This page provides some general background on profiling, on the types of profiling available with Stackdriver Profiler, and information about them.

Profiling is a form of dynamic code analysis. That is, profiling lets you capture characteristics of the code *as it runs* (that's the *dynamic* aspect). It lets you look at the actual resource consumption or performance traits of the program.

Profiling your code during development and testing can help you optimize the design of the code and find bugs, reducing the risk of catastrophic failures in production.

Profiling production code can help you anticipate when future problems might arise and help diagnose problems that do occur.

Unlike static code analysis, which examines the source code of the application rather than the running application, profiling puts an additional load on the program as it collects statistics about the running code. In addition, profiling needs a way to collect and retrieve these statistics from the execution environment. This additional work adds load to the program.

There are many ways to collect profiling data, and many ways to try to minimize the additional load on the running application. These typically involve tradeoffs between the accuracy of the profiled characteristics and the drag on the running application.

Stackdriver Profiler is a statistical, or sampling, profiler. It does not require pervasive changes to the program code to collect data. Instead, a piece of code, called the *profiling agent*, is essentially attached to the code, where it can periodically look at the call stack of the program to collect information about, for example, CPU usage or memory usage.

Sampling profilers are typically less accurate and precise, because they sample the profiled traits, but they have minimal impact on the performance of the profiled application, particularly important in continuous profiling of production code. Accuracy improves as the number of samples improves, though it is a statistical approximation.

For information on running the Stackdriver Profiler agent with your application, see the following pages:

- Profiling Go applications (/profiler/docs/profiling-go)
- Profiling Java applications (/profiler/docs/profiling-java)
- Profiling Node.js applications (/profiler/docs/profiling-nodejs)
- Profiling Python applications (/profiler/docs/profiling-python)
- Profiling applications running outside Google Cloud (/profiler/docs/profiling-external)

After collecting profiler data, you analyze it using the Profiler interface (/profiler/docs/using-profiler).

Stackdriver Profiler supports different types of profiling based on the language in which a program is written. The following table summarizes the supported profile types by language:

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

- **CPU time** is the time the CPU spends executing a block of code.
- **Wall-clock time** (also called *wall time*) is the time it takes to run a block of code.

The CPU time for a function tells you how long it took to execute the code in the function. This measures the time the CPU was busy processing instructions. It doesn't include the time the CPU was waiting (or processing instructions for something else).

Wall-clock time for a function measures the time elapsed between entering and exiting a function. This includes time waiting for locks for database access, waiting for thread synchronization, waiting for locks, and so forth. The wall time for a block of code can never be less than the CPU time.

When the number of samples is low, the wall time can appear to be less than the CPU time in a sampling profile such as Stackdriver Profiler.

A block of code can take a long time to run but actually require little CPU time. If the wall time is much greater than the CPU time, the code spends a lot of time waiting for other things to happen. A block of code that spends a vast amount of its time waiting for other things to happen might indicate a resource bottleneck, where too many requestors are trying to access some limited resource.

If the CPU time is close to the wall time, the block of code is CPU intensive; almost all the time it takes to run is spent by the CPU. Long-running CPU-intensive blocks of code might be candidates for optimization: is there a more CPU-efficient way to do the work, something that involves fewer or faster operations?

- **Heap consumption** (also called *heap*) is the amount of memory allocated in the program's heap when the profile is collected.
- **Heap allocation** (also called *allocated heap*) is the total amount of memory that was allocated in the program's heap, including memory that is freed and no longer in use.

As programs run, they consume memory. They create objects; those objects take up space. They call functions; those functions take up space.

A well-behaved program uses memory efficiently and judiciously. It uses only the memory it needs; that is, it doesn't have an overly large memory footprint. It also returns that memory when it no longer needs it; that is, it doesn't leak memory.

A program that consumes more memory than it truly needs or that holds onto memory it no longer needs might start slowly, might gradually slow down or even crash, and might even affect resources available to other applications. A program which allocates memory more frequently than it truly needs, in a garbage collected language, creates more work for the garbage collector.

Profiling heap consumption helps you find potential inefficiencies and memory leaks in your programs. Profiling heap allocations helps you know which allocations are causing the most work for the garbage collector.

Applications that create threads can suffer from blocked threads, threads that are created but never actually get to run, and from thread leaks, where the number of threads created keeps increasing. The first problem is one cause of the second.

In a multi-threaded program, the time spent waiting to serialize access to a shared resource can be significant. Understanding contention behavior can guide the design of the code and provide information for performance tuning.

To use the Stackdriver Profiler with your service, for all languages except Java, you need to modify your service to instantiate a profiling agent when your service starts. For Java applications, you need to modify how you start your service. For each instance of the application, a profiling agent is instantiated.

The role of an agent is to capture profile data from your service and to transmit this data to the Profiler backend using the Profiler API. Each profile is for a single instance of a service and it includes four fields that uniquely identify its deployment:

- GCP project
- Service name
- Service zone
- Service version

When an agent is ready to capture a profile, it issues a Profiler API command to the Profiler backend. The backend receives this request and, in the simplest scenario, immediately replies to the agent. The reply specifies the type of profile to capture. In response, the agent captures the profile and transmits it to the backend. Lastly, the Profiler backend associates the profile with your Google Cloud project. You can then view and analyze it by using the Profiler UI.

The actual handshake sequence is more complex than described in the previous paragraph. For example, when the Profiler receives a request from an agent that is ready to collect a profile, the backend checks its database to determine if it has received previous requests from the agent. If not,

the backend adds the agent information to its database. A new deployment is created if the agent deployment fields don't match those of any other recorded agent.

Each minute, on average, and for each deployment and each profile type, the backend selects an agent and instructs it to capture a profile. For example, if the agents for a deployment support Heap and Wall time profiling, on average, 2 profiles are captured each minute:

- For all profile types except heap consumption and thread consumption, a single profile represents data collected for 10 seconds.
- For heap consumption and thread profiles, each profile is collected instantaneously.

The key observation is that after the agent notifies the Profiler backend that it's ready to capture data, the agent idles until it receives a reply from the backend that specifies the type of profile to capture. If you have 10 instances of a service running in the same deployment, then you create 10 profiling agents. However, most of the time these agents are idle. Over a 10-minute period, you can expect 10 profiles; each agent receives one reply for each profile type, on average. There is some randomization involved, so the actual number might vary.

The Profiler backend uses Profiler API quotas and the profile deployment fields to limit the profiles ingested. For information on viewing and managing your Profiler quotas, see Quotas & limits (/profiler/quotas).