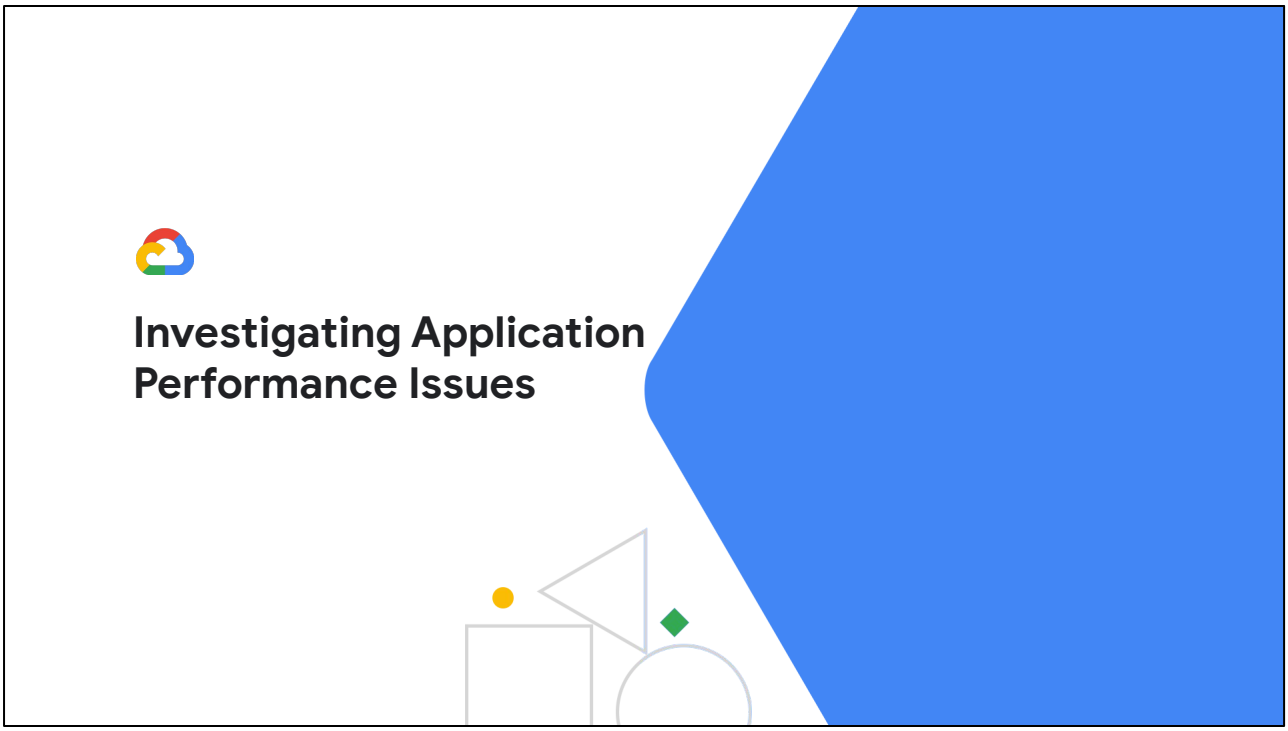


When deploying applications to Google Cloud, the Application Performance Management products (Cloud Trace, Cloud Debugger, and Cloud Profiler) provide a suite of tools to give insight into how your code and services are functioning, and to help troubleshoot where needed.

Agenda

Error Reporting

Debugger

Trace

Profiler

Google Cloud

In this module, you 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

Error Reporting

Debugger

Trace

Profiler

Google Cloud

Let’s get started.

Error Reporting

- Find and analyze code crashes in your cloud based services
- Centralized error management interface
- Supports Alerts for new errors
- Views of error details, time charts, occurrences, and stack trace
- Support for many popular languages
  - Node.js, Python, Java, .NET, PHP, Ruby, C#, and GO
  - API available

Google Cloud

Error Reporting counts, analyzes, and aggregates the crashes in your running cloud services.

It provides a centralized error management interface and displays the results with sorting and filtering capabilities.

And a dedicated view shows the error details: time chart, occurrences, affected user count, first- and last-seen dates, and a cleaned exception stack trace. Opt in to receive email and mobile alerts on new errors.

Support is available for many popular languages, including Node.js, Python, Java, .NET, PHP, Ruby, C#, and Go. Use our client libraries, REST API, or send errors with Cloud Logging.

Lecture Notes:
Error reporting is available for most languages, but not all

Error Reporting: https://cloud.google.com/error-reporting/docs
Setting up: https://cloud.google.com/error-reporting/docs/setup

In order to enable, you will need to load the error reporting API in your code, and configure it

Google Cloud Product Support

Error Reporting can aggregate and display errors for:

- App Engine standard and Flex
- Cloud Functions
- Apps Script
- Cloud Run
- Compute Engine
- Kubernetes Engine

Google Cloud

Error Reporting can aggregate and display errors for:

- App Engine standard and Flex.
- Cloud Functions.
- Apps Script.
- Cloud Run.
- Compute Engine.
- And Kubernetes Engine.

Error Reporting setup

- To report errors, code will need the Error Reporting Writer IAM role
- Details of Error Reporting setup are language and environment specific
  - Check the documentation

Note: A log message from any language or environment using ReportedErrorEvent formatting will be reported to Error Reporting

Google Cloud

First, to report errors, code will need the Error Reporting Writer IAM role.

Error Reporting is then available in many Google compute environments.

The details of setup/enabling Error Reporting depends on the language and environment.

A side node: A log message from any language or environment using ReportedErrorEvent formatting will be reported to Error Reporting automatically.

Setting up Error Reporting for an Express NodeJS app

```
// Import the GCP ErrorReporting library
const {ErrorReporting} = require('@google-cloud/error-reporting');

// Get ready to talk to the Error Reporting GCP Service
const errors = new ErrorReporting({
 reportMode: 'always' //as opposed to only while in production
});
```

Note: The full source for this example can be found on GitHub.

Google Cloud

Let's look at an error-catching example written in Node.js (JavaScript), and run on Google's Cloud Run.

The full source for this example can be found on GitHub and you'll also see it in the upcoming lab.

To manually log errors to Error Reporting in Node, the easiest way is to import the Error Reporting library.

Lecture Notes:
No easy way to enable Error Reporting unless you put it in code. These are all unhandled error. If your code handles it, it will not be sent to Error reporting unless explicitly sent.

Create ErrorReporting and configure to always report

```
// Import the GCP ErrorReporting library
const {ErrorReporting} = require('@google-cloud/error-reporting');

// Get ready to talk to the Error Reporting GCP Service
const errors = new ErrorReporting({
 reportMode: 'always' //as opposed to only while in production
});
```

Google Cloud

When creating the new ErrorReporting object, the **reportMode** configuration option is used to specify when errors are reported to the Error Reporting console.
It can have one of three values:

- **'production':** (default): Only report errors if the NODE_ENV environment variable is set to "production".
- **'always':** Always report errors regardless of the value of NODE_ENV.
- **'never':** Never report errors regardless of the value of NODE_ENV.

Set up a listener to catch all uncaught exceptions

```
//Setup a listener to catch all uncaught exceptions
process.on('uncaughtException', (e) => {
    // Write the error to stderr.
    console.error(e);
    // Report that same error the Error Service
    errors.report(e);
});
```

Google Cloud

If you'd like to catch any uncaught exceptions, add a listener to the *uncaughtException* event. Error Reporting doesn't do this automatically because it could conflict with a handler that you had already added into your code.

Since it's an Express app, add *errors* to middleware

```
// Note that express error handling middleware should be
// attached after all the other routes and use() calls.
app.use(errors.express);
```

```
const port = process.env.PORT || 8080;
app.listen(port, () => {
 console.log('Hello world listening on port', port);
});
```

Google Cloud

Since it's an Express app, you'll want to add the ErrorReporting middleware to make sure the errors all get sent to Error Reporting.

Here's an uncaught exception example

```
//Uncaught exception, auto reported
app.get('/uncaught', (req, res) => {
  doesNotExist();
  res.send("Broken now, come back later.")
});
```

Google Cloud

In languages like Node, exceptions are used to wrap error messages. Exceptions are a language way of saying, "Hey, something bad happened, and here are some details."

Exceptions can be caught and handled by code, or they can be bubbled out to the environment, as seen here.

The doesNotExist() function really doesn't exist

So calling it generates an error which will be reported to Error Reporting

```
//Uncaught exception, auto reported
app.get('/uncaught', (req, res) => {
 doesNotExist();
res.send("Broken now, come back later.")
});
```

Google Cloud

In this case, the *doesNotExist()* function literally isn't defined by the code.

As a result, calling it will generate an unhandled exception, which in turn, will generate a report in Error Reporting.

Manually log and report to Error Reporting

```
//Manually report an error
app.get('/error', (req, res) => {
  try{
    doesNotExist();
  }
  catch(e){
    //This is a log, will not show in Error Reporter
    console.error("Error processing /error " + e);
    //Let's manually pass it to Error Reporter
    errors.report("Error processing /error " + e);
  }
 res.send("Broken now, come back later.")
});
```

Google Cloud

Now, let's manually handle an error.

This example method also calls *doesNotExist*, but it handles the error itself by catching it, instead of just letting it bubble out to the environment.

