Google
Automotive
Partner
Bootcamp

# Performance Analysis / Tuning 101

# Ethan
# Lee

Software Engineer

# Agenda

# A Scientific Approach to Performance Analysis

# What is Performance Analysis?

- Performance issues require a systematic process to uncover their root cause.
- The right tools need to be identified to gather insights into critical parts of complex systems.
- There are a number of techniques which engineers can use to delve deeper into the execution of a system.

# What is Performance Analysis?

There are two techniques that are widely used for performance analysis: Tracing and Profiling

## Tracing

- Tracing involves collecting highly detailed data about system execution.
- Traces contain enough detail to build a timeline of events.
- Traces give us insight into what a program does over time (e.g. which functions are being run) and context about execution (e.g. function call parameters).

## Profiling

- Profiling involves sampling some usage of a resource by a program.
- The most common types are memory profiling and CPU profiling.
- Memory profiling surfaces information about heap memory allocation.
- CPU profiling gathers information about the call stack running on a CPU over time.

# Why Choose Perfetto?

Profiling and tracing have different use cases:

**Why use profiling over tracing?**

- Traces, while detailed, are impractical for capturing high-frequency events like every function call due to the sheer volume of data involved.
- Profilers address this limitation through sampling, selectively recording data points to drastically reduce storage requirements.

**Why use tracing over profiling?**

- Profilers offer valuable insights into where resources are consumed within a program's call stack, but they lack the ability to explain the underlying reasons behind those resource allocations.
- For instance, a profiler might reveal that function foo() called malloc numerous times and allocated X bytes, but it cannot tell us why foo() was making those calls.
- Traces fill this gap by combining application and kernel events, providing in-depth context to understand the root cause of resource consumption.

**Perfetto addresses this by supporting the collection, analysis and visualization of both tracing and profiling.**

# How to Use Perfetto Effectively

## How does Perfetto and our performance analysis flow fit into our goals?

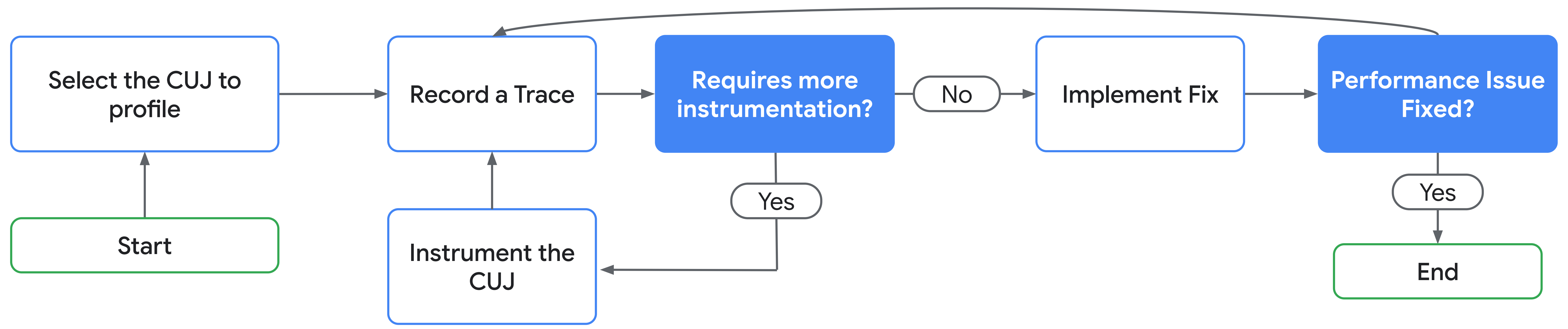
This approach allows one to easily compare the delta of a potential regression. To achieve this, one should have an established baseline to compare against.

Perfetto will enable one to gather insights beyond just surface level observations. It is imperative that we can translate user-perceptible signals into measurable metrics that can be tested.

Using Perfetto in this approach means that multiple iterations will need to be collected. In order to establish reliable metrics, it is necessary to gather information on a large enough population to capture reproducible issues.

# How to Use Perfetto Effectively

## How does Perfetto and our performance analysis flow fit into our goals?



# Perfetto: A Feature Rich Tool

## Ease of Use

Perfetto provides an end-to-end solution to capture Android system traces quickly to identify issues in critical user flows.

## Flexibility

Via Perfetto trace configs, users are able to modify tracing behavior via buffers or data sources. For example, one can easily change data sources to capture various ftrace events or atrace events.

## Trace Analysis

Perfetto provides a comprehensive trace viewer web UI that empowers one to inspect, visualize, and analyze the collected data.

## Data Mining

One can leverage SQL-like syntax to query the trace data, making complex analysis easier.

# How does Perfetto Work?

https://perfetto.dev/docs

## Record traces

### System tracing

- Linux ftrace
- /proc pollers
- Heap profilers

Data sources
Linux/Android

Tracing daemon
UNIX socket

### Chrome tracing

Chrome-specific data-sources

Tracing service
Mojo

### In-app tracing

App-specific data-sources

In-process service thread

Track event library
TRACE_EVENT(...)

Tracing C++ Library
Android / Linux / MacOS / Windows

## Analyze traces

**Trace Processor**
Android / Linux / MacOS / Win

Trace importers
Protobuf, JSON, systrace

SQL query engine
Based on SQLite

Trace-based metrics
JSON / Protobuf / CSV

## Visualize traces

**Perfetto UI**
HTML / JS

Trace Processor
Web Assembly

ADB over WebUSB
For Android

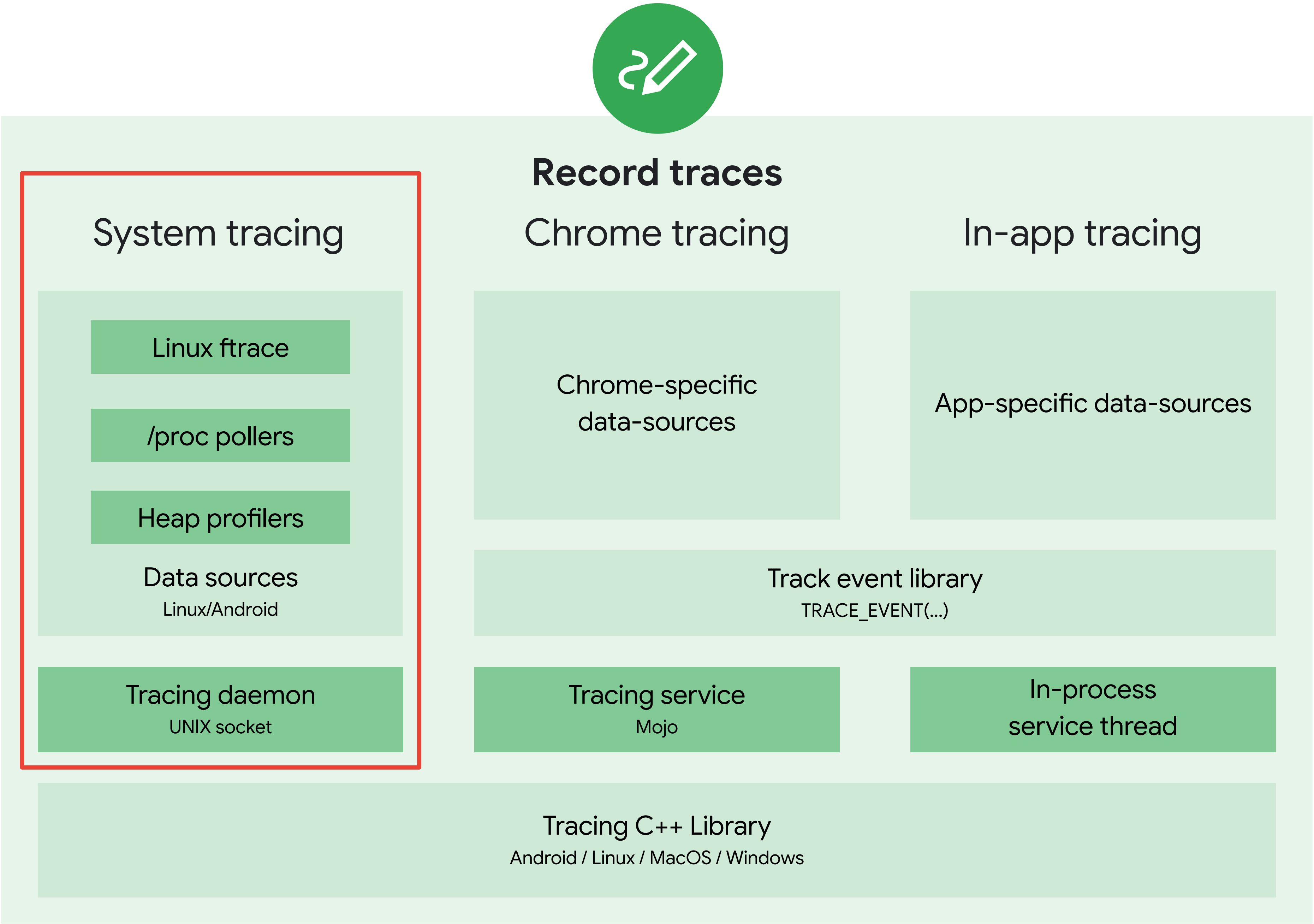
Works offline
After first visit

# How does Perfetto Work?

### System Wide Tracing for Android and Linux

- Kernel tracing is enabled via Linux ftrace, which allows kernel events such as scheduling events and syscalls to be recorded.
- /proc pollers allow the sampling of process-wide cpu and memory counters over a time period.
- Heap profilers also enable capturing information for the Native and Java heap.

