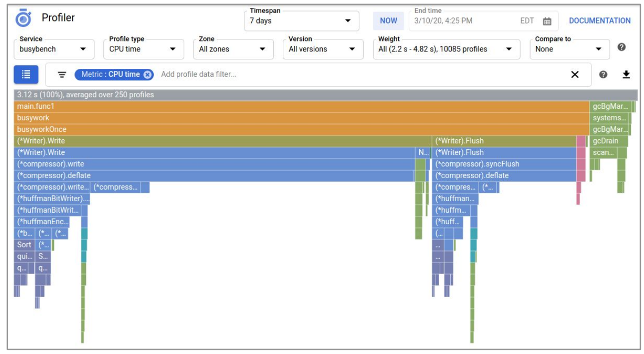
Here, we see an example from a Python Flask web application. When a GET request comes into *index.html*, we pull a reference to the tracer we created a couple of slides ago, and we use it to create a new span named 'index.'

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.

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



Google Cloud

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

```
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
```

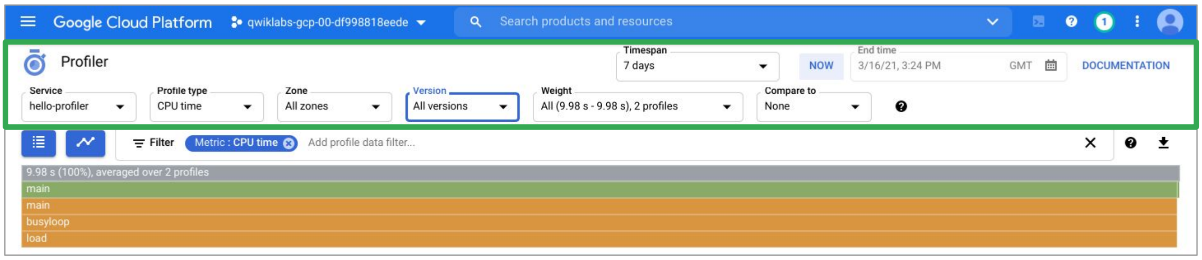
Like with Google's other application performance management products, the exact setup steps will vary by language, so [check the documentation](). Here, we are sticking with our Python application, which will be running on App Engine.

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

Start by importing the *googlecloudprofiler* package.

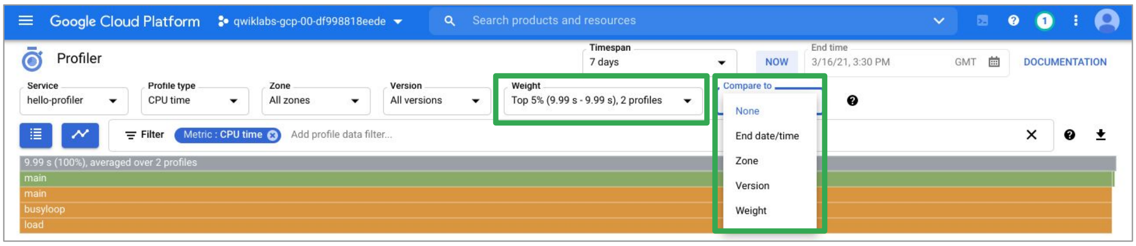
Then, early as possible in your code, start the profiler. In this example, we are setting the logging level (verbose) to 3, or debug level. That will log all messages. The default would be 0 or error only.