The catch first logs the error to standard error

```
//Manually report an error
app.get('/error', (req, res) => {
  try{
    doesNotExist();
  }
  catch(e){
    //This is a log, will not show in Error Reporter
    console.error("Error processing /error " + e);
    //Let's manually pass it to Error Reporter
    errors.report("Error processing /error " + e);
  }
 res.send("Broken now, come back later.")
});
```

Google Cloud

The catch first logs the error to standard error. That will show up in the logs but not in Error Reporting.

Then it reports it to Error Reporting

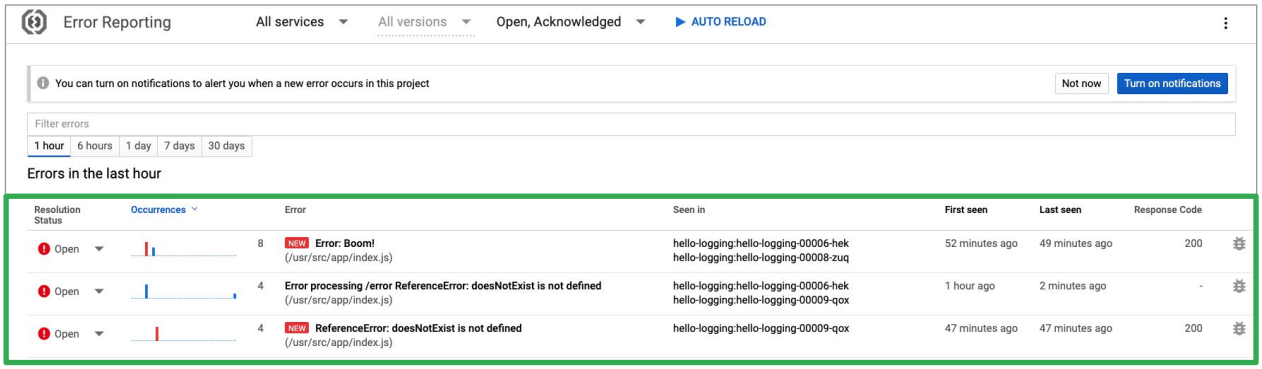
```
//Manually report an error
app.get('/error', (req, res) => {
  try{
    doesNotExist();
  }
  catch(e){
    //This is a log, will not show in Error Reporter
    console.error("Error processing /error " + e);
    //Let's manually pass it to Error Reporter
    errors.report("Error processing /error " + e);
  }
 res.send("Broken now, come back later.")
});
```

Google Cloud

Then, we manually report the error to Error Reporting using *errors.report*.

Make sure to check the GitHub repository for Node's Error Reporting library for syntax details.

Grouped List of Errors (Filtered)



Google Cloud

To see your errors, open the Error Reporting page in the Google Cloud Console.

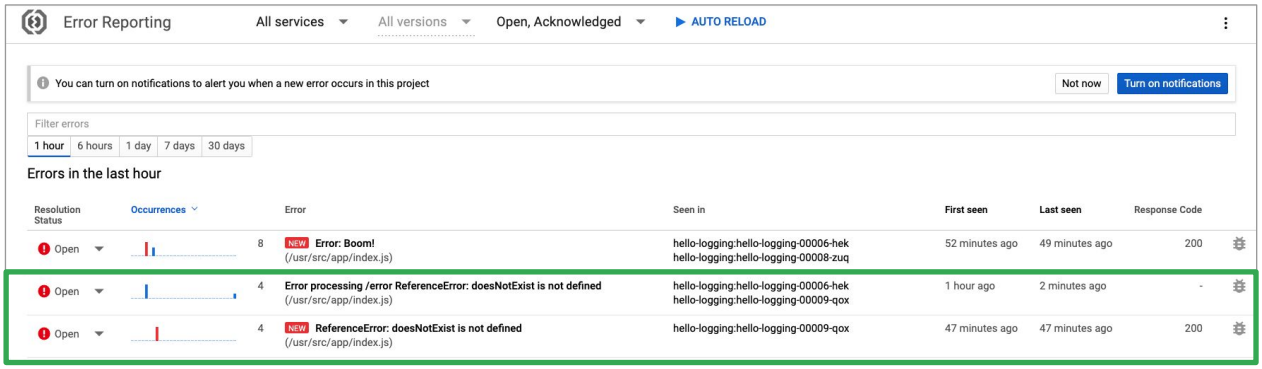
By default, Error Reporting will show you a list of recently occurring open and acknowledged errors, in order of frequency.

Errors are grouped and de-duplicated by analyzing their stack traces.

Error Reporting recognizes the common frameworks used for your language and groups errors accordingly.

Lecture Notes:
"Turn on Notification" from here is not very matured. Currently you can only send email to the project owner or some of the "Essential contacts" captured in IAM section.

Our two errors



Google Cloud

Here, you see the two groups of errors generated by our code. The first is from calls to /error, and the second is from our uncaught error event handler. The histogram is correlated to the time window. Since we are looking at errors in the last hour, its range covers that time.

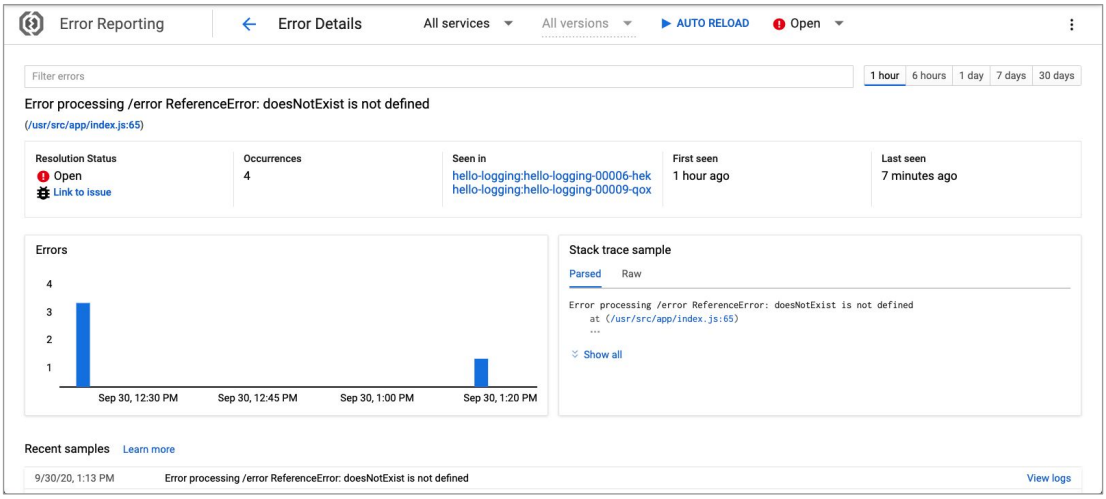
Error Reporting samples up to 1,000 errors per hour.

When this limit is reached, the displayed counts are estimated.

If too many events are received for the whole day, Error Reporting can sample up to 100 errors per hour and continue to extrapolate the counts.

You can filter, sort, and view additional details about errors, as well as restrict the errors that appear in the list to a specific time range.

Select an error for details

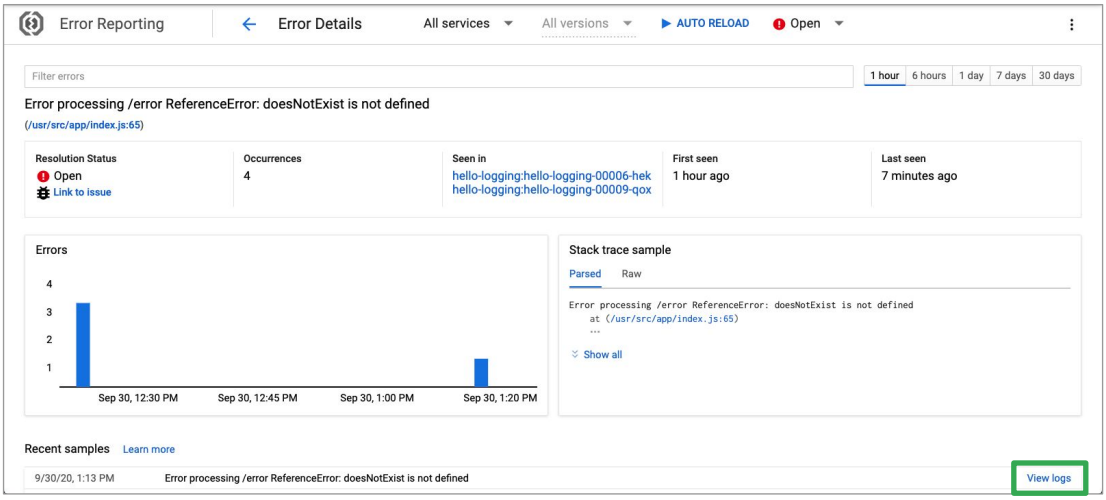


Google Cloud

Selecting an error entry will allow you to drill down into the **Error Details** page.

On this page, you can examine information about the error group, including the history of a specific error, specific error instances, and diagnostic information like stack traces.

Easily link to available logs



Google Cloud

To view the log entry associated with a sample error, click **View logs** from any entry in the **Recent samples** panel.

This takes you to the **Logs Viewer** in the Cloud Logging console.

