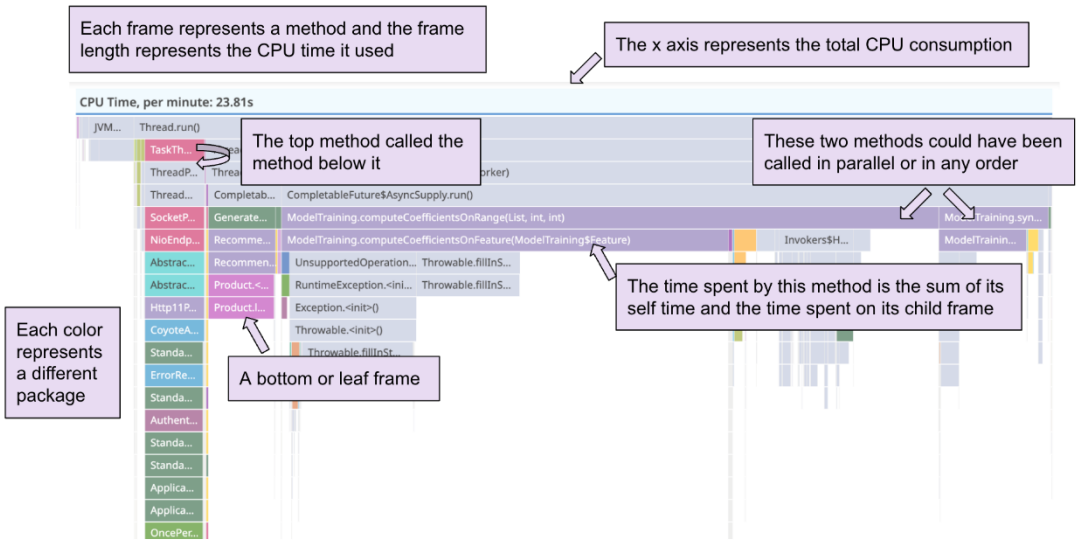
Datadog Continuous Profiler is an always-on, production code profiler that enables you to analyze code-level performance across your entire environment, with minimal overhead.

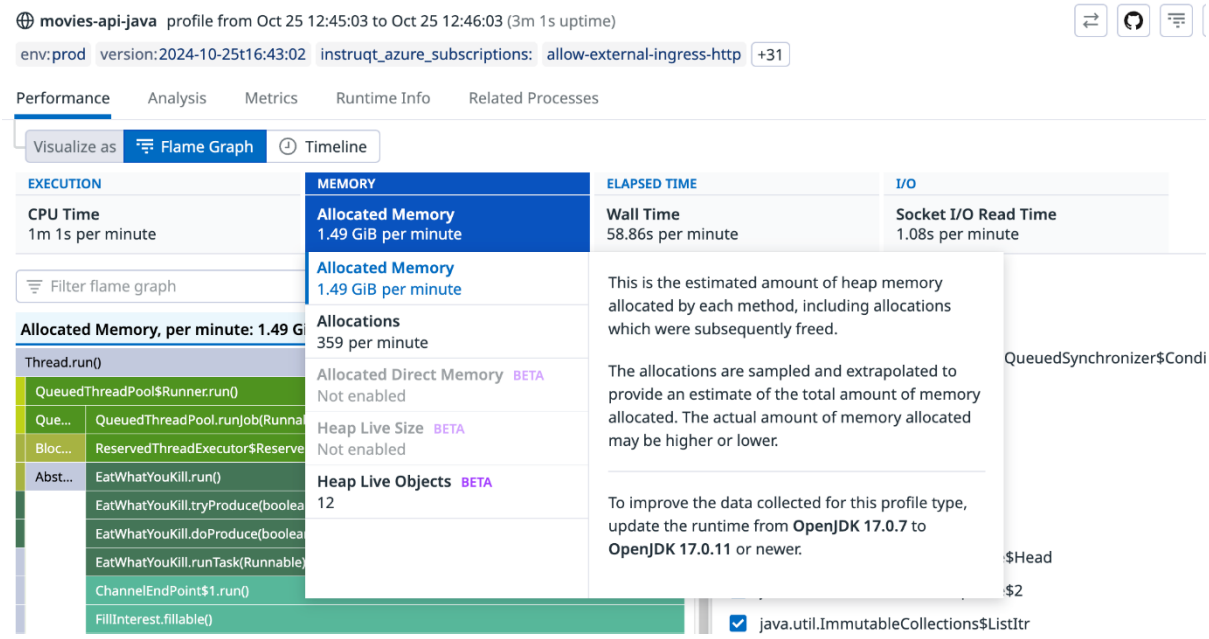
Profiles reveal which methods/functions consume the most resources, such as CPU, memory allocation, wall time, and I/O time spent.