https://perfetto.dev/docs

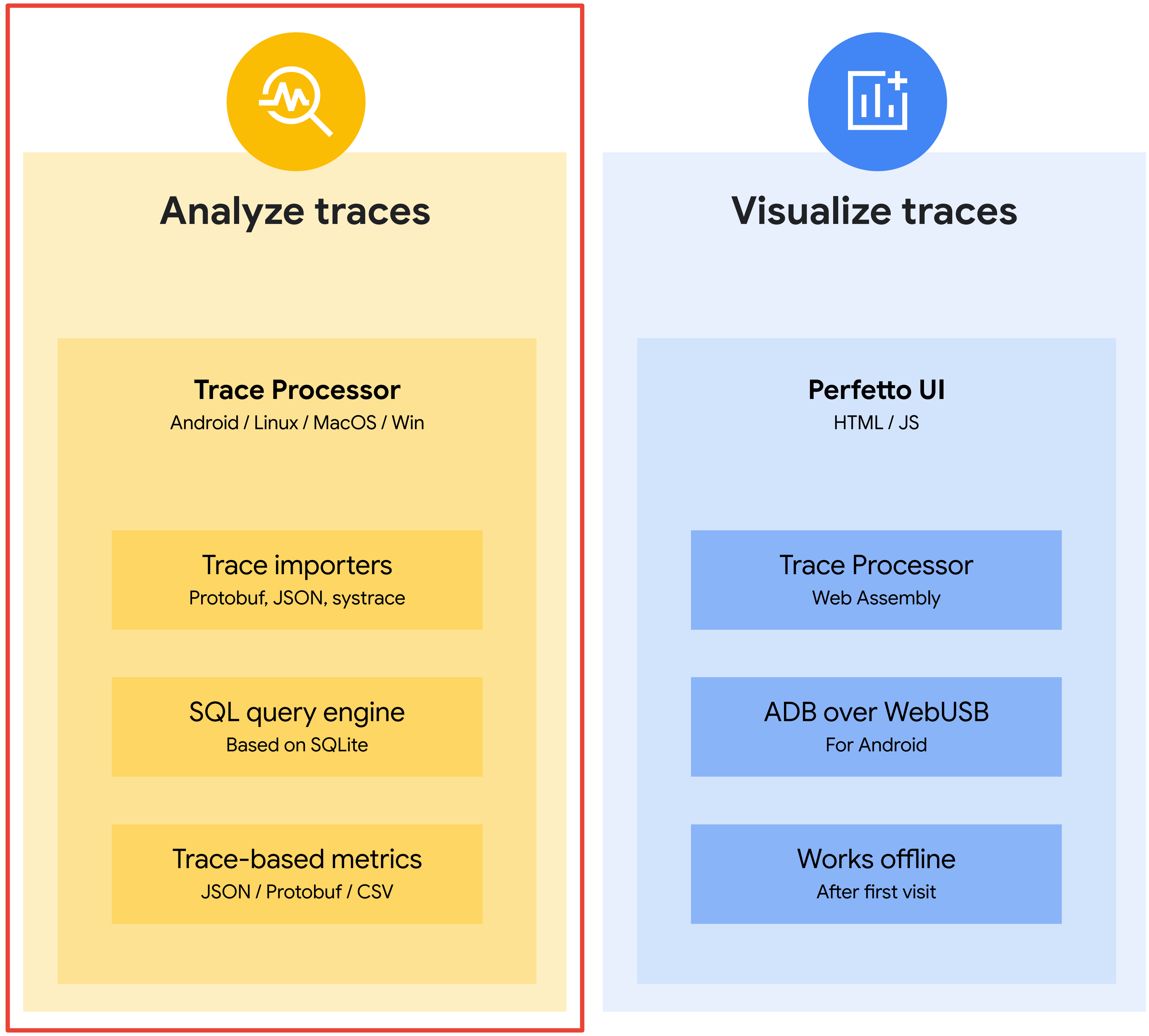


# How does Perfetto Work?

## Trace Analysis

- The Trace Processor is a C++ library that takes in raw trace data and surfaces it through an SQL interface for straight-forward querying.
- Trace importers allow simple ingestion of multiple formats
- Trace-based metrics creates pre-formatted and extensible queries that provide trace summaries. (e.g. CPU usage at different frequency states).

[https://perfetto.dev/docs](https://perfetto.dev/docs)

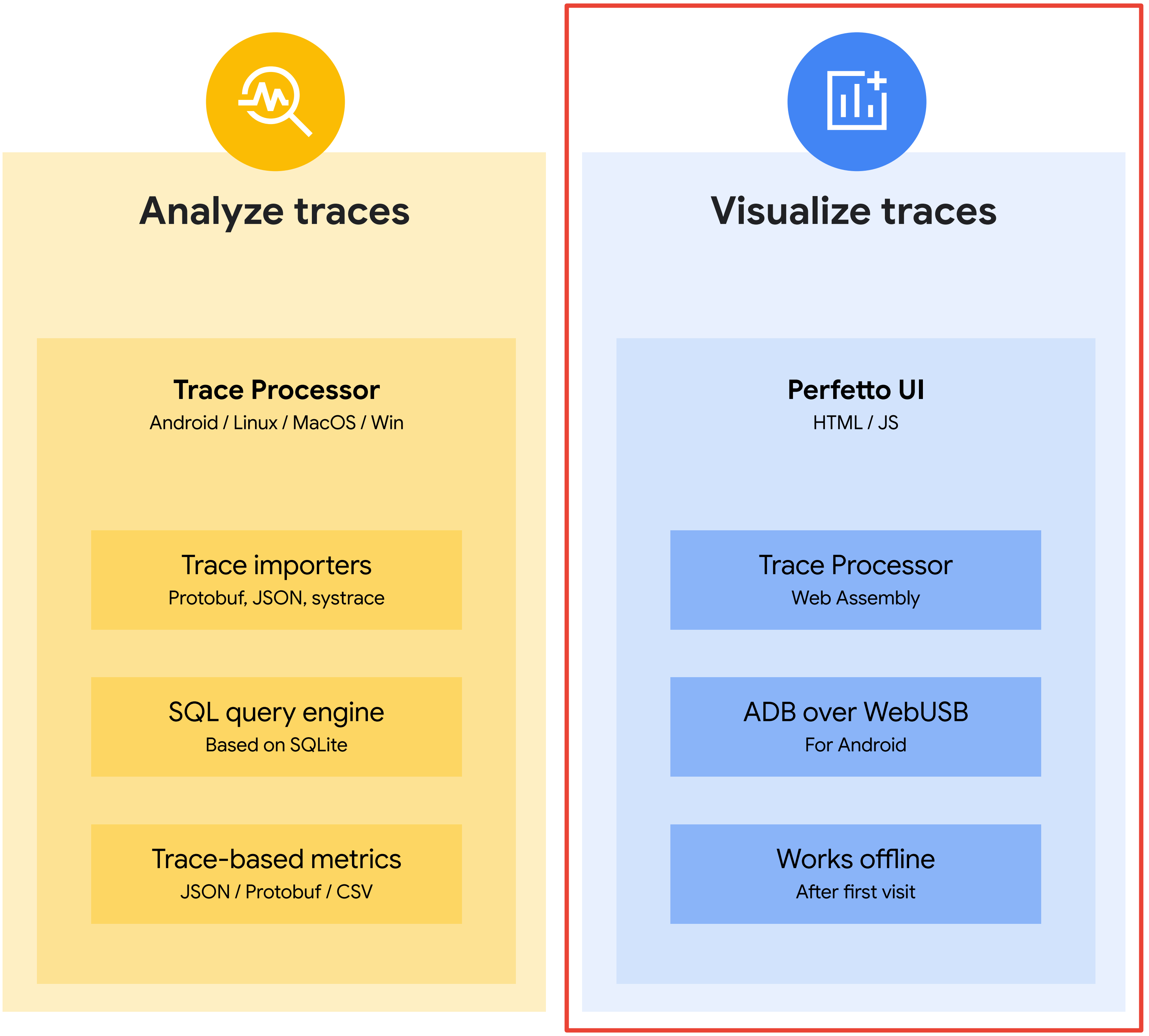


# How does Perfetto Work?

## Trace Visualization

- A trace visualizer is instrumental for analysis and is powered by WebAssembly.
- The Perfetto UI works fully offline after initial opening.

https://perfetto.dev/docs



# Getting Started with Perfetto

# Quick Start: Collecting a Perfetto Trace

**After defining an appropriate trace configuration, one can run the trace collection.**

1. Download the recording script using the below command:

```
$ curl -O https://raw.githubusercontent.com/google/perfetto/master/tools/record_android_trace
$ chmod u+x record_android_trace
```

2. Start tracing using:

```
$ ./record_android_trace -o <trace-name>.trace -c <previous trace file>
```

3. Run the desired CUJ or experiment
4. End the trace using Ctrl+C for the command run in Step 2
5. The trace will be automatically be opened in the browser after the collection has completed

# Quick Start:
# Viewing a Trace

If one wants to open an existing trace file, navigate to **ui.perfetto.dev** to open and access a trace:

# Quick Start:
# Viewing a Trace

Once the trace is generated, one can also generate a permalink to the existing trace that can be shared:

# Demo Video

An example video of a trace being collected from beginning to end.

# Special Case: Collect Boot Time Tracing

In Android TM+, the trace can be collected as seen previously. However the following setting must be enabled before the device is restarted:

```
adb shell setprop persist.debug.perfetto.boottrace 1
```

In Android SC-, the following steps are required to setup the device:

1. Boot tracing in Android SC- requires selinux to be set to permissive.
2. The following .rc file on the right must be created.
3. adb root && adb remount must be run to remount the device.
4. Use the following commands to push the .rc and config file to the device:

```
adb push perfetto_boot.rc /etc/init/perfetto_boot.rc
adb push perfetto_trace_config.textproto
/data/misc/perfetto-traces/boottrace.pbtxt
```

```
cat >> perfetto_boot.rc << 'EOF'
service perfetto_boot /system/bin/perfetto --txt -c
/data/misc/perfetto-traces/boottrace.pbtxt -o
/data/misc/perfetto-traces/boottrace.perfetto-trace
    class late_start
    disabled
    user shell
    group nobody
    oneshot
    seclabel u:object_r:perfetto_exec:s0
    stdio_to_kmsg
    capabilities DAC_READ_SEARCH

on property:persist.perfetto.boottrace=1
    rm /data/misc/perfetto-traces/boottrace.perfetto-trace
    start perfetto_boot
EOF
```

# Special Case: Collect Boot Time Tracing

The following steps are required to collect the trace:

1. Reboot the device using adb reboot
2. Stop perfetto and pull the trace:

```
adb shell pkill perfetto

adb pull /data/misc/perfetto-traces/boottrace.perfetto-trace
```