Agenda

Error Reporting

Debugger

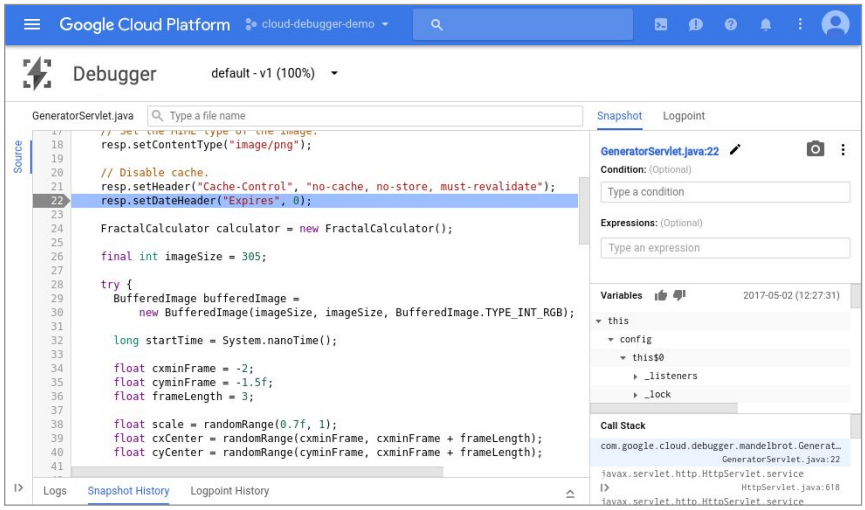
Trace

Profiler

Google Cloud

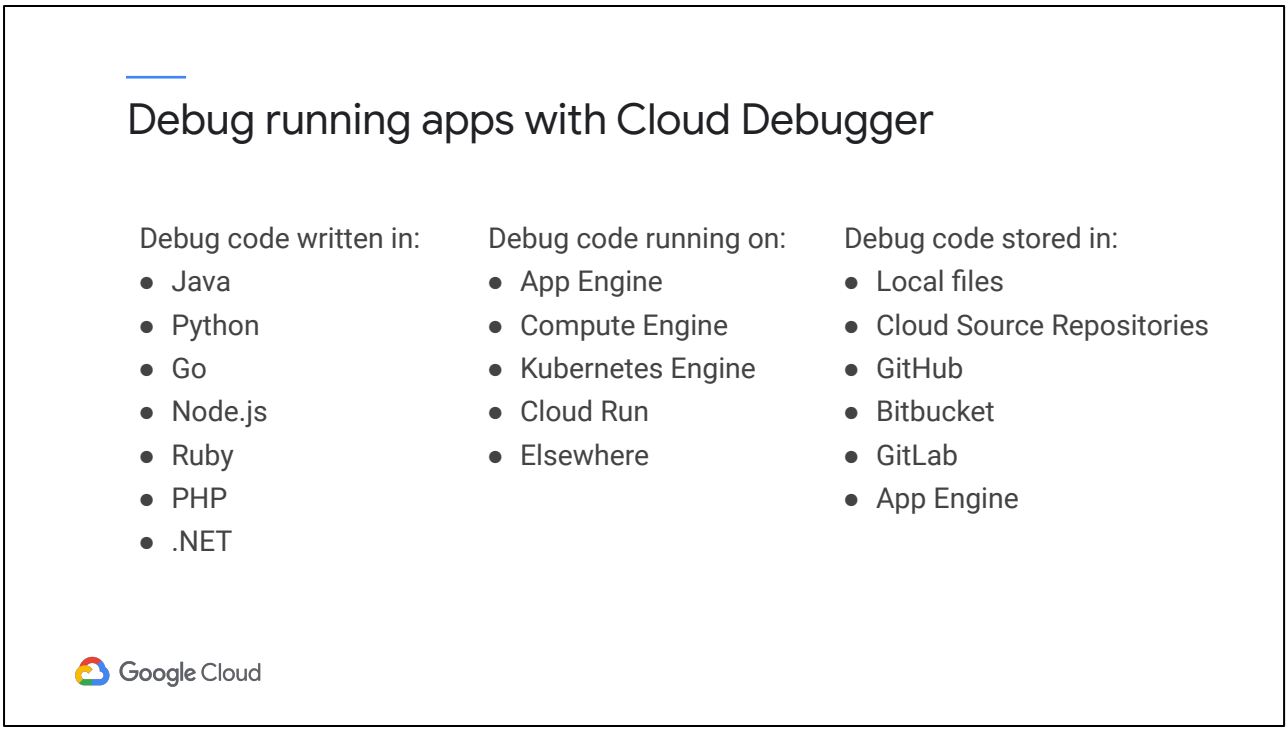
Let’s start with the debugger.

Google Cloud Debugger



Google Cloud

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.



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, and can pull it from the local file system, 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 on 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(
   breakpoint_enable)canary=False
  )
except ImportError:
  pass
```

Google Cloud

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

The Debugger agent for Python can use canary snapshots and logpoints every time you set a snapshot or logpoint. The agent canaries snapshots and logpoints to protect large jobs from any potential bug in the Debugger agent which can take the entire job down when a snapshot or a logpoint is applied.

To mitigate this, Debugger canaries snapshots and logpoints on a subset of your running instances each time they are set. After Debugger verifies the snapshot or logpoint does not adversely affect your running instances, Debugger then applies the snapshot or logpoint to all instances.

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

Google Cloud

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

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

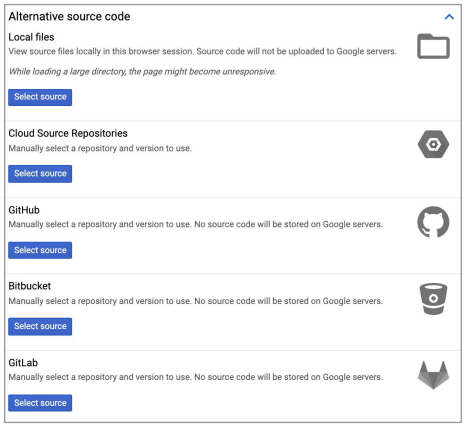
As a best practice, for Compute Engine VMs, Kubernetes Engine nodes, and external systems, create a service account with the Cloud Debugger Agent role, plus any other access the code will need, and make sure the code runs as that account.

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



Google Cloud

Debugger will also need access to your application's source code.

App Engine standard will select the source automatically.

App Engine Flex, Compute Engine, Kubernetes Engine, and Cloud Run can also select the source code automatically, but you'll have to provide a source context in the application root to support the feature.

You can generate the requisite *source-context.json* file using the gcloud command on this 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 hit, a snapshot of your running app is taken
  - You can then inspect variable values

Google Cloud

Unlike traditional debuggers, Cloud Debugger breakpoints do not stop code execution.

When the flow of execution passes the breakpoint, a nonintrusive snapshot is taken.

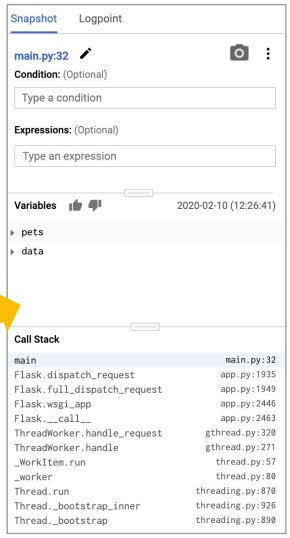
The snapshot captures local variables and the call stack state at that line, and you can inspect these at your leisure.

Lecture Notes:
Currently there are no breakpoints but snapshots and logpoints.
You can set conditions for when to take the snapshot (say when errorId = xxx)

Set breakpoints to debug from Snapshots

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

```
27          print('Pets Home Page Requested!')
28
29      if request.method == 'POST':
30          data = request.form.to_dict(flat=True)
31          pets = pet_db.search_pets(data['search'])
32          model = {"title": "My Pet's Great",
33                   "header": "Some Pets!", "pets": pets}
!           logpoint("Searched for {data['search']}")
34          print('Search Requested: ' + data['search'])
35
36      # Throm Randoms Errors approx 2% of the time
```



Google Cloud

In the example, a breakpoint was set at line 34.

When the execution flow passed the breakpoint, a snapshot was taken.

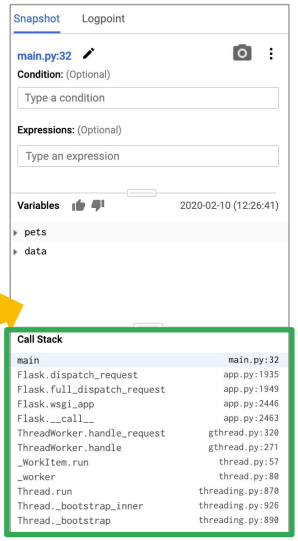
You can see the details in the snapshot to the right.

Lecture Notes:
Snapshot is taken only once, unless you manually retake.
If there are multiple snapshots, an indcator at the breakpoint shows up and you can see the history.

Set breakpoints to debug from Snapshots

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