Profiler flame graph with important details explained.

- The x-axis represents the total CPU consumption (not the time range over which the profiles were aggregated).
- Each horizontal bar is a frame (whereas in a trace, each bar is a span).
- Each frame represents a method. The frames are arranged from top to bottom, in the order that each method was called during a program's execution.
- Each color represents a different package.
- The top frame defined here is usually called the "root frame" and its value is the sum of the child frames. Comparing it to a pie chart, the root frame is the total pie (sum of the resource consumption) and each stack trace represents a different piece of the pie.
- The width of each frame corresponds to its resource consumption. The longer the frame, the more CPU Time was used. The width of the frame is the cumulative time, which is equal to the amount of time spent on the method itself (called the self time) plus the time spent on its child frames.
- Two methods appearing side by side could have been called in parallel or in any order. Frames are ordered alphabetically from left to right.
- The bottom frame is called the leaf frame and represents the last method called in the stack. The leaf frame only represents its self time because it has no child frames.

There are a variety of profile types that you can view in Continuous Profiler to help you understand and investigate different aspects of your code's performance.



Example of available profile types based on the filtered profile data.

Each profile type represents a type of resource consumption, such as CPU, wall time, or memory. The profile types available for you to select may differ based on the language being profiled. The following are the most common profile types:

- **CPU** profiling measures which methods consume the most CPU on an application.
- **Allocation** profiling measures the amount (both in terms of count and size in Bytes) of memory allocated by a given method. Note: This isn't retained memory, which means that the allocated memory that is measured may or may not be garbage collected.
- **Heap** profiling measures the amount of heap memory allocated by each function that hasn't been garbage collected (yet). This is useful for investigating the overall memory usage of your service and identifying potential memory leaks.
- **Lock** profiling measures the amount of time a thread is waiting to acquire a lock and is hence doing nothing.
- **Wall Time** profiling measures the effective time spent by methods (regardless of whether those methods were running on CPU, waiting for network I/O, blocked by another thread, or just idle). It can be useful to debug latency at first glance and then dig into the other profiling types to find out what was causing the latency. The wall time profile can be considered to be the most similar to the associated APM flame graph.
- **File I/O** and **Socket I/O** measure the number of time spent by methods on disk (for example, reading a file from disk) and network I/O operations (for example, waiting for an API call to return).
- **Exceptions** measures the amount of exceptions thrown. The profiler doesn't catch/handle exceptions, but it tracks their creation. (For Java, exceptions and errors aren't synonymous, and errors are more likely to be unrecoverable, such as in the case of an OutOfMemoryError).

Endpoint profiling allows you to scope profiler flame graphs by any API endpoint of a service, so you can find endpoints that are slow and causing poor end-user experience.

Debugging and understanding why an endpoint has high latency can be tricky. For example, high latency could be caused by a method that is CPU heavy and unknowingly on the critical path of a request process where latency is important.