# Trace Anatomy

# Trace Config Setup

**Selecting the right trace config will allow one to collect the necessary data from the system.**

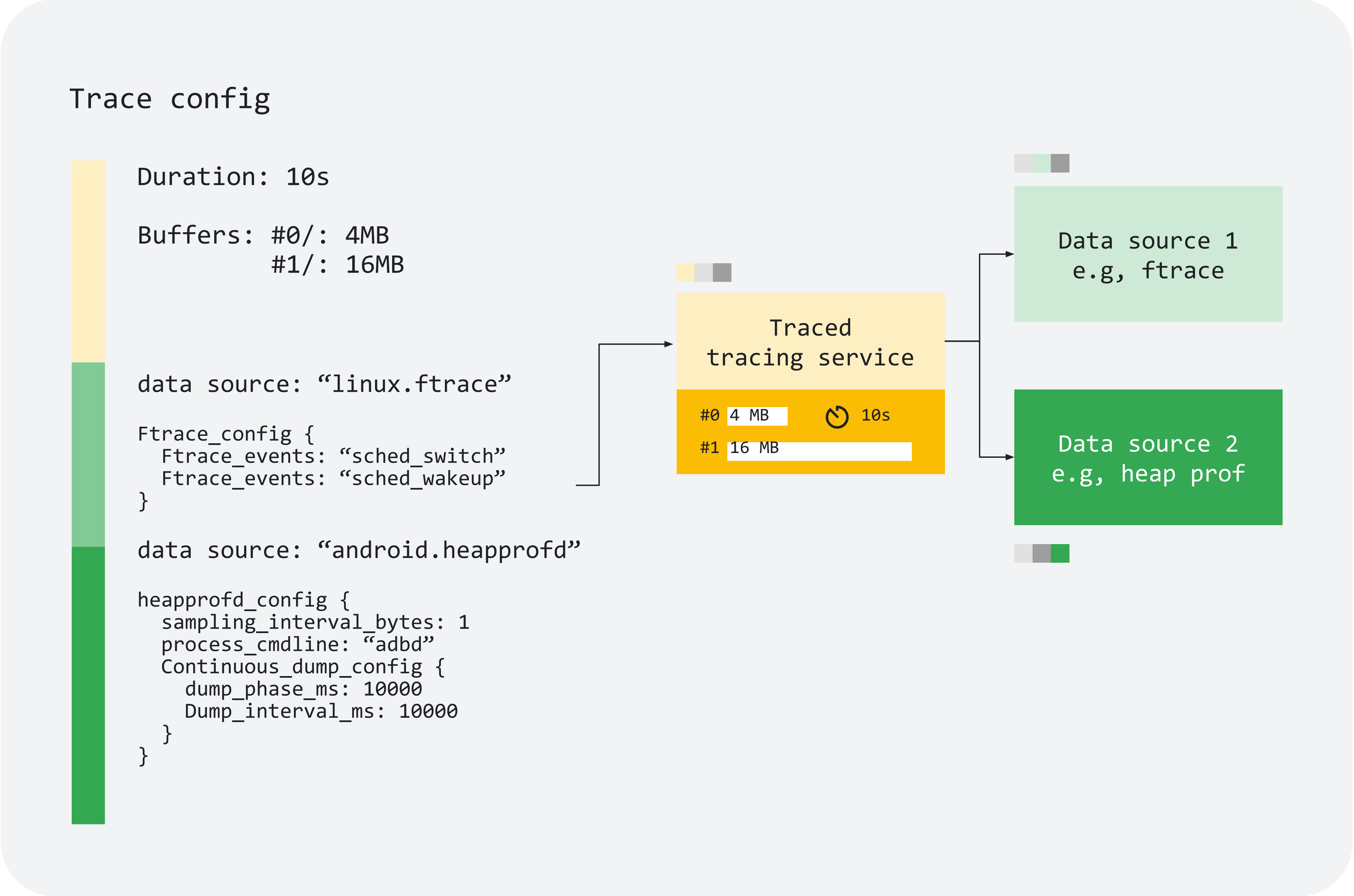
- Perfetto provides granular control over data collection. Unlike always-on logging systems (e.g., Linux's rsyslog, Android's logcat), its tracing data sources start in an idle state.
- The TraceConfig is a protobuf message that controls your Perfetto tracing session. It outlines:
- **System-wide Settings:**
  - Maximum trace duration.
  - Number and size of memory buffers.
  - Maximum output file size.
- **Data Source Specifications:**
  - For kernel tracing, which ftrace events to enable.
  - For the heap profiler, the target process name and sampling rate.
- **Data Routing:** Specifies which buffer each data source should write into

**Note: a sample config can be found at perfetto.dev/docs/concepts/config**

# Perfetto Trace Config

## How the Tracing Service Uses the TraceConfig

- The tracing service (traced) is your config manager. When you start a tracing session, the service:
    - **Reads System Settings:** It determines its behavior based on the TraceConfig's outer section (duration, buffers, etc.).
    - **Activates Data Sources:** It finds Producers that match the data sources listed in the config. Then, it starts each Producer and provides the relevant DataSourceConfig settings.



# Perfetto Trace Config

### Defining Buffers:

- This section defines the number, size and policy of in-memory buffers owned by the tracing service.
- Fill Policy:
  - A RING_BUFFER (default) fill policy will wrap over when full and replace the oldest trace data in the buffer.
  - A DISCARD fill policy will stop accepting data once full.

### Dynamic Buffer Mapping:

- The target_buffer field can be specified to indicate different buffers for data sources.

```
# Defining several buffers
buffers: {
  size_kb: 4096
  fill_policy: RING_BUFFER
}
buffers {
  size_kb: 4096
  fill_policy: RING_BUFFER
}
```

# Perfetto Trace Config

## Defining Data Sources (Logcat)

This data source will enable Android logcat messages to be shown:

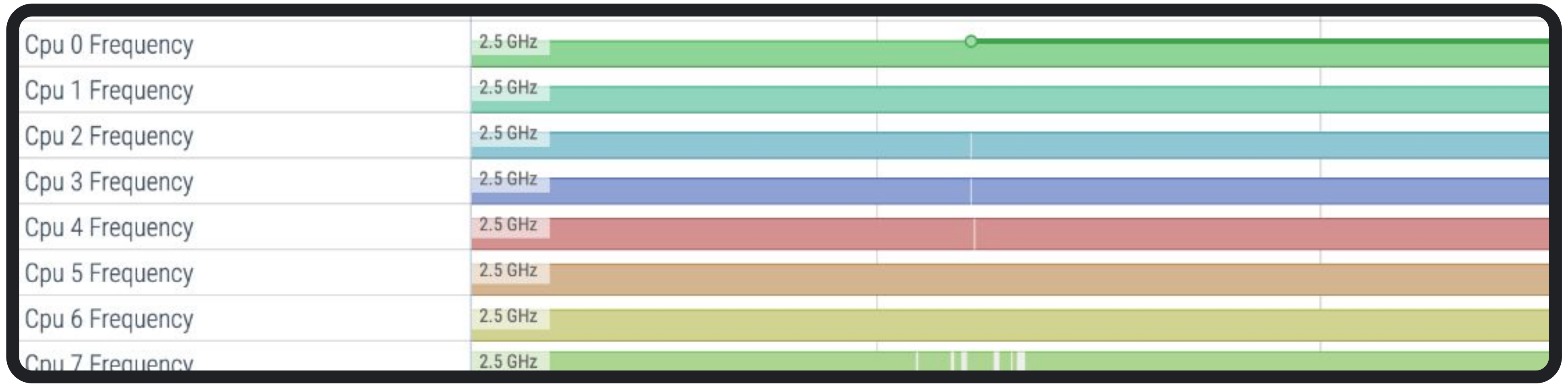
```
data_sources: {
  config {
    name: "android.log"
    android_log_config {
    }
  }
}
```

# Perfetto Trace Config

## Defining Data Sources (CPU Frequency)

Various CPU frequency stats can be collected with the following data sources:

- Enabling the power/cpu_frequency ftrace event
- Setting cpufreq_period_ms > 0 **(Note: only works on Android SC-V2 and above)**




```
data_sources: {
  config {
    name: "linux.sys_stats"
    target_buffer: 1
    sys_stats_config {
        cpufreq_period_ms: 500
    }
}

data_sources: {
    config {
        name: "linux.ftrace"
        ftrace_config {
            ftrace_events: "power/cpu_frequency"
            ftrace_events: "power/cpu_idle"
            ftrace_events: "power/suspend_resume"
        }
    }
}
```

# Perfetto Trace Config

## Defining Data Sources (Jankiness)

Jankiness can be examined with the frame timeline data source.

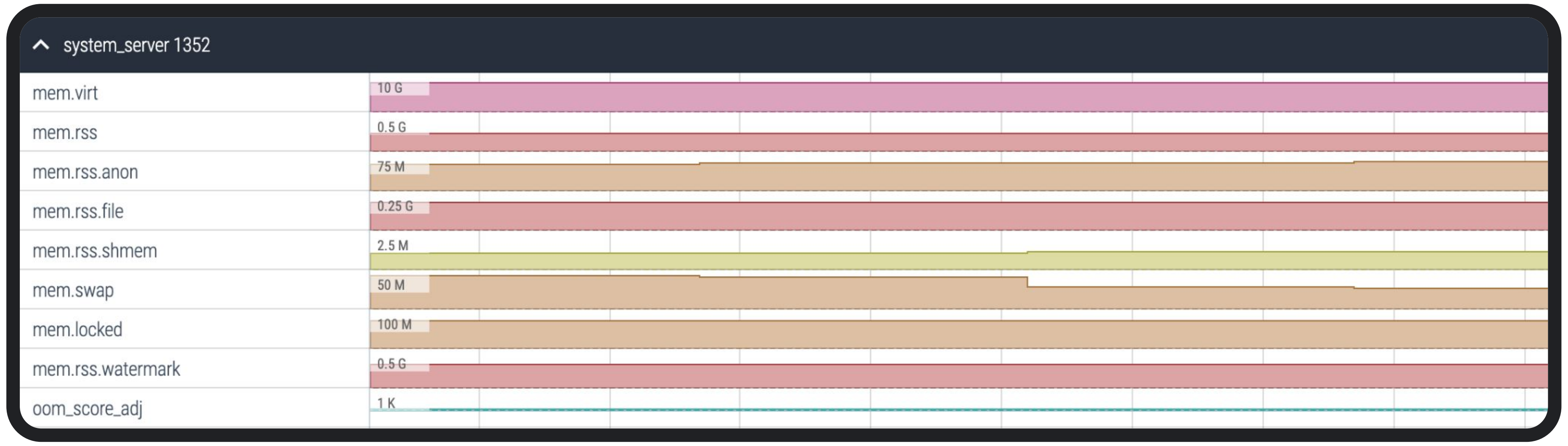
```
data_sources: {
  config {
    name: "android.surfaceflinger.frametimeline"
    target_buffer: 2
  }
}
```

# Perfetto Trace Config

## Defining Data Sources (linux.process_stats)

- The linux.process_stats data source gathers per-process statistics from the /proc/<pid>/status and /proc/<pid>/oom_score_adj files on Linux systems
    - Process memory usage (RSS, VMSize, etc.)
    - Open file descriptors
    - Out-of-memory (OOM) score (indicates how likely the kernel is to terminate the process when memory is low)

```
data_sources: {
  config {
    name: "linux.process_stats"
    process_stats_config {
      scan_all_processes_on_start: true
      proc_stats_poll_ms: 1000
    }
  }
}
```



# Perfetto Trace Config

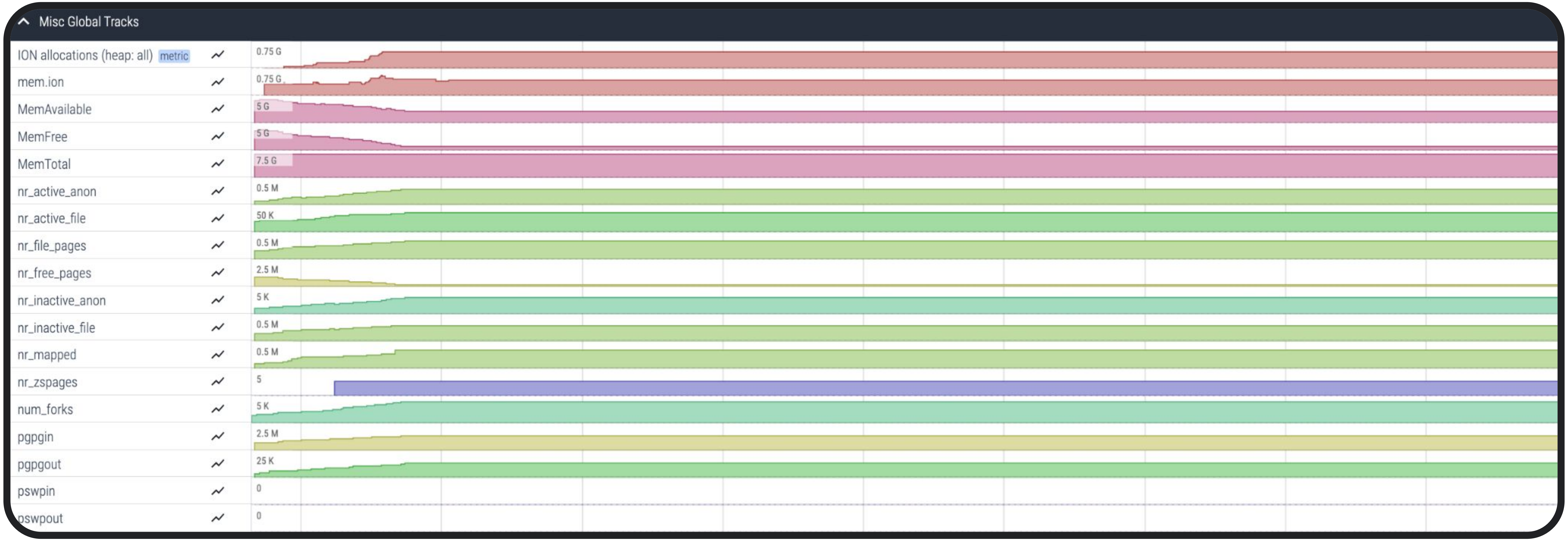
## Defining Data Sources (linux.sys_stats)