```
27          print('Pets Home Page Requested!')
28
29      if request.method == 'POST':
30          data = request.form.to_dict(flat=True)
31          pets = pet_db.search_pets(data['search'])
32          model = {"title": "My Pet's Great",
33                   "header": "Some Pets!", "pets": pets}
!           logpoint("Searched for {data['search']}")
34          print('Search Requested: ' + data['search'])
35
36      # Throm Randoms Errors approx 2% of the time
```



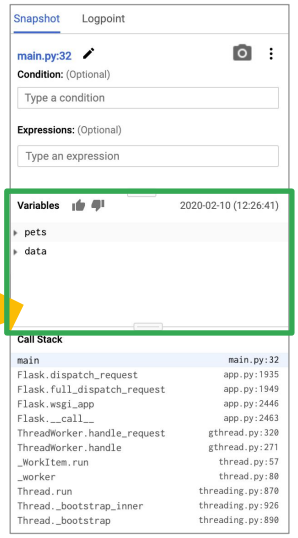
Google Cloud

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 hit, a snapshot of your running app is taken
  - You can then inspect variable values

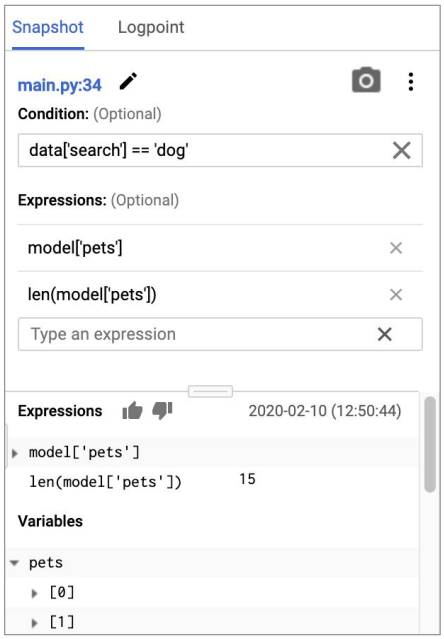
```
27          print('Pets Home Page Requested!')
28
29      if request.method == 'POST':
30          data = request.form.to_dict(flat=True)
31          pets = pet_db.search_pets(data['search'])
32          model = {"title": "My Pet's Great",
33                   "header": "Some Pets!", "pets": pets}
!           logpoint("Searched for {data['search']}")
34          print('Search Requested: ' + data['search'])
35
36      # Throm Randoms Errors approx 2% of the time
```



Google Cloud

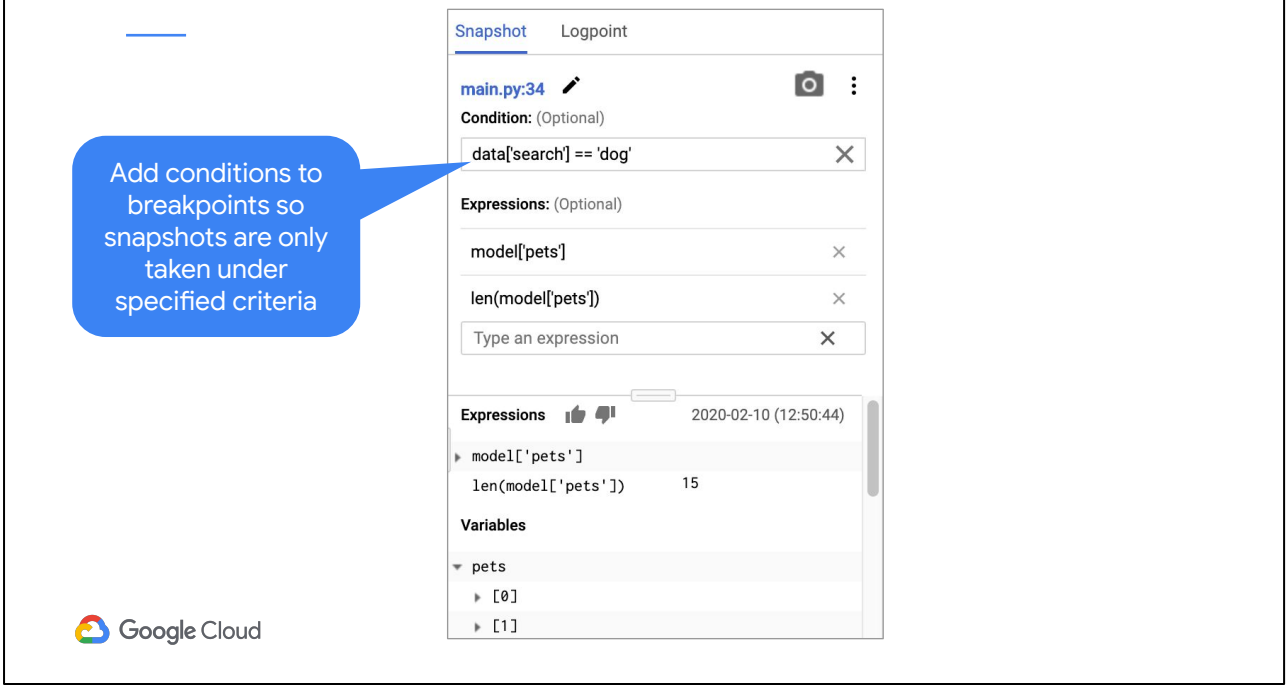
Above that, you could expand *pets* or *data* to see what those variables contained at that moment in time.

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

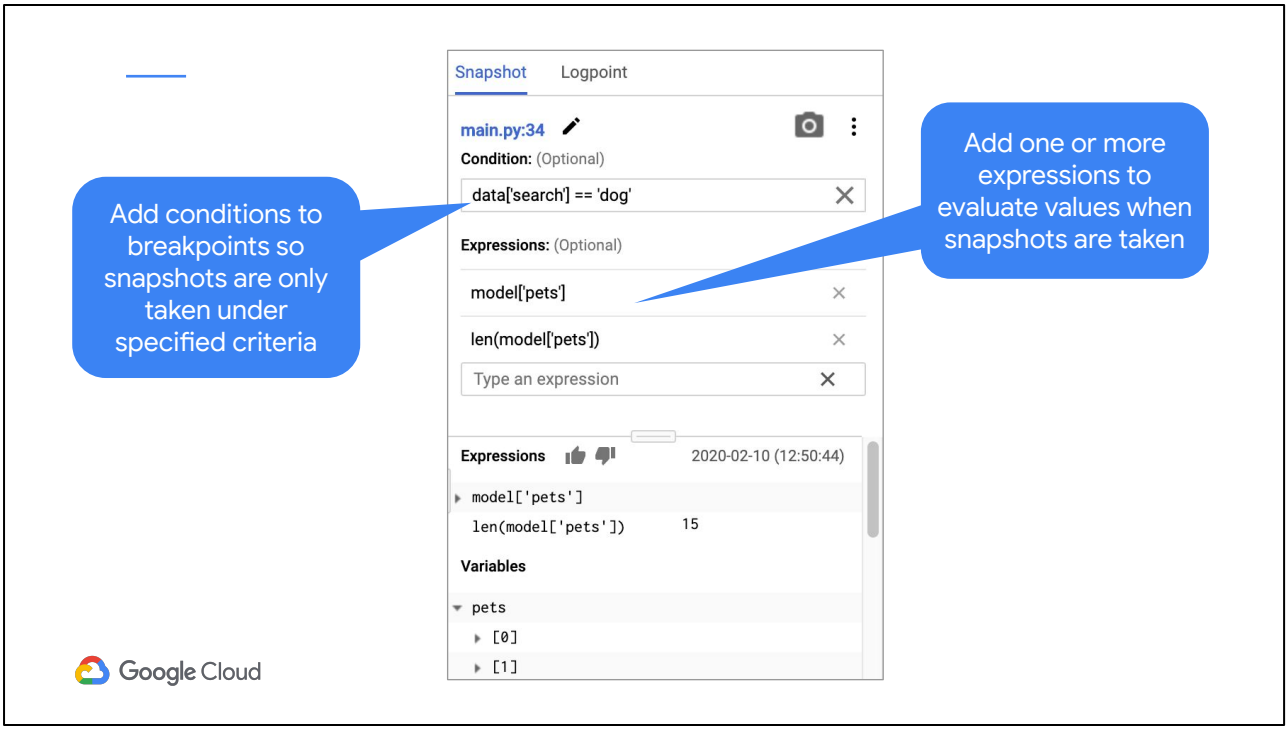


Google Cloud

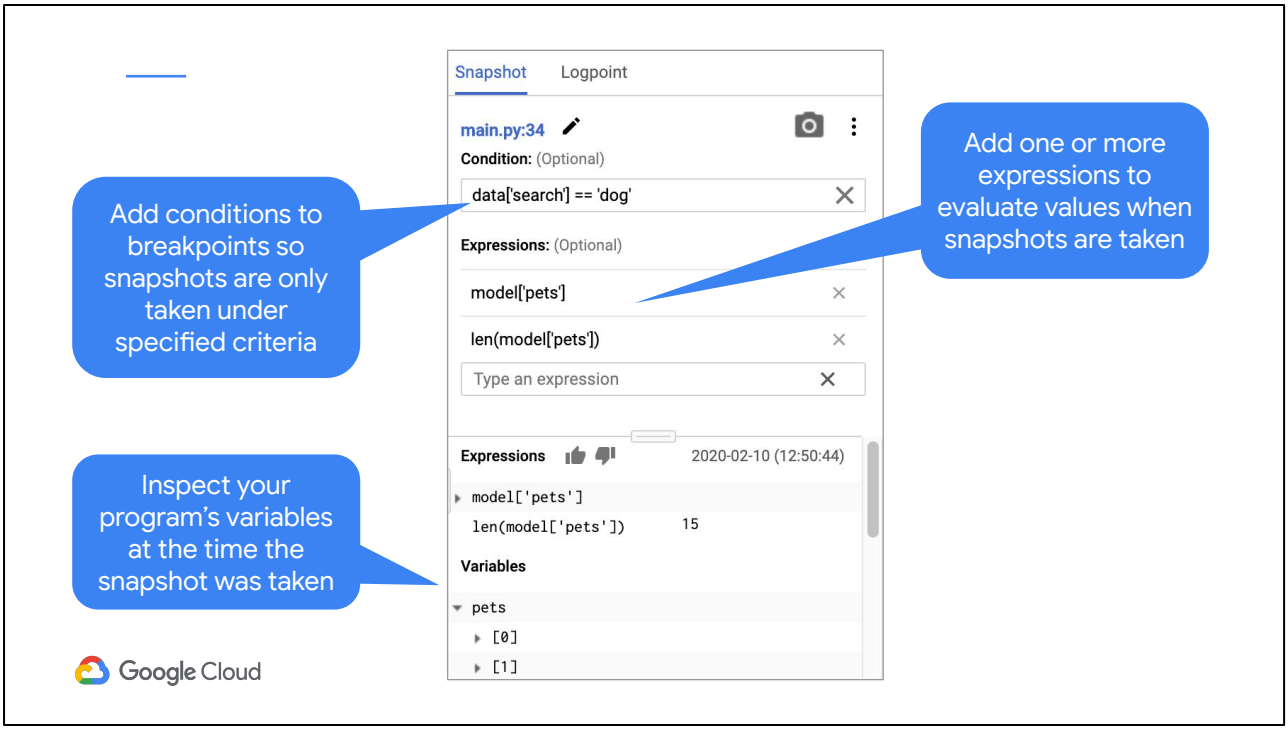
Here, you see a closeup of the previous slide's line 34 snapshot.



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."

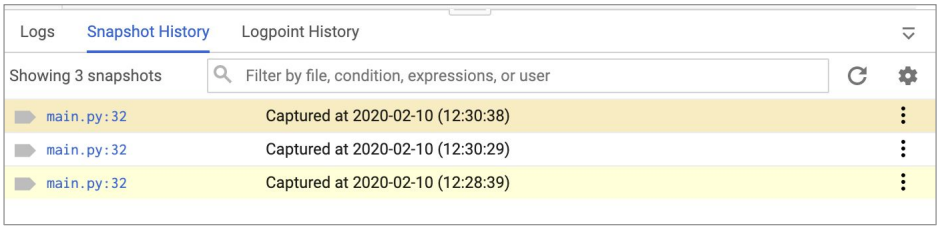


Below that, notice the expressions that have been created. Expressions will be evaluated when the snapshot is taken, and their values appear below. You might use an expression to access a variable that's out of the local set (global or static), or to simplify deep navigation.



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.