You can do the following with endpoint profiling:

- Identify the bottleneck methods that are slowing down the endpoint's overall response time.
- Isolate the top endpoints that are responsible for consuming resources like CPU and memory. This is particularly helpful when you're trying to optimize your service for performance gains.
- Understand if third-party code or runtime libraries are the reason for endpoints being slow or heavy on resource consumption.

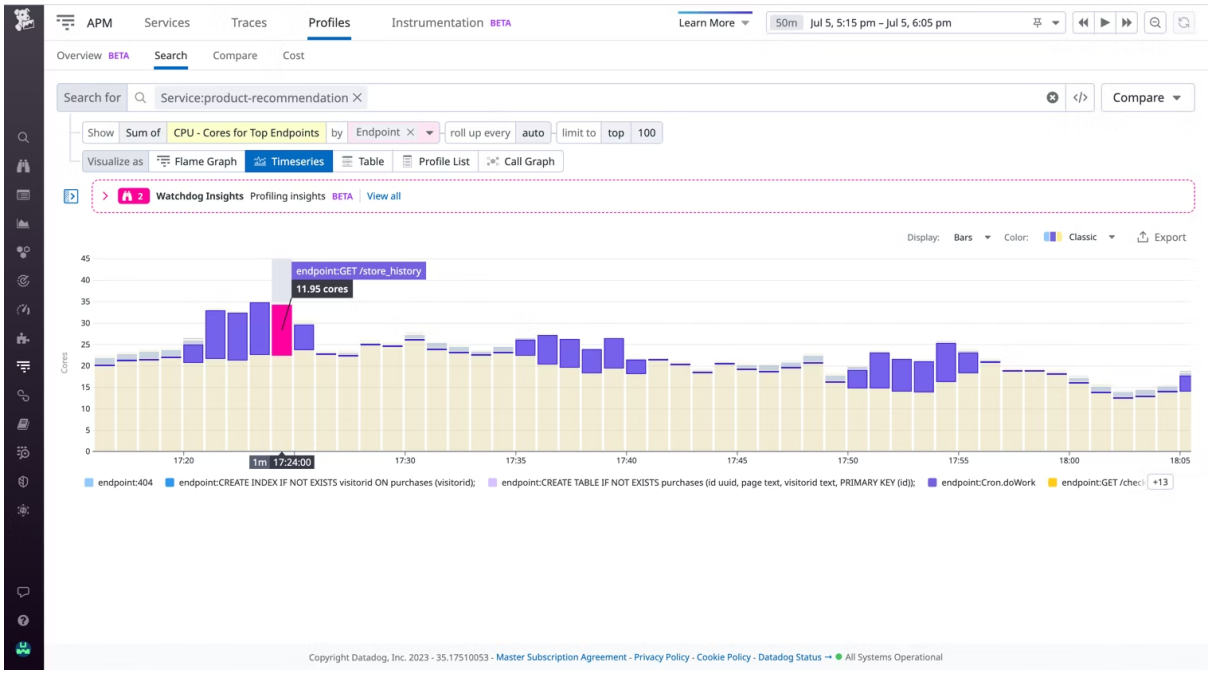
Example of list of endpoints that appears in the summary list for a profile.

In general, it's valuable to track which endpoints are consuming the most valuable resources, such as CPU and memory. The list can help you identify if your endpoints have regressed or if you have newly introduced endpoints that are drastically consuming more resources than expected and slowing down your overall service.

Track the endpoints that consume the most resources

It is valuable to track top endpoints that are consuming valuable resources such as CPU and wall time. The list can help you identify if your endpoints have regressed or if you have newly introduced endpoints that are consuming drastically more resources, slowing down your overall service.

The following image shows that `GET /store_history` is periodically impacting this service by consuming 20% of its CPU:



Track average resource consumption per request

Select Per endpoint call to see behavior changes even as traffic shifts over time. This is useful for progressive rollout sanity checks or analyzing daily traffic patterns.
The following video shows that CPU per request doubled for /GET train.