- The linux.sys_stats data source gathers a range of system-level statistics from Linux. The following stat counters can be collected:
  - Stat Counters (proc/stat):
    - STAT_CPU_TIMES
      - user: Time spent running in user mode
      - nice: Time spent running niced user processes
      - system: Time spent in system (kernel) mode
      - idle: Time the process was idle
  - Mem Info Counters (proc/meminfo):
    - Provides information such as free memory, anonymous memory.
  - VM Stat Counters (proc/vmstat):
    - Provides information on virtual memory such as page faults, pages in and out, etc.
  - Note: cpufreq_period_ms is only available above SC-V2.
    - The following error will be encountered otherwise:
    - `No field named "cpufreq_period_ms" in proto SysStatsConfig.`

```
data_sources: {
  config {
    name: "linux.sys_stats"
    target_buffer: 1
    sys_stats_config {
      stat_period_ms: 500
      stat_counters: STAT_CPU_TIMES

      meminfo_period_ms: 1000
      meminfo_counters: MEMINFO_ACTIVE_ANON
      meminfo_counters: MEMINFO_ACTIVE_FILE
      meminfo_counters: MEMINFO_INACTIVE_ANON
      meminfo_counters: MEMINFO_INACTIVE_FILE
      meminfo_counters: MEMINFO_KERNEL_STACK
      meminfo_counters: MEMINFO_MLOCKED
      meminfo_counters: MEMINFO_SHMEM
      meminfo_counters: MEMINFO_SLAB
      meminfo_counters: MEMINFO_SLAB_UNRECLAIMABLE
      meminfo_counters: MEMINFO_VMALLOC_USED
      meminfo_counters: MEMINFO_MEM_FREE
      meminfo_counters: MEMINFO_SWAP_FREE

      vmstat_period_ms: 1000
      vmstat_counters: VMSTAT_PGFAULT
      vmstat_counters: VMSTAT_PGMAJFAULT
      vmstat_counters: VMSTAT_PGFREE
      vmstat_counters: VMSTAT_PGPGIN
      vmstat_counters: VMSTAT_PGPGOUT
      vmstat_counters: VMSTAT_PSWPIN
      vmstat_counters: VMSTAT_PSWPOUT
      vmstat_counters: VMSTAT_PGSCAN_DIRECT
      vmstat_counters: VMSTAT_PGSTEAL_DIRECT
      vmstat_counters: VMSTAT_PGSCAN_KSWAPD
      vmstat_counters: VMSTAT_PGSTEAL_KSWAPD
      vmstat_counters: VMSTAT_WORKINGSET_REFAULT

      # Below field not available on < Android SC-V2 releases.
      cpufreq_period_ms: 500
    }
  }
}
```

# Perfetto Trace Config

## Defining Data Sources (linux.sys_stats)



# Perfetto Trace Config

## Defining Data Sources (ftrace)

Capturing ftrace events allows developers insights into kernel code. They are useful for analyzing latency or performance issues outside of userspace.

- Memory Events
- Low Memory Killer Events
- Sched Events

```
data_sources: {
  config {
    name: "linux.ftrace"
    target_buffer: 2
    ftrace_config {
      # Memory events
      ftrace_events: "power/suspend_resume"
      ftrace_events: "mm_event/mm_event_record"
      ftrace_events: "kmem/rss_stat"
      ftrace_events: "ion/ion_stat"
      ftrace_events: "dmabuf_heap/dma_heap_stat"
      ftrace_events: "kmem/ion_heap_grow"
      ftrace_events: "kmem/ion_heap_shrink"
      # LMKD events
      ftrace_events: "lowmemorykiller/lowmemory_kill"
      ftrace_events: "oom/oom_score_adj_update"
      ftrace_events: "oom/mark_victim"
      # sched events
      ftrace_events: "sched/sched_process_exit"
      ftrace_events: "sched/sched_process_free"
      ftrace_events: "sched/sched_switch"
      ftrace_events: "sched/sched_wakeup"
      ftrace_events: "sched/sched_wakeup_new"
      ftrace_events: "sched/sched_waking"
    }
  }
}
```

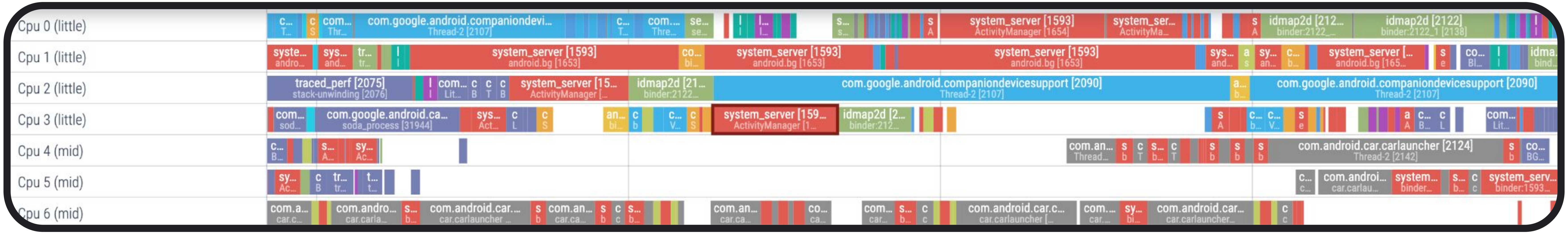
# Perfetto Trace Config

## Defining Data Sources (ftrace)

In order to capture CPU scheduling events, *ftrace_events: "sched/sched_switch"* needs to be added to the *linux.ftrace* data source.

With this enabled the following can be captured:

- Threads scheduled per CPU
- Why a thread got de-scheduled (pre-emption, blocked by a mutex)
- When a thread becomes runnable



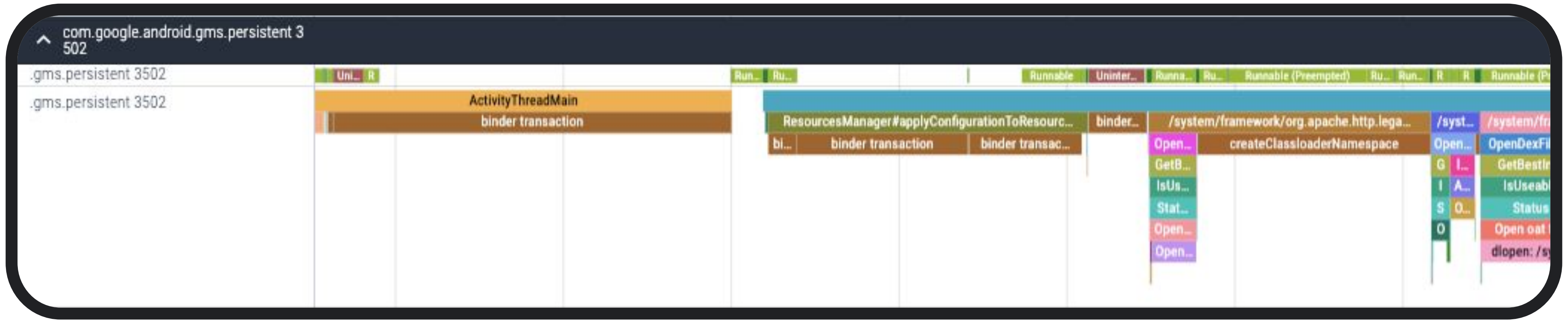
# Perfetto Trace Config

## Atrace Categories:

Predefined groups of trace events that make it easier to enable tracing for specific areas of the system.

## Fine-grained Process Tracing:

The atrace_apps functionality in Perfetto enables selective tracing of specific applications on Android. It allows you to capture trace data only from the processes of interest.




```
data_sources: {
  config {
    name: "linux.ftrace"
    target_buffer: 2
    ftrace_config {
      # Memory events
      atrace_categories: "aidl"
      atrace_categories: "am"
      atrace_categories: "dalvik"
      atrace_categories: "binder_lock"
      atrace_categories: "binder_driver"
      atrace_categories: "disk"
      atrace_categories: "freq"
      atrace_categories: "idle"
      atrace_categories: "gfx"
      atrace_categories: "hal"
      atrace_categories: "pm"
      atrace_categories: "power"
      atrace_categories: "rro"
      # atrace apps
      atrace_apps: "lmkd"
      atrace_apps: "system_server"
      atrace_apps: "com.android.systemui"
      atrace_apps: "com.google.android.gms"
      atrace_apps: "com.google.android.gms.persistent"
      atrace_apps: "android:ui"
      atrace_apps: "com.google.android.apps.maps"
    }
  }
}
```

# Perfetto Trace Config

## Writing to a Trace Output File:

If not recording time limit is specified, one will have to manually terminate the tracing session.

If duration_ms is specified then, the trace will terminate automatically.

If write_into_file is true, then Perfetto will periodically stream results into a trace file.

**Flush_period_ms** defines the default drain period. A shorter period means a smaller userspace buffer is required. However, this will increase the performance intrusiveness of tracing.

**Max_file_size_bytes** is used to cap the size of a trace file.

**Flush_period_ms** is used to periodically issue a Flush() to all data sources, forcing them to commit their data into the tracing service.

```
# No recording time limit (press CTRL+C to stop recording).
# Alternatively: uncomment the line below to set a time limit.
#duration_ms: 1800000
write_into_file: true
file_write_period_ms: 5000
max_file_size_bytes: 100000000000
flush_period_ms: 5000
```

# Anatomy of a Trace: Binder Transactions

There are two types of binder transactions:

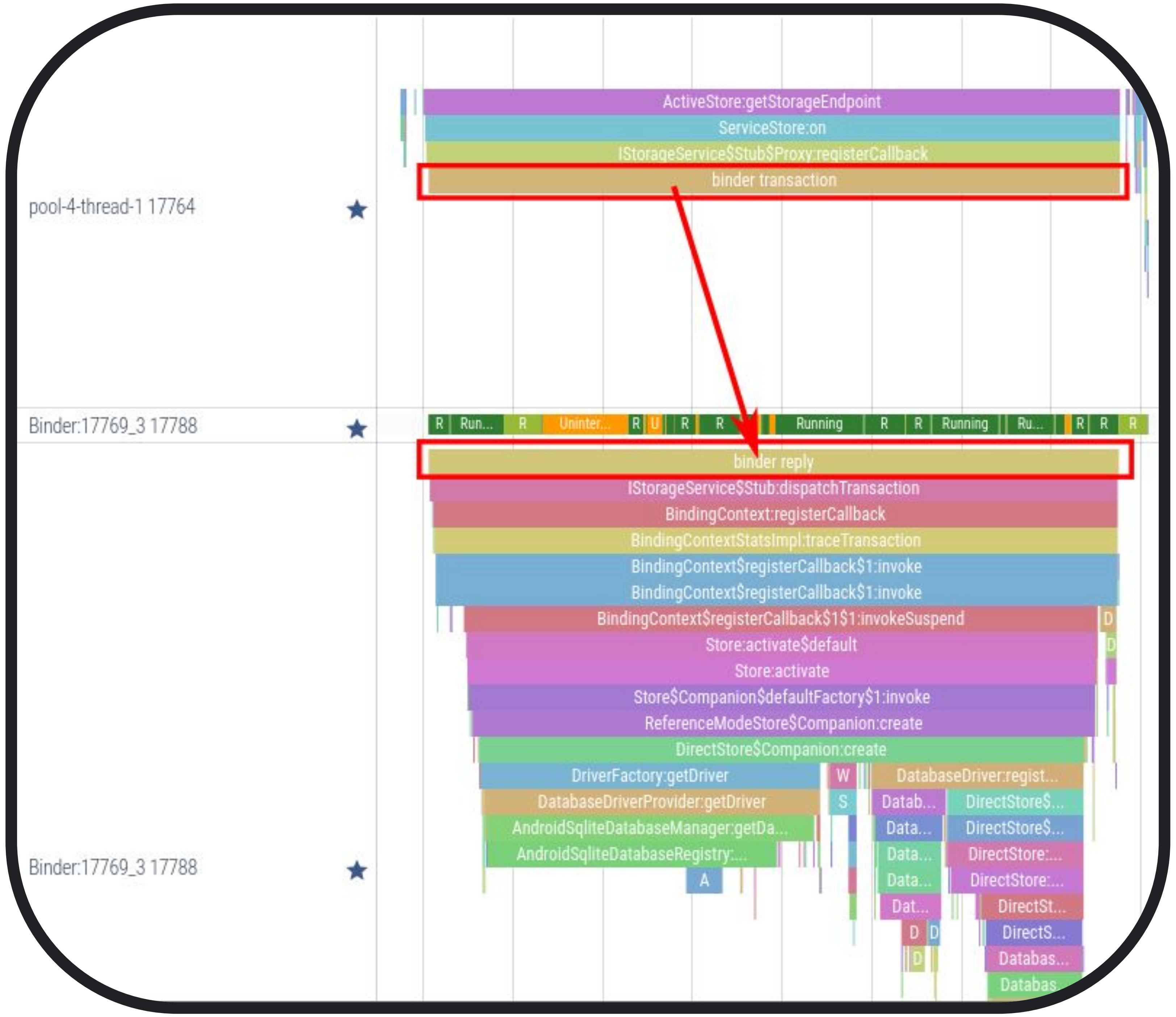
1. Unidirectional: Using the **oneway** keyword in the AIDL language, these transactions do not wait for a reply after sending a parcel.
2. Bidirectional: The transmitting end is blocked until it receives a reply.

**Note: This is only available in UDC onwards.**

# Anatomy of a Trace: Bidirectional Transactions

**Bidirectional Transaction:**

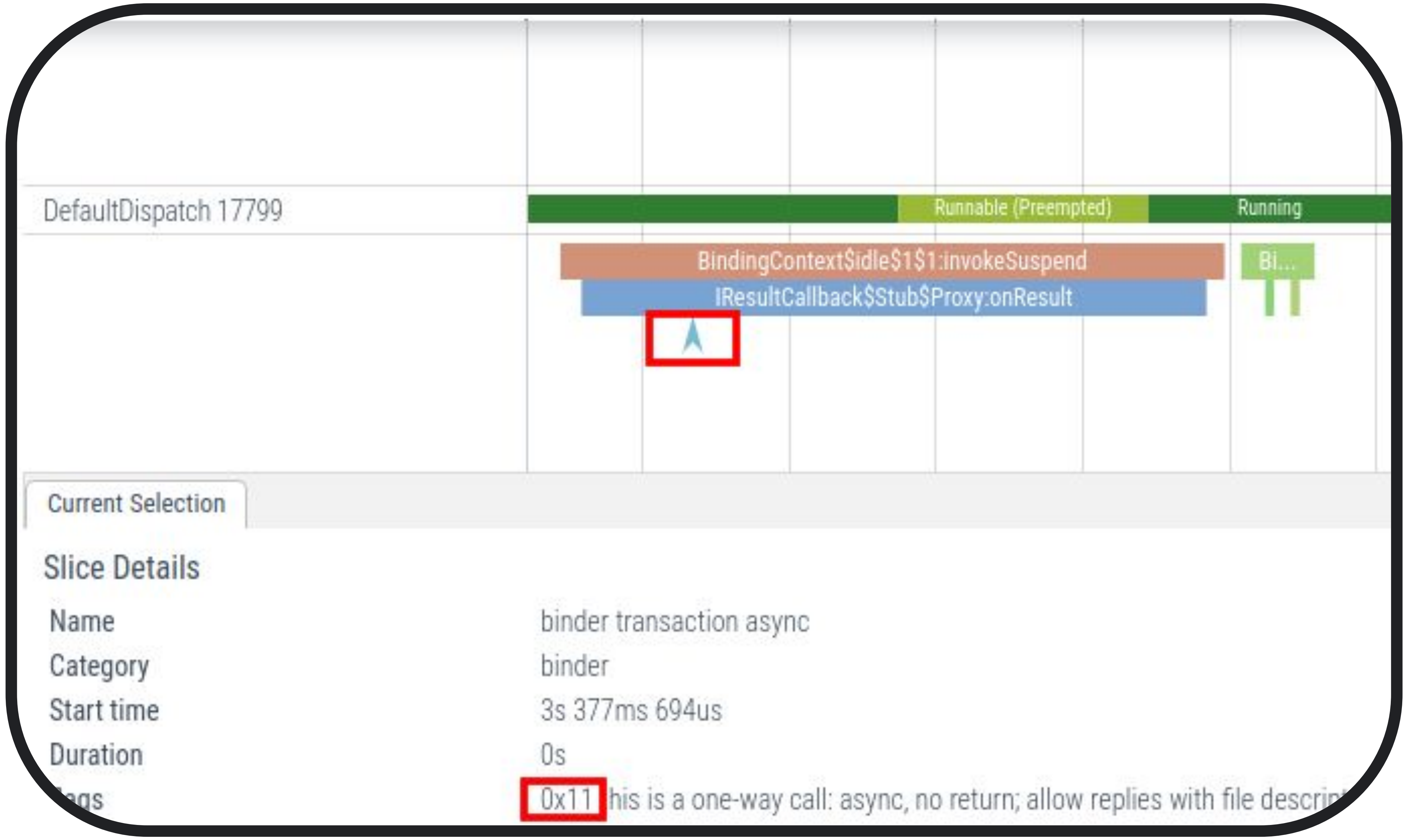
Identified by a corresponding binder transaction and binder reply pair.



# Anatomy of a Trace: Unidirectional Transactions

**Unidirectional Transaction:**

Indicated by an arrow in a Perfetto trace.



# Common Pitfalls

# Common Pitfalls

## External Process Interference

One of the reasons that a trace may be empty is that another process is using ftrace.

Run the following command to determine if the current_tracer is nop:

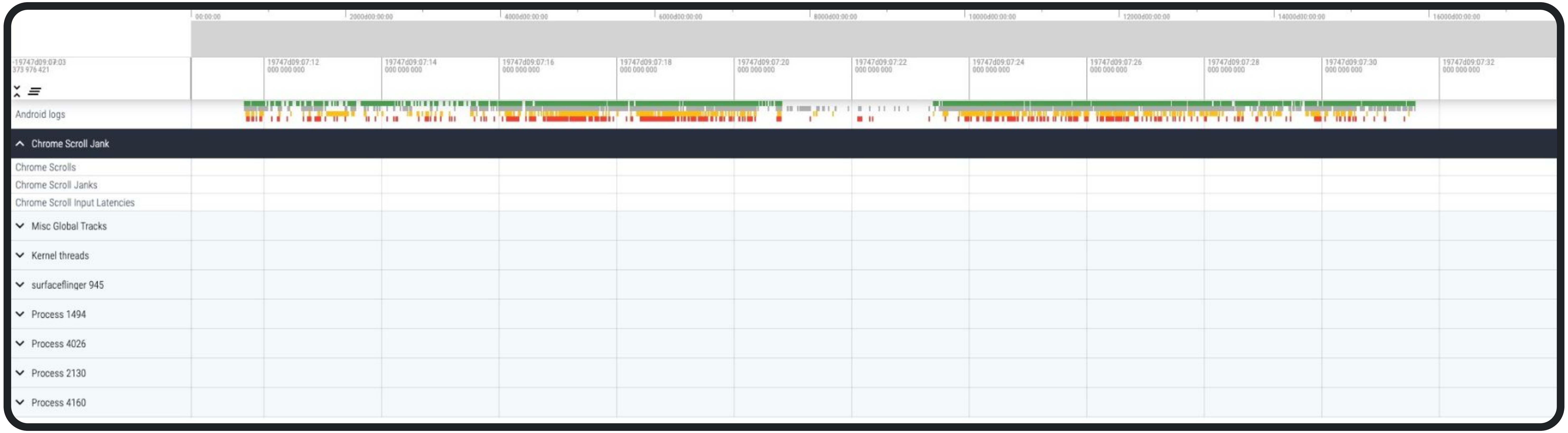
> adb shell cat /sys/kernel/tracing/current_tracer

> adb shell cat /sys/kernel/debug/tracing/current_tracer # older kernel may still use debugfs

Run either of the following to set the current_tracer to nop:

> adb shell echo “nop > /sys/kernel/tracing/current_tracer”

> adb shell echo “0 > /sys/kernel/tracing/tracing_on"



# Common Pitfalls

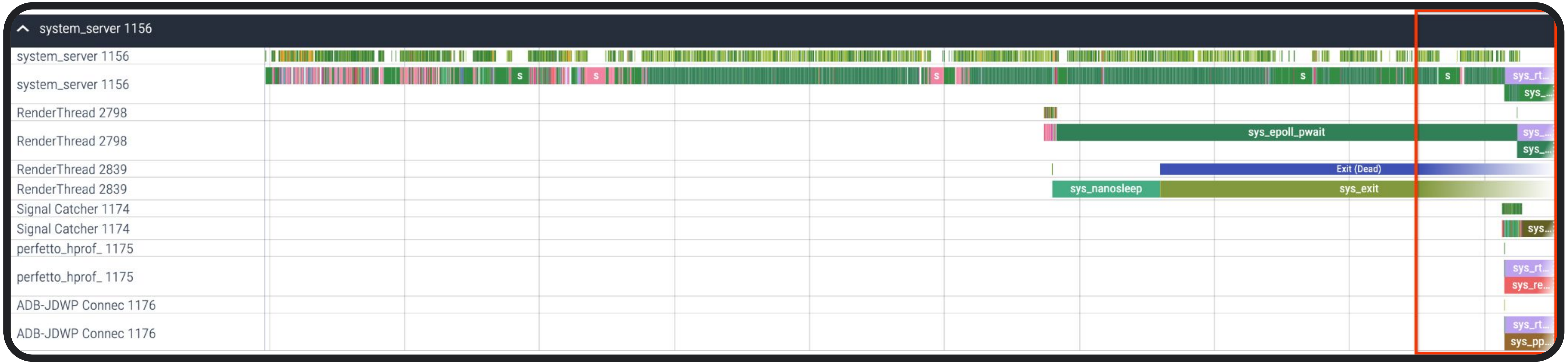
## Insufficient Buffer Size

Another common pitfall is insufficient data buffer size. Increasing buffer size may alleviate scenarios in which key CUJ data is dropped.

## Excessive event collection

If too many events are being collected, there are some that can be dropped to avoid trampling the output trace file. Including sys_enter and sys_exit will lead to all system calls being logged. The below trace demonstrates this, where the tracks do not terminate.

```
data_sources: {
  config {
    name: "linux.ftrace"
    target_buffer: 2
    ftrace_config {
       # Do not include the below:
       ftrace_events: "raw_syscalls/sys_enter"
       ftrace_events: "raw_syscalls/sys_exit"
```



# Trace Analysis

# Trace Analysis Key Steps

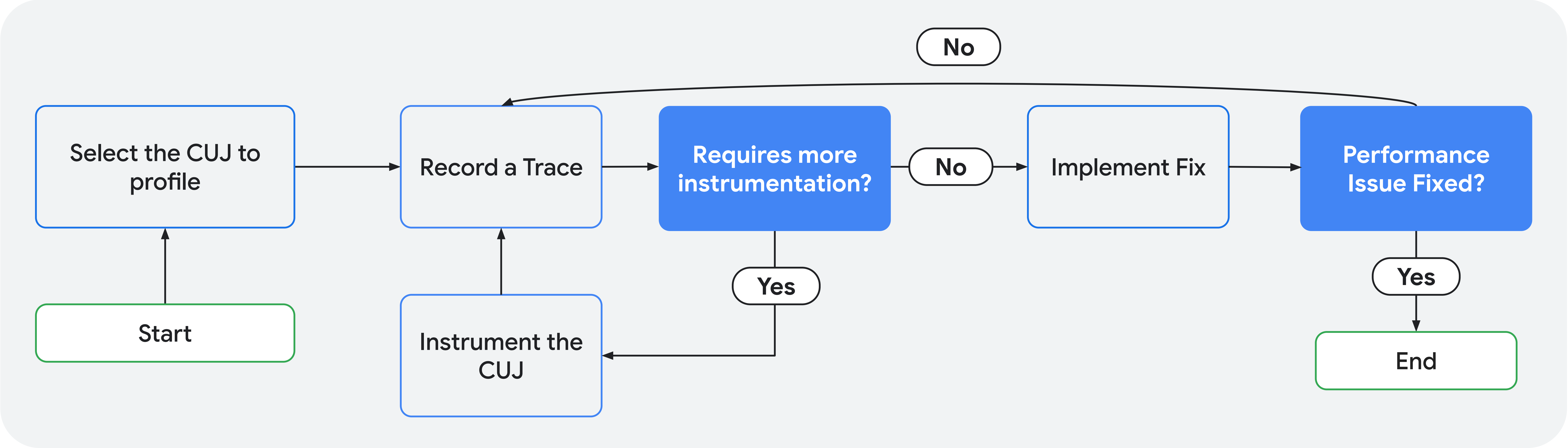
**Summary:**

1. Narrow the search space: one can achieve this by determining the beginning and ending points via Android system logs or atrace logs.
2. Inspect CPU, memory tracks, etc: This will help identify symptoms of a regression so that the analysis window can be tightened.
3. Understand context: After capturing a smaller window, it is possible to understand what actions are being performed. (Ex. What is occurring during at this point in the user switch lifecycle?)
4. Identify Culprit Process: Given context, it is possible to visualize offending processes in the trace. Adding more logging via atrace will also allow one to trace points in a codepath.
5. Analyze thread-level interactions: Looking at markers such as thread state and binder transactions during the window will allow one to make informed hypotheses.

# Trace Analysis Summary

**Sample flow illustrating:**

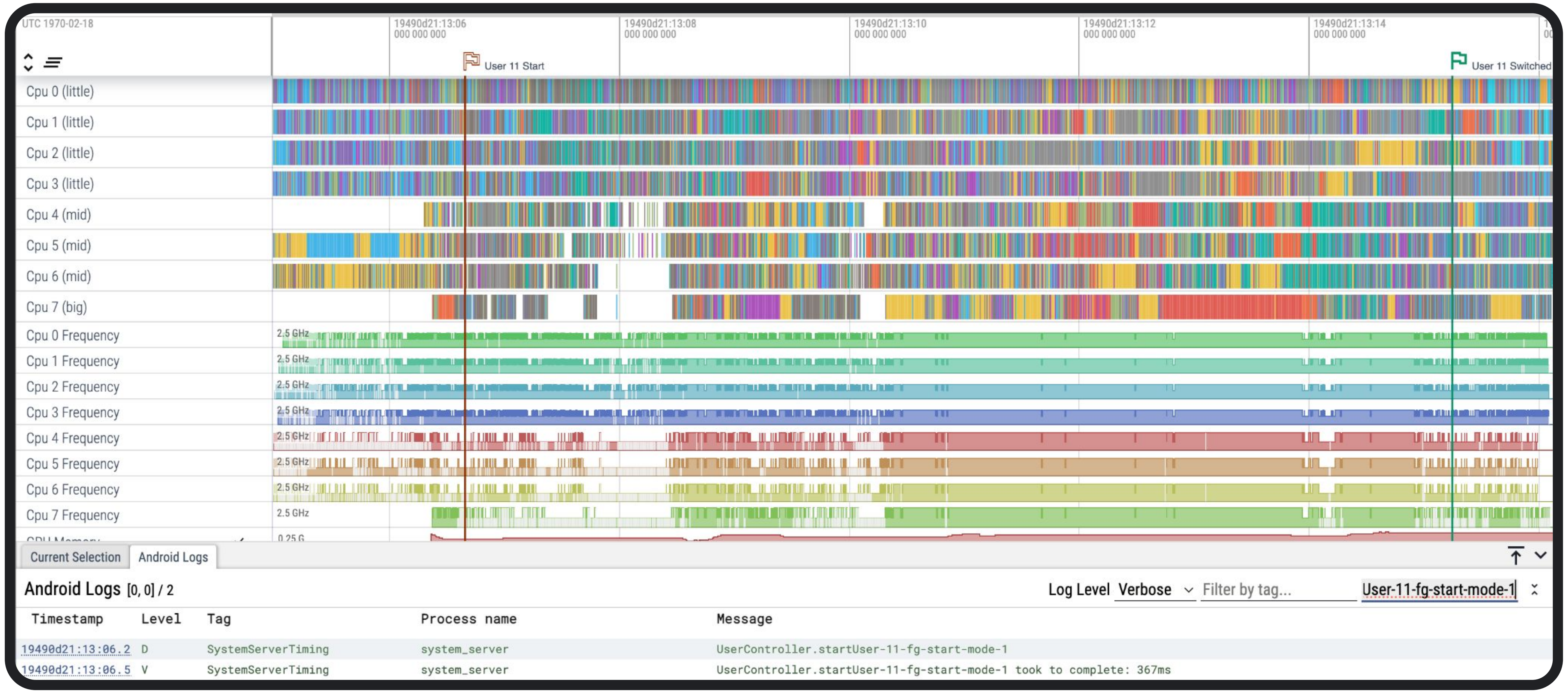
1. CUJ profiling
2. Trace recording
3. CUJ instrumenting
4. Performance Fixes



# Trace Analysis Walkthrough

Looking at a trace can be overwhelming. There are several key steps to help narrow down a problem area to root cause a performance issue. A trace analysis walkthrough will help guide investigations. Initially a trace was collected that captured a user switch from user 10 to user 11.

1. **Narrow the search space:** Use Android logs to identify key starting and stopping points. In this case flag ***UserController.startUser-11-fg-start-mode-1*** and ***onCompletedEventUser 11*** act as the stop and start points.



# Trace Analysis Walkthrough

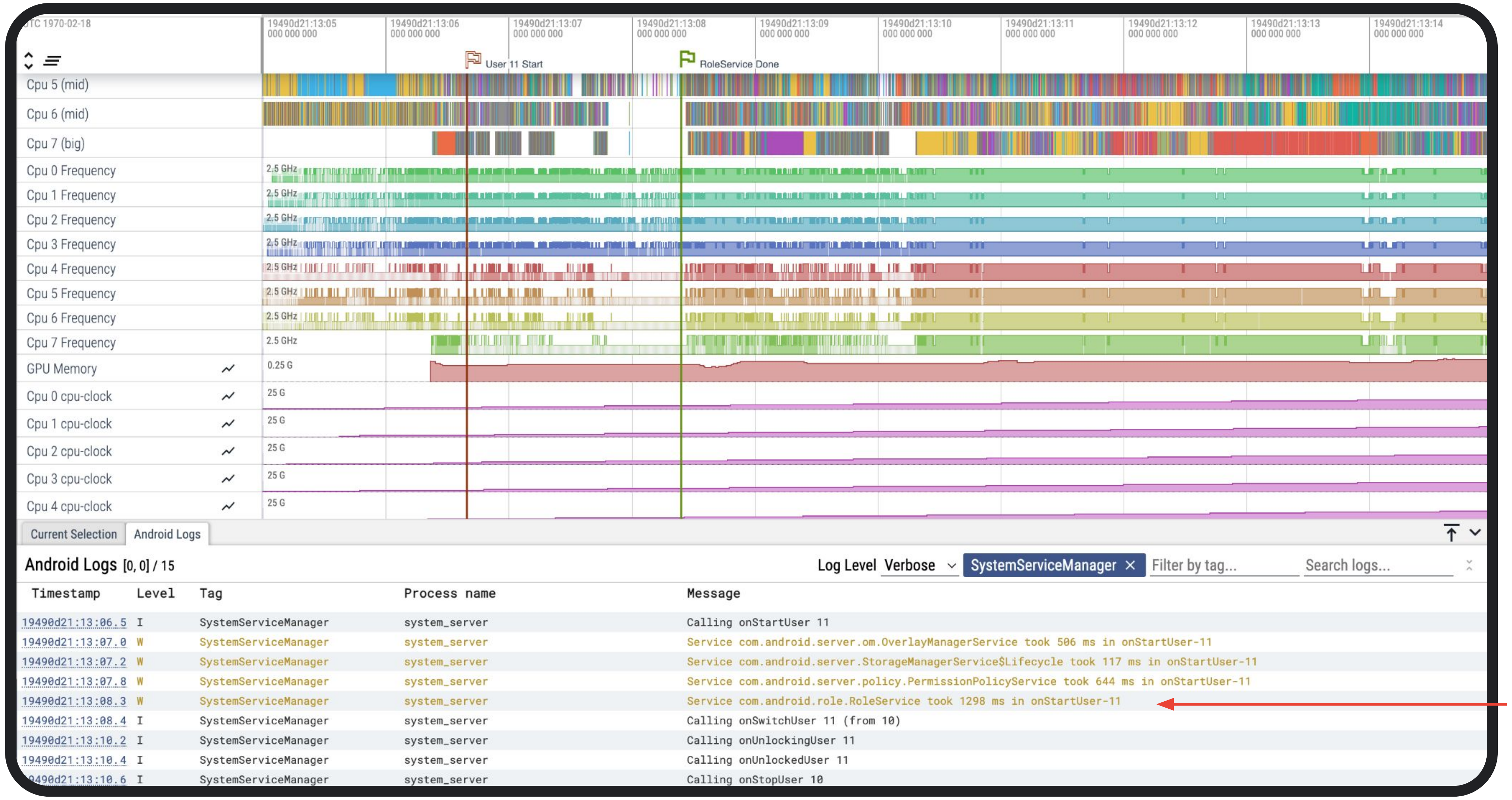
2. **Inspect CPU and memory tracks:** In this case, it is apparent that there are big and gold cores being underutilized during the user switch.

# Trace Analysis Walkthrough

After identifying the area of interest, it is necessary to zoom in on events occurring during this underutilization.
This can be achieved by zeroing in on log events or processes that have significant activity during that period.

# Trace Analysis Walkthrough

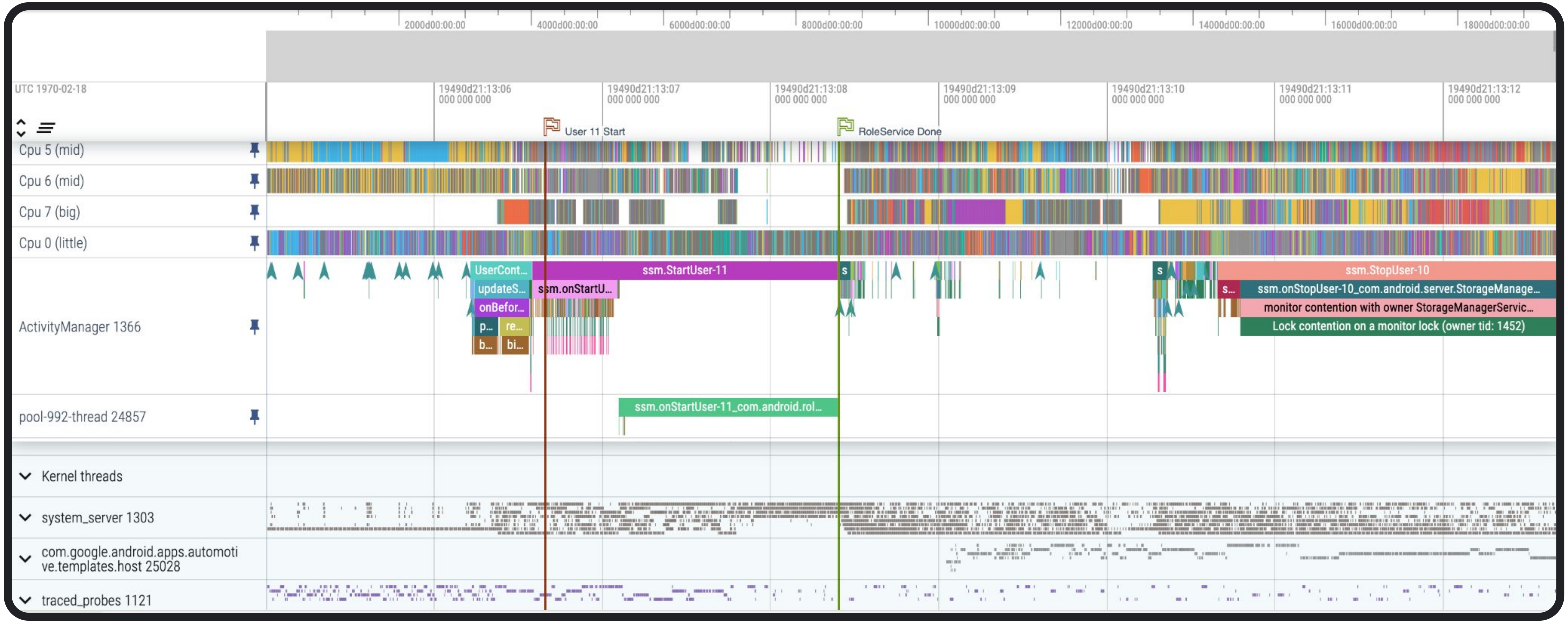
**3. Understand Context:** Here, only 4 out of 8 cores are being utilized, which undoubtedly contribute to a prolonged user switch. This idleness appears early on in the user switch when user 11 is being started. SystemServiceManager is responsible for starting system services during user initialization. SystemServiceManager will wait until all services are created. It is clear that com.android.role.RoleService is the last service to be initialized and also requires the most time.




com.android.role.RoleService onUserStarting method returns

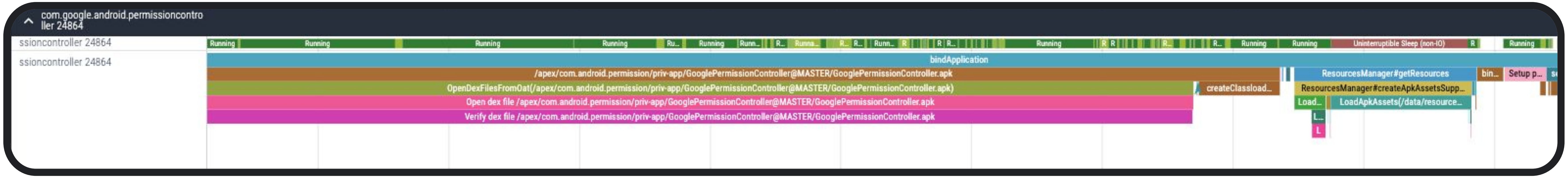
# Trace Analysis Walkthrough

**4. Identify Culprit Process:** Android logs allow us to identify that the com.google.android.permissioncontroller process is largely responsible for starting the RoleService. Zooming in further, it is apparent that the RoleControllerService thread handles majority of the initialization.



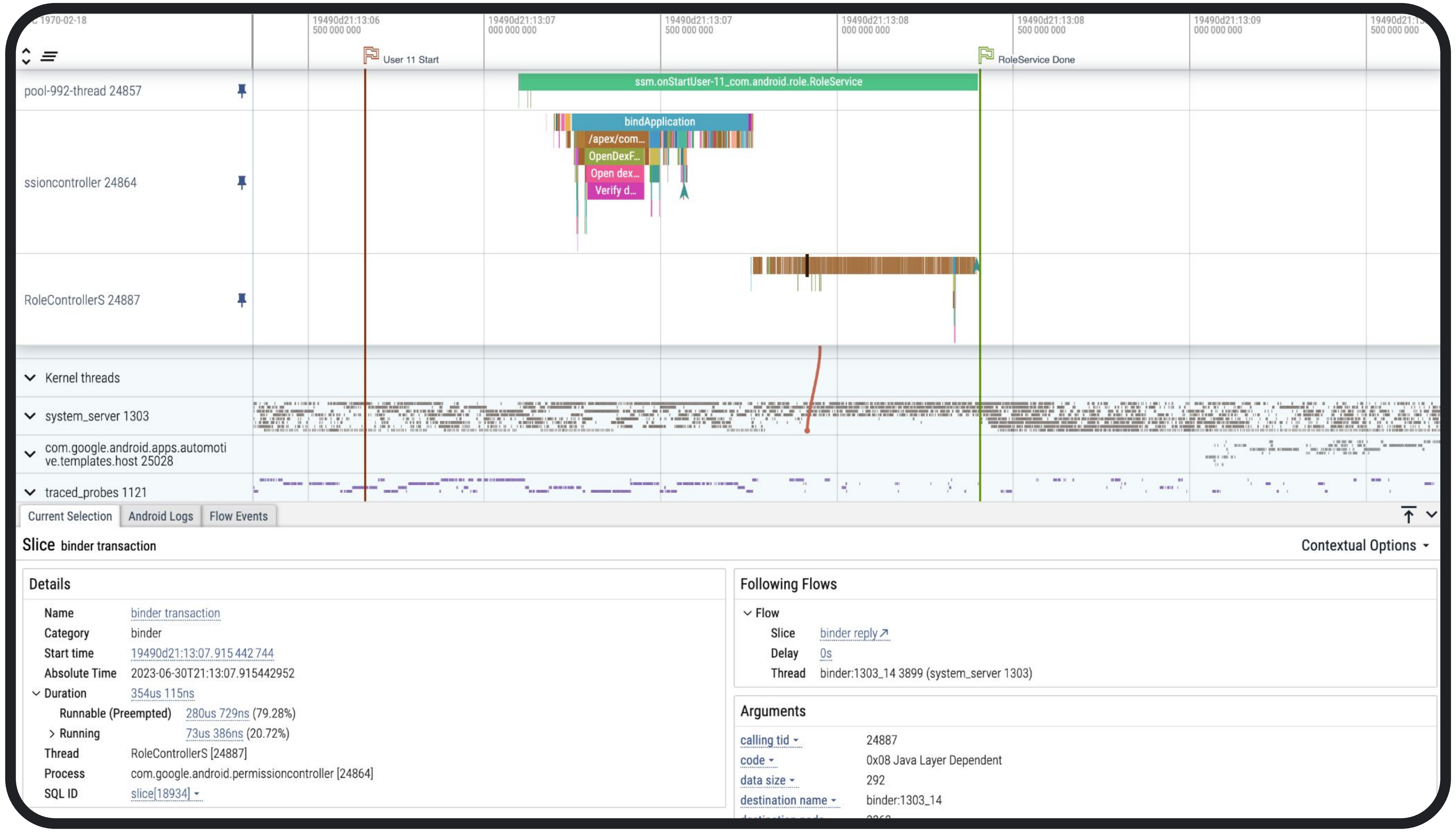
# Trace Analysis Walkthrough

**5. Look for thread level interactions:** Inspecting the com.google.android.permissioncontroller process and its threads may reveal further details about thread state. For example, a long period of uninterruptible sleep could indicate heavy I/O usage. In this case there is nothing that indicates anything out of the ordinary.



# Trace Analysis Walkthrough

**5.** Analyzing the RoleControllerService thread reveals that there is an excess of costly binder transactions occurring. It is clear that these inter-process communication transactions are the cause of the slow down.

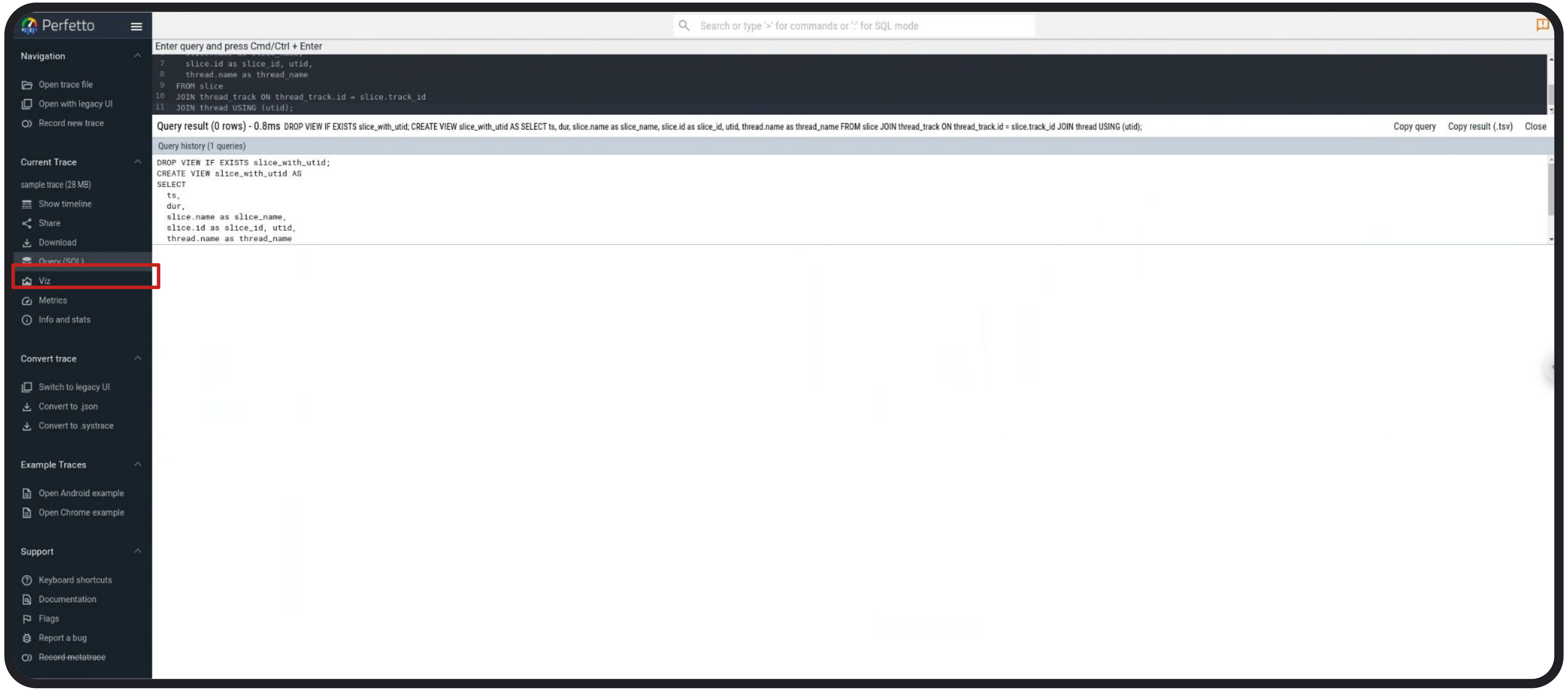


# Advanced Topics: SQL Queries

# Data Mining Using SQL Queries

Beyond visually inspecting system issues via the Perfetto UI, it is also possible to gain a deeper understanding through SQL queries.

One can access SQL queries via the below interface:



# Data Mining Using SQL Queries

One common example is collecting the CPU Time for slices. The first step is to build a table that links slices with their thread state.

```
DROP VIEW IF EXISTS slice_with_utid;
CREATE VIEW slice_with_utid AS
SELECT
  ts,
  dur,
  slice.name as slice_name,
  slice.id as slice_id, utid,
  thread.name as thread_name
FROM slice
JOIN thread_track ON thread_track.id = slice.track_id
JOIN thread USING (utid);

DROP TABLE IF EXISTS slice_thread_state_breakdown;
CREATE VIRTUAL TABLE slice_thread_state_breakdown
USING SPAN_LEFT_JOIN(
  slice_with_utid PARTITIONED utid,
  thread_state PARTITIONED utid
);
```

# Data Mining Using SQL Queries

From the previous table, the CPU time for each slide in a Running state can be listed.

```
SELECT slice_id, slice_name, SUM(dur) AS cpu_time
FROM slice_thread_state_breakdown
WHERE state = 'Running'
GROUP BY slice_id;
```


Enter query and press Cmd/Ctrl + Enter

```
SELECT slice_id, slice_name, SUM(dur) AS cpu_time
FROM slice_thread_state_breakdown
WHERE state = 'Running'
GROUP BY slice_id;
```

Query result (10000 rows) - 686.1ms SELECT slice_id, slice_name, SUM(dur) AS cpu_time FROM slice_thread_state_breakdown WHERE state = 'Running' GROUP BY slice_id; Copy query Copy result (.tsv) Close

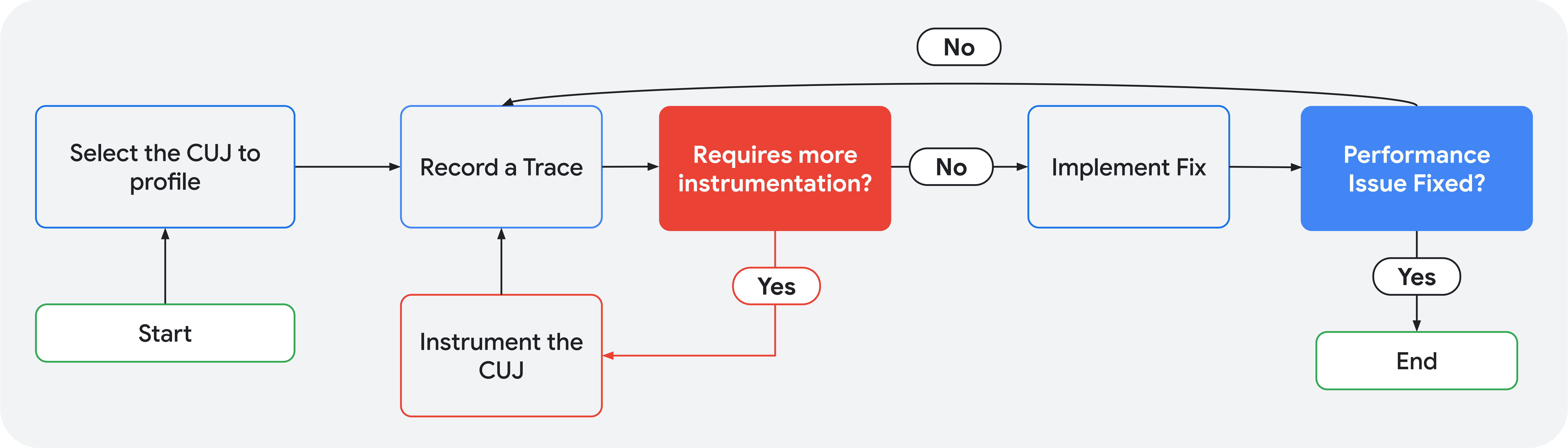
| slice_id | slice_name | cpu_time |
|---|---|---|
| 3 | Contending for pthread mutex | 28594 |
| 12 | sys_epoll_pwait | 11146 |
| 46 | sys_ioctl | 17916 |
| 47 | sys_read | 4479 |
| 55 | sys_ioctl | 39323 |
| 56 | binder transaction async | 0 |
| 57 | sys_getuid | 1042 |

# Making Debugging Easier

# Make App Debugging Easier

**Performance Analysis Flow:**

1. CUJ profiling
2. Trace recording
3. **CUJ instrumenting**
4. Performance Fixes



# Make App Debugging Easier

A powerful feature that can help with debugging is adding atrace logs that will appear in Perfetto.

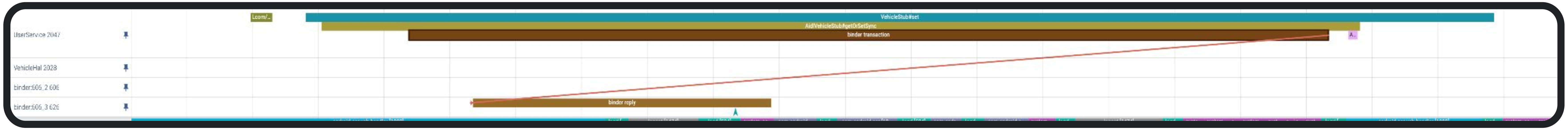
Java applications can add trace logs using android.os.Trace.

Native applications can add trace logs using ATrace_beginSection() / ATrace_setCounter() defined in <trace.h>

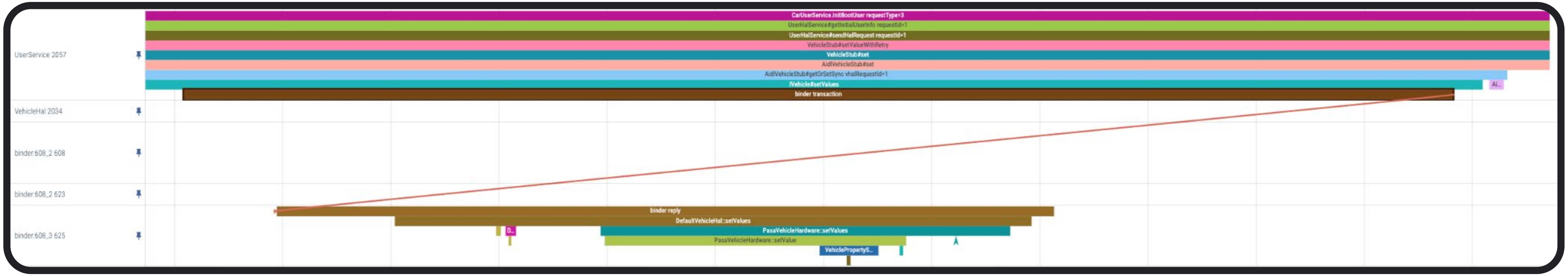
```
Trace.traceBegin(TRACE_TAG, "Class#method");
...
Trace.traceEnd(TRACE_TAG);
```

# Make App Debugging Easier

**Before:**




**After:**



# Performance Tuning

# Performance Tuning After Boot

## What is Performance Tuning?

WIth the aid of Perfetto, it is possible to identify performance issues and analyze their root cause. The next step is to implement solutions to solve these issues. One of the ways to achieve this is by iteratively tuning the performance of the system.

## Post Boot Tuning

One of the opportunities for performance tuning is during post boot. After boot complete, there is heavy resource contention as multiple applications attempt to perform initialization. Classically, this is known as the Thundering Herd Problem, where lack of system resources leads to degraded performance. There is opportunity to both improve memory and CPU usage during the critical window after boot complete.

# Memory Tuning