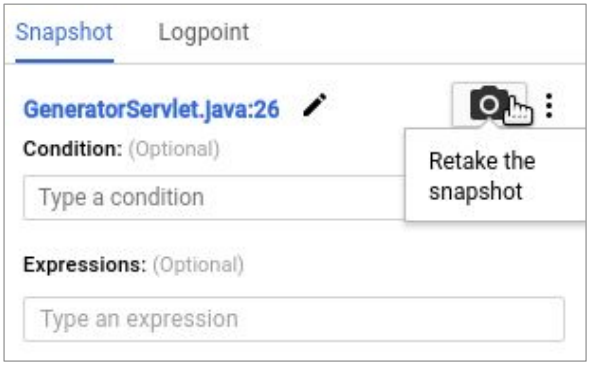
Snapshots history is available



Google Cloud

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.

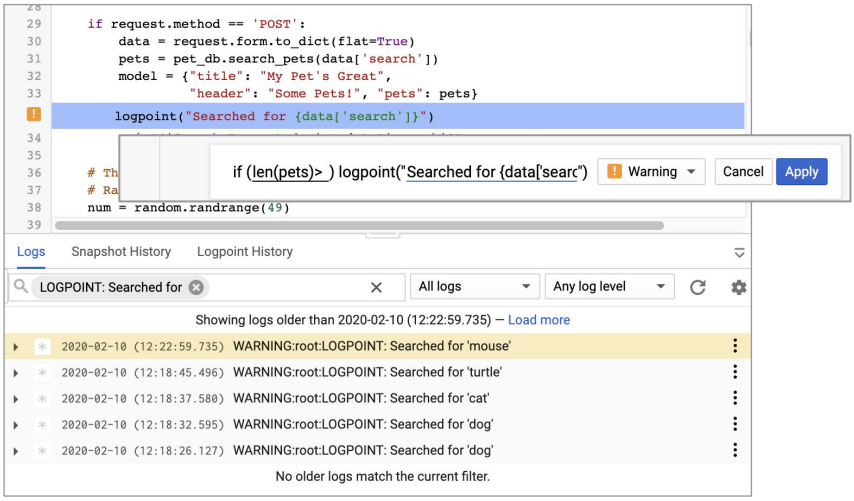
Retake a Snapshot



Google Cloud

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



Google Cloud

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

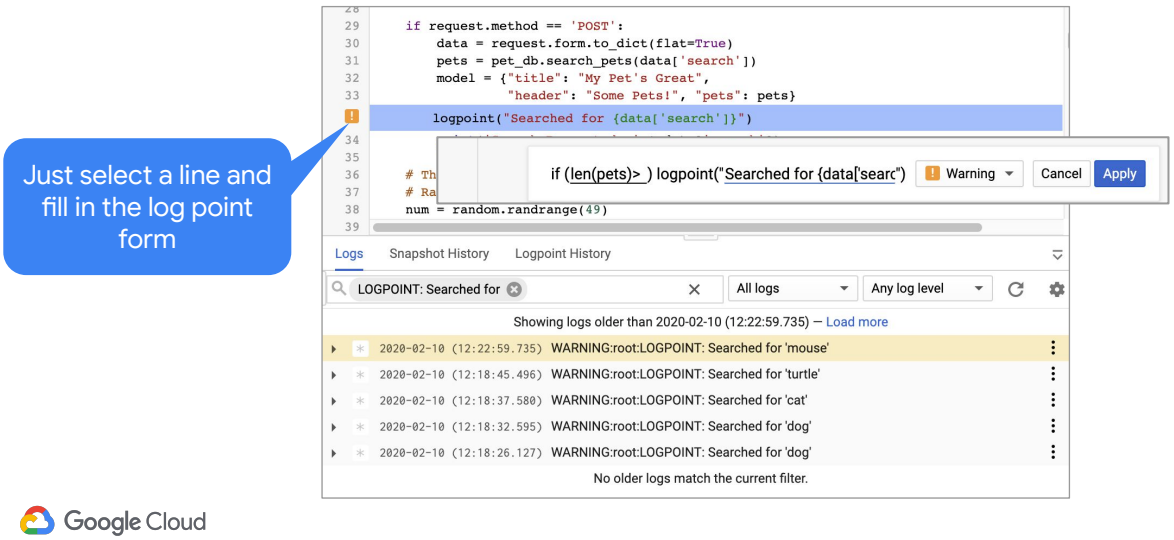
"Made it this far."

"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 redeploy the application.

Lecture Notes:
This gives you ability to add logs temporarily to output something while your code is running.
You need not update code and redeploy.
Auto deleted after 24 hours
Unlike snapshots, logpoint logs a message for every single visit

Dynamically add log messages with log points



Google Cloud

Logpoints allow you to inject logging into running services without editing, restarting, or interfering with the normal function of the service.

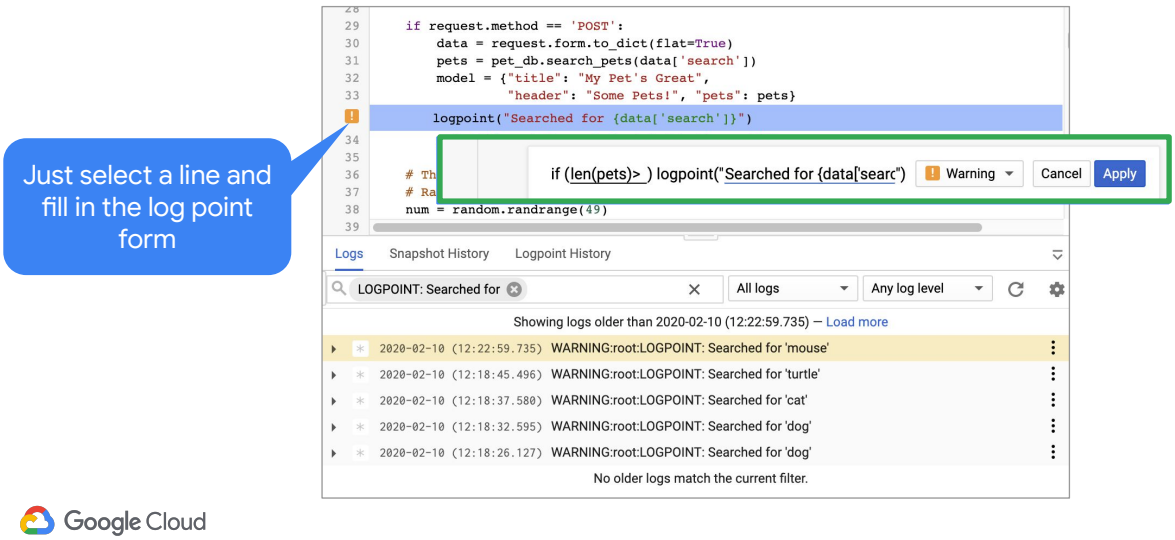
Every time any instance executes code at the logpoint location, Cloud Debugger logs a message.

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

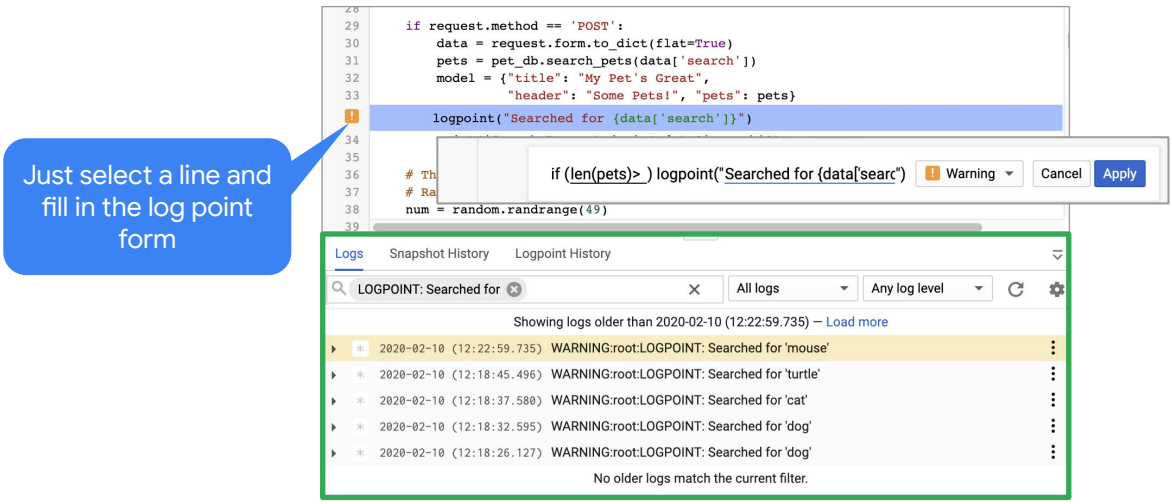


Google Cloud

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



Google Cloud

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

Agenda

Error Reporting

Debugger

Trace

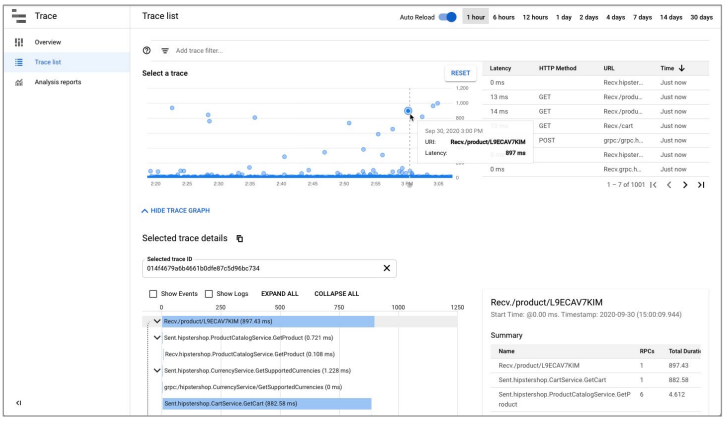
Profiler

Google Cloud

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



Google Cloud

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.

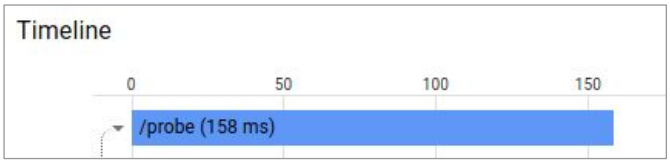
Cloud Trace automatically analyzes all of your application's traces to generate in-depth latency reports to surface performance degradations, and can capture traces from all of your VMs, containers, or App Engine projects.

Trace supports several mainstream languages.

Lecture Notes:
By default Trace tracks RPC latency (call from 1 component to another)

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

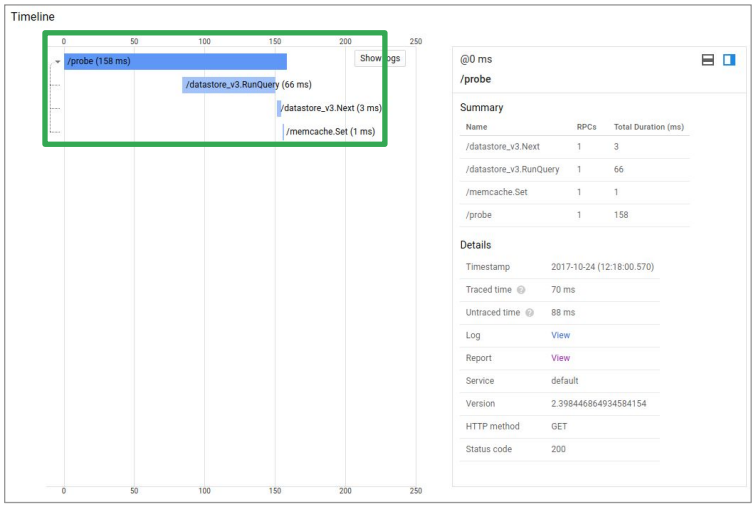


Google Cloud

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



Google Cloud

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.

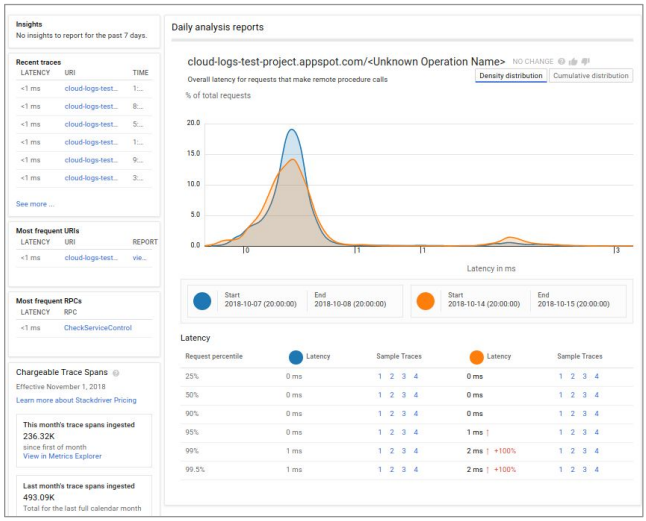
But */probe* didn't do all the work itself. */probe* called a method to RunQuery in Datastore, which took 66ms. Then */probe* took that result and called next on it, which took 3ms.

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

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

Lecture Notes:
One problem with tracing is it is for RPC only (when my code is calling some other services). Does not fit monoliths very well. Or microservices that do a bunch of things internally, will not be very successful with traceability
In the latter case, you can update your code and introduce manual trace span start and ends. for e.g. slide #52

Trace overview



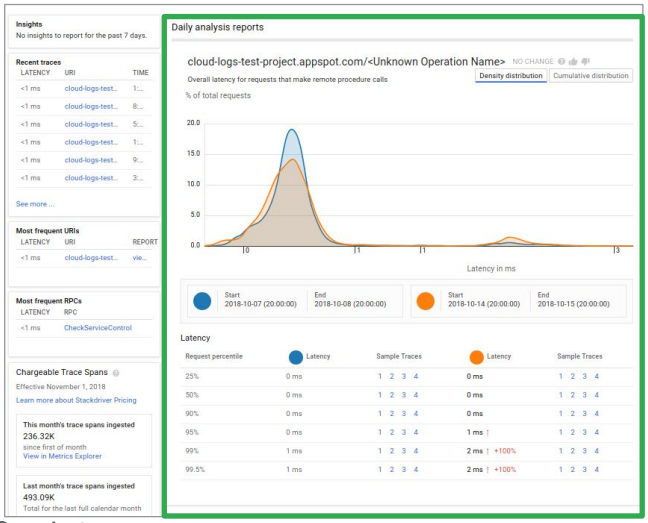
Google Cloud

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

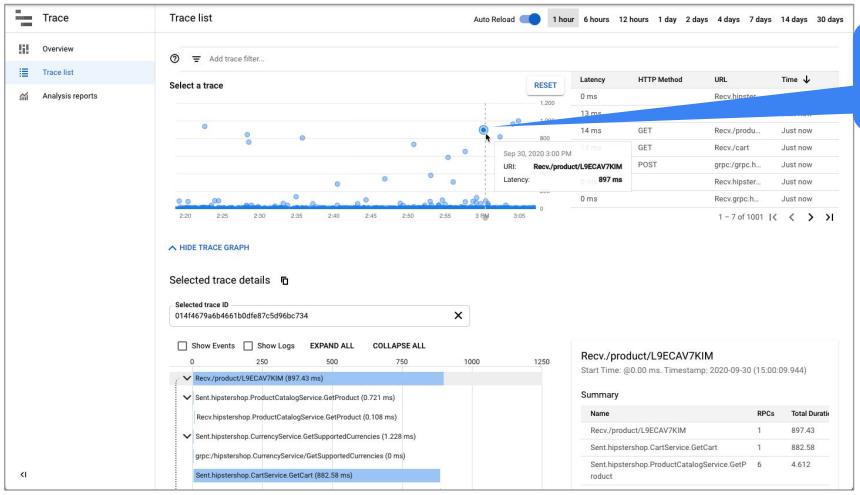


Google Cloud

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



Google Cloud

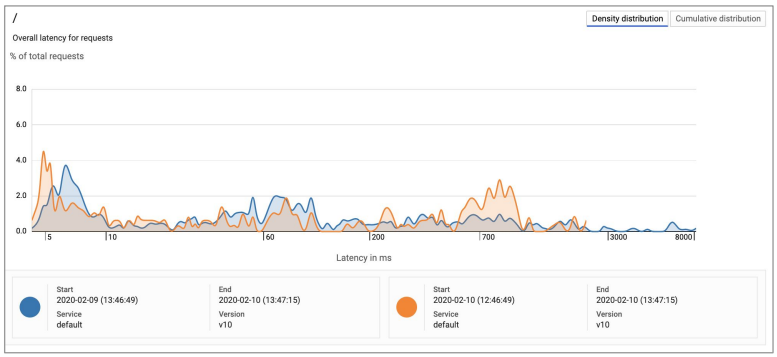
The **Trace list** window lets you find and examine individual traces in detail. Select a time interval, and you can view and inspect all spans for a trace, view summary information for a request, and view detailed information for each span in the trace from this window.

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.

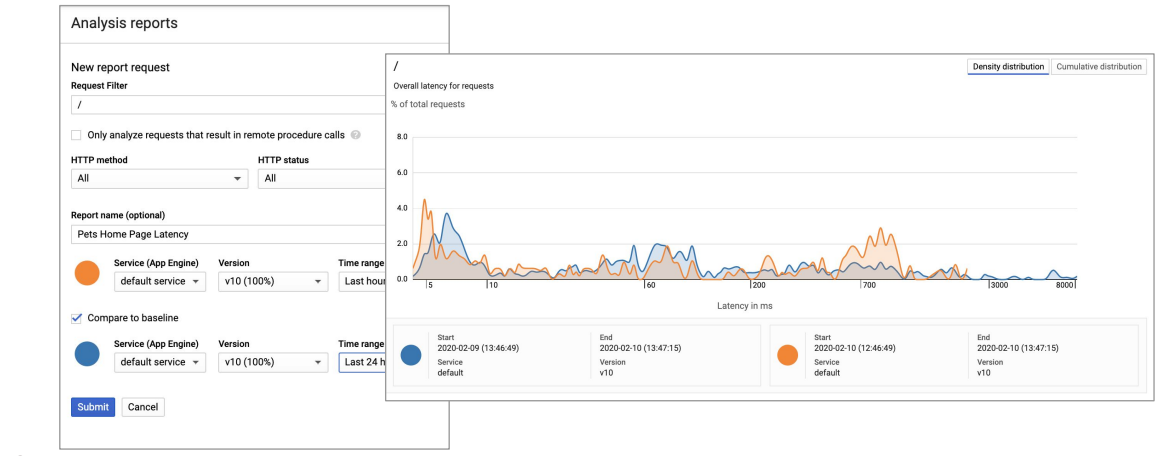
Analysis reports show requests over time and compares versions and timespans



Google Cloud

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



Google Cloud

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

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

Google Cloud

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

Google Cloud

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

App Engine, Cloud Run, 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

```
@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

Google Cloud

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.

Lecture Notes:
Some languages like Java allows you to inject aspects (aspect oriented programming) to start/end tracer spans instead of having to do it individually for each function.

Agenda

Error Reporting

Debugger

Trace

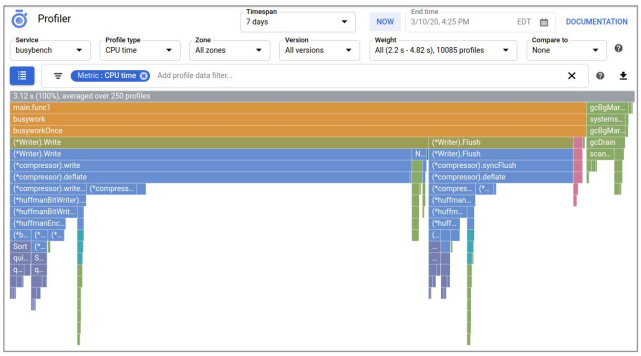
Profiler

Google Cloud

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.

Lecture Notes:
Memory profiling on-prem is the easier option to go and more mature
Very limited support for profiler in GCP today.

Profiler: https://cloud.google.com/profiler/docs
About Profiler: https://cloud.google.com/profiler/docs/about-profiler
Reading Flame graphs: https://cloud.google.com/profiler/docs/concepts-flame

In profiler, (flame chart) we are looking for gaps. The gap means the code above is taking time on its own

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

Google Cloud

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

Google Cloud

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

Google Cloud

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

Google Cloud

And for Threads you have:

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

Supported compute technologies

Environment	Go	Java	Node.js	Python
Compute Engine	Y	Y	Y	Y
GKE	Y	Y	Y	Y
App Engine (Std + Flex)	Y	Y	Y	Y
Dataproc		Y		
Outside Google Cloud	Y	Y	Y	Y

Profiler will instrument applications running in most Google and non-Google compute technologies.

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
```

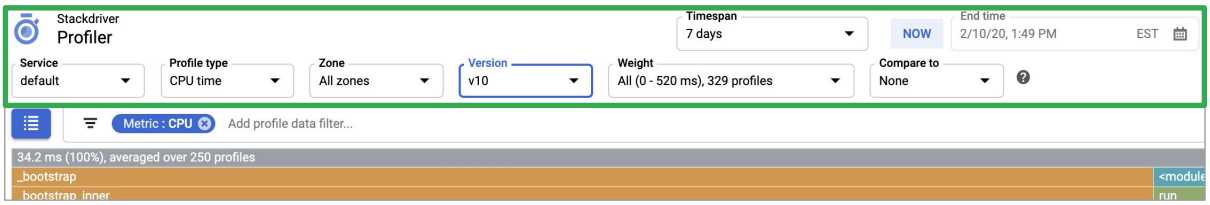
Like with Google's other application performance management products, the exact setup steps will vary by language, so check the documentation. Here, we are sticking with our Python application, which will be running on App Engine.

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



Google Cloud

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.

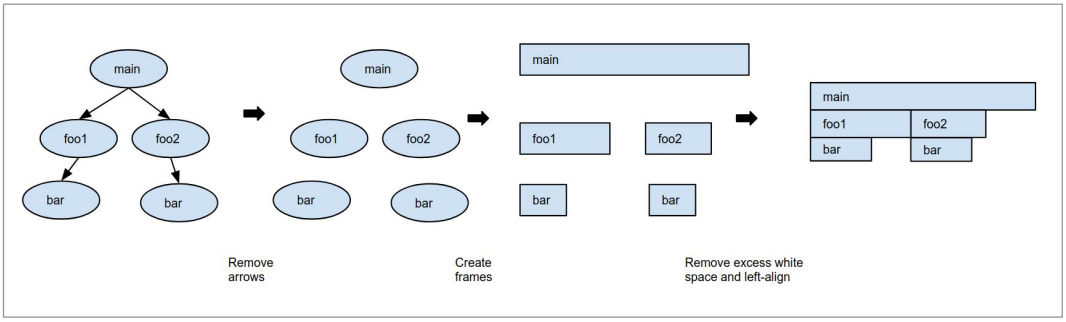
Select profile to be analyzed

Google Cloud

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



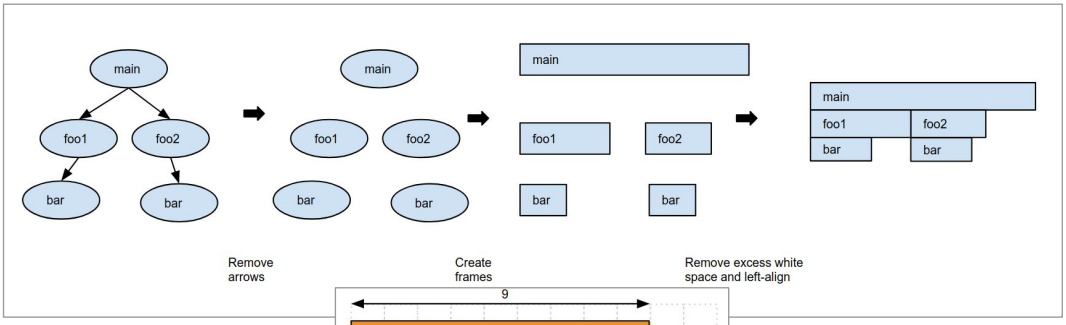
Google Cloud

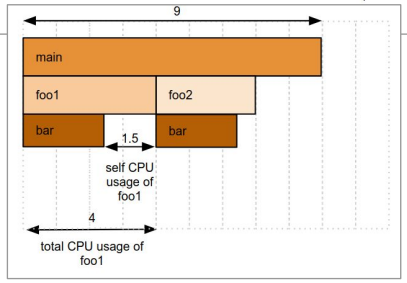
Cloud Profiler displays profiling data by using Flame Graphs. Unlike trees and standard graphs, flame graphs make efficient use of screen space by representing a large amount of information in a compact and readable format.

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





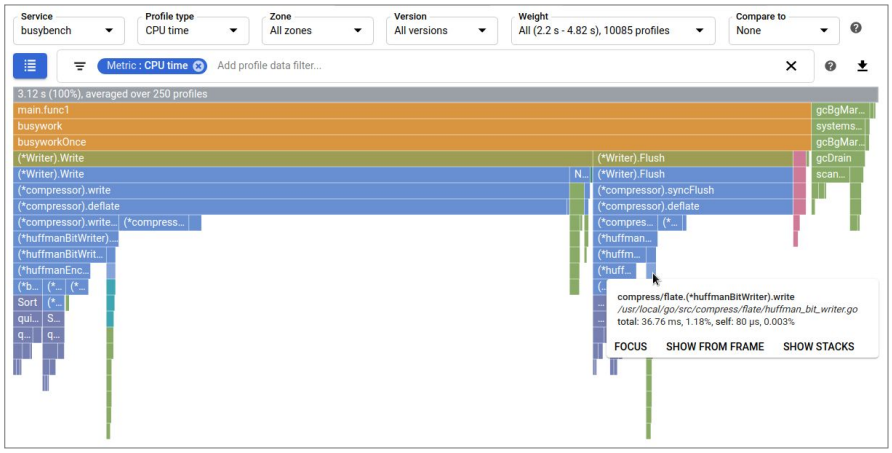
Google Cloud

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



Google Cloud

Here we have a full example.

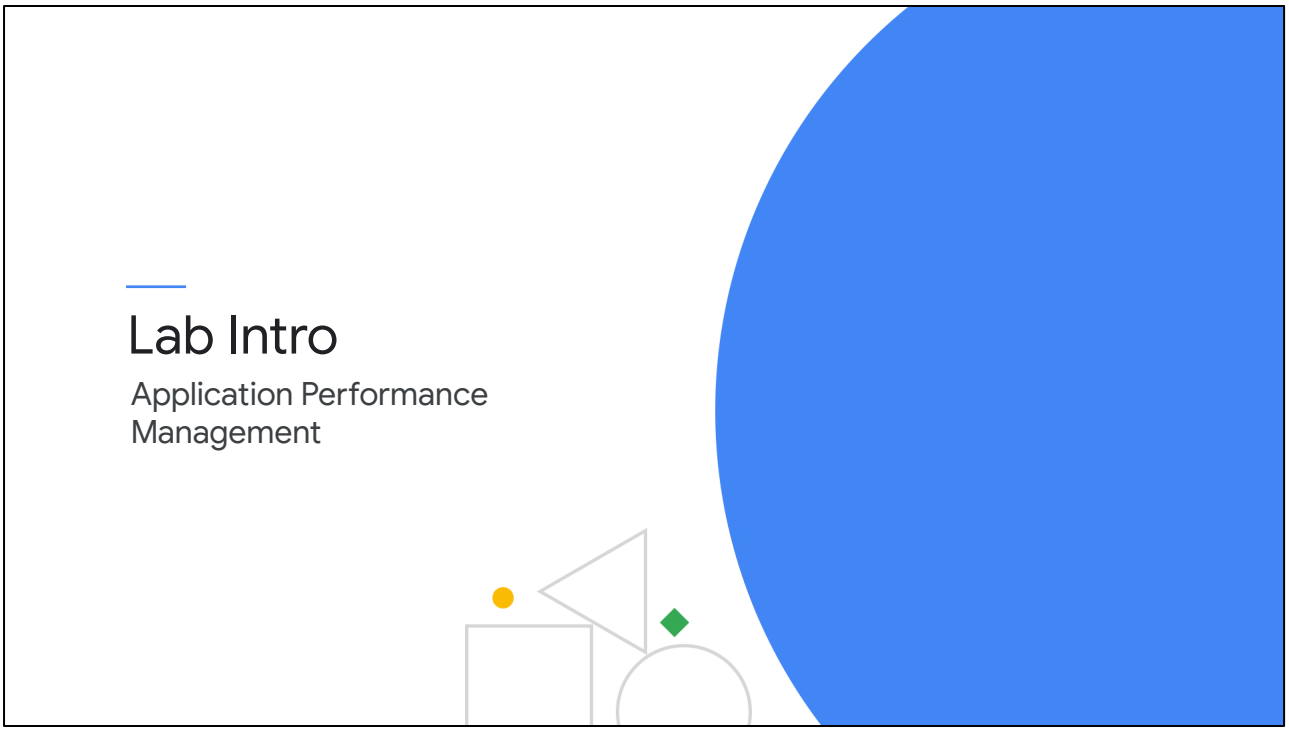
When you hold the pointer over a frame, a tool tip opens and displays additional information including:

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

Select a frame to zoom

Google Cloud

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.

Quiz

Your App Engine application crashes sometimes and you don't know why. It works on your development machine. Which tool would most likely help you figure out the problem?

A. Profiler
B. Trace
C. Debugger
D. Logging

Quiz

Your App Engine application crashes sometimes and you don't know why. It works on your development machine. Which tool would most likely help you figure out the problem?

A. Profiler
B. Trace
C. Debugger
D. Logging

Quiz

You have an SLO that states 90% of your http requests need to respond in under 100 ms. You'd like a report that compares latency for your last two versions. What tool would you use to most easily create this report?

A. Profiler
B. Trace
C. Debugger
D. Logging

Quiz

You have an SLO that states 90% of your http requests need to respond in under 100 ms. You'd like a report that compares latency for your last two versions. What tool would you use to most easily create this report?

A. Profiler
B. Trace
C. Debugger
D. Logging

Quiz

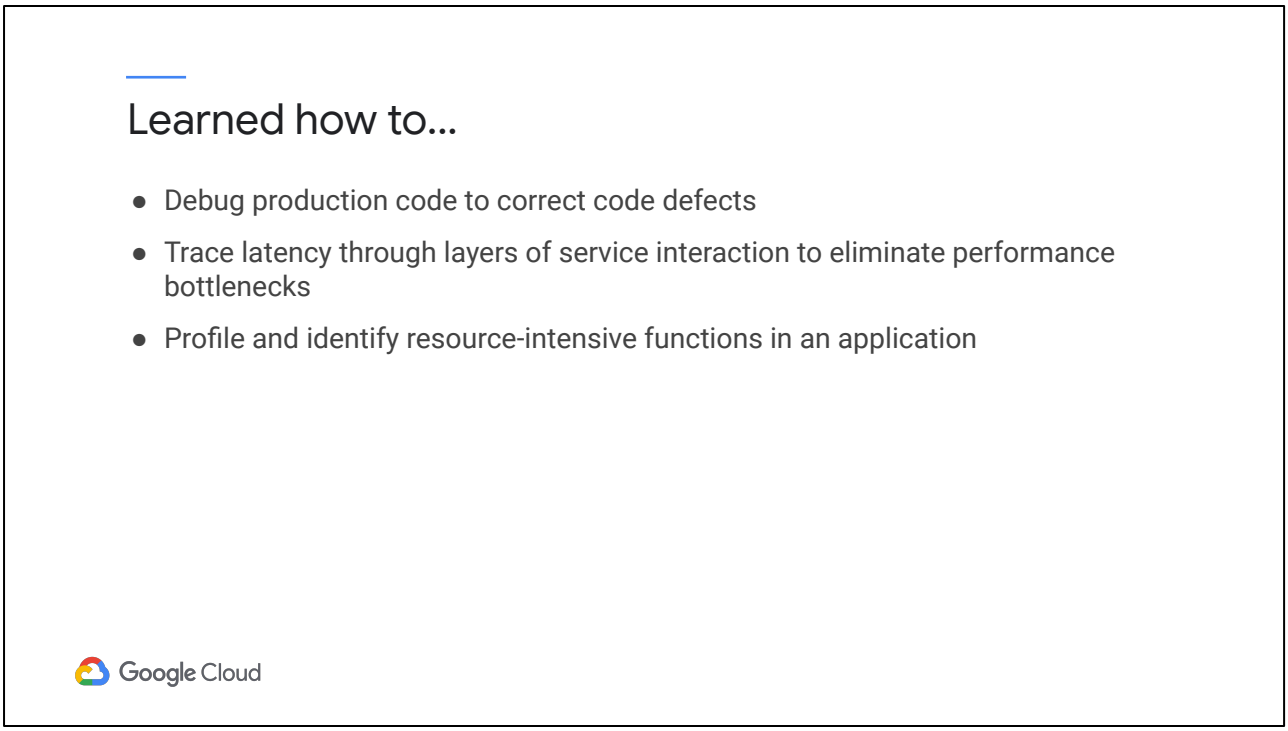
You've deployed a new version of a service and all of a sudden significantly more instances are being created in your Kubernetes cluster. You service scales when average CPU utilization is greater than 70%. What tool would help you investigate the problem?

A. Profiler
B. Trace
C. Debugger
D. Logging

Quiz

You've deployed a new version of a service and all of a sudden significantly more instances are being created in your Kubernetes cluster. You service scales when average CPU utilization is greater than 70%. What tool would help you investigate the problem?

A. Profiler
B. Trace
C. Debugger
D. Logging



Good job. Another module done.

In this module, you learned how to:

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

Fantastic work.

Google Cloud