Select Profile to be Analyzed



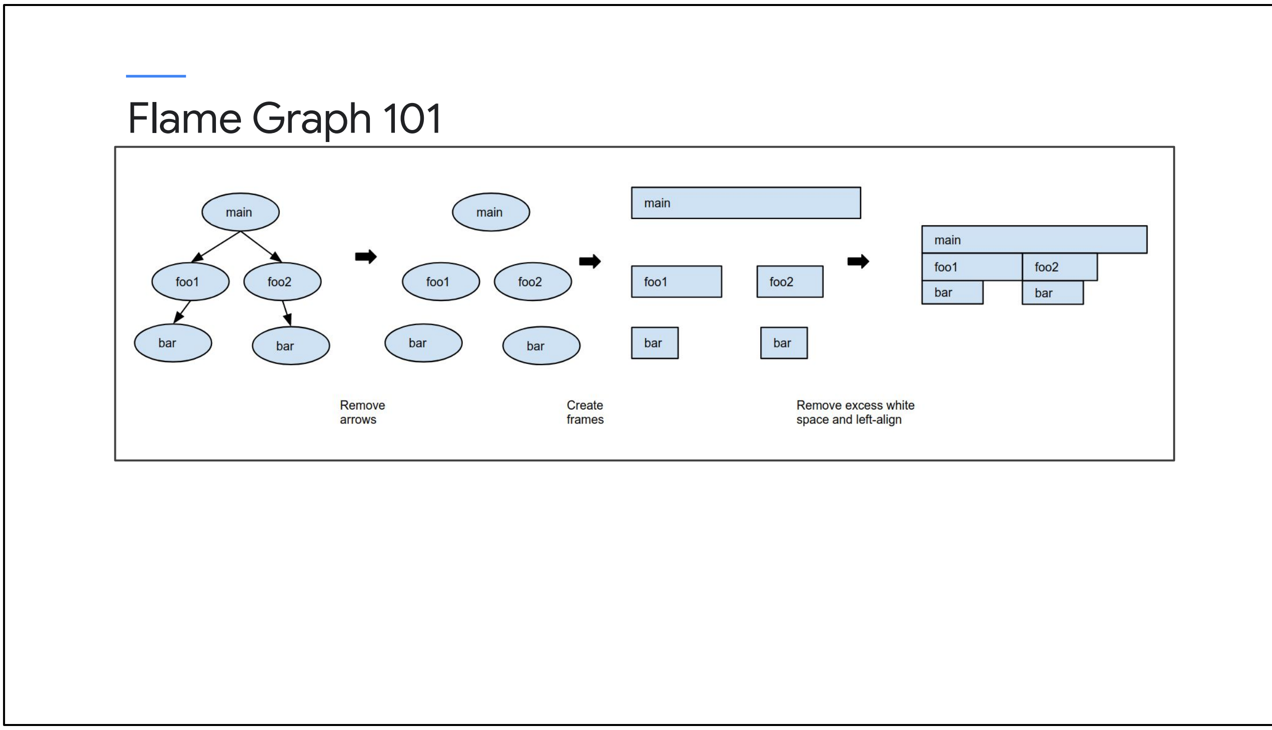
At the top of the Profiler interface, you can select the profile to be analyzed. You can select by timespan, service, profile type, zone, version, etc.

Select Profile to be Analyzed



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

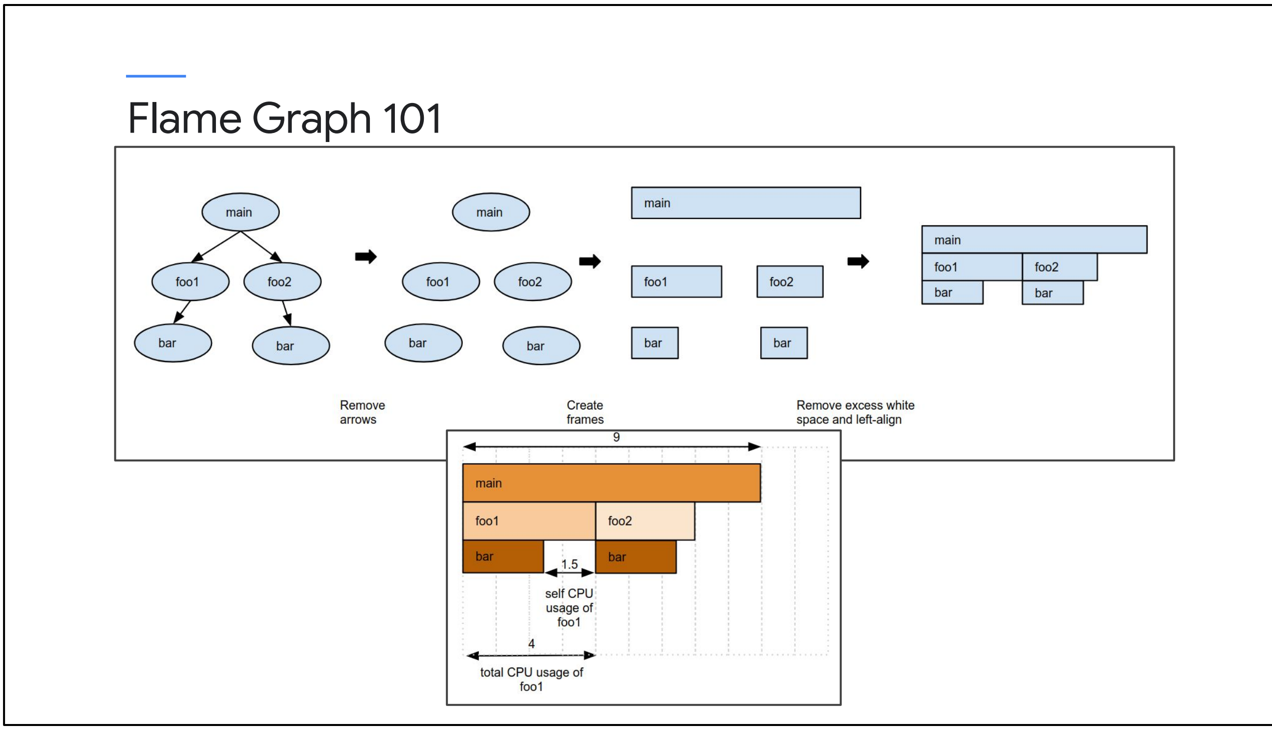
Compare to allows the comparison of two profiles.



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.

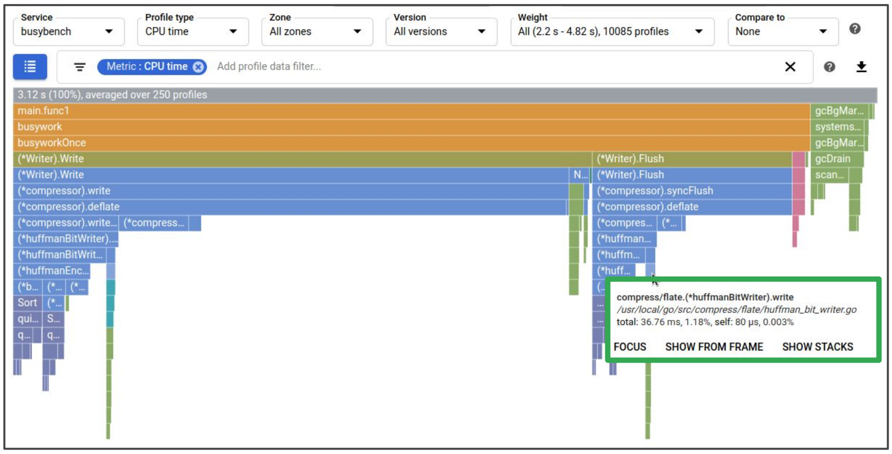
As you move through the graphic left to right, you can see how the information is collapsed. First, by removing arrows, then by creating frames, and finally by removing spaces and left aligning.



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

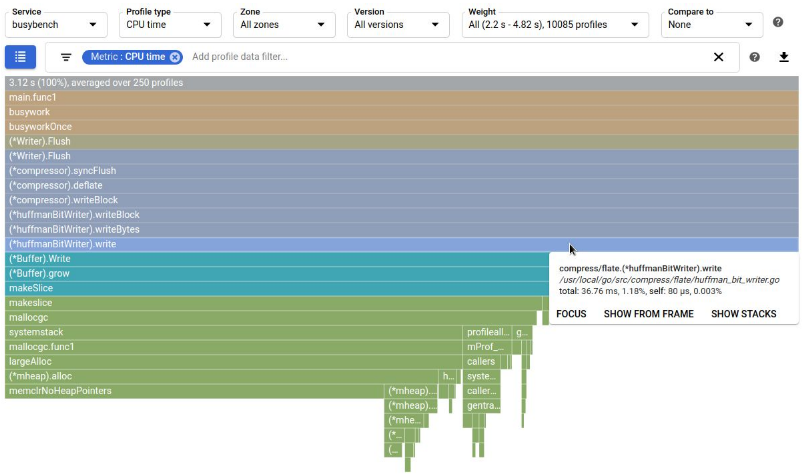


Here we have a full example.

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



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

In this lab, you will use the Google Cloud Debugger service to track down a bug in an application deployed to App Engine. You will see how to create breakpoints, inspect values, and dynamically add logging information to a running Google Cloud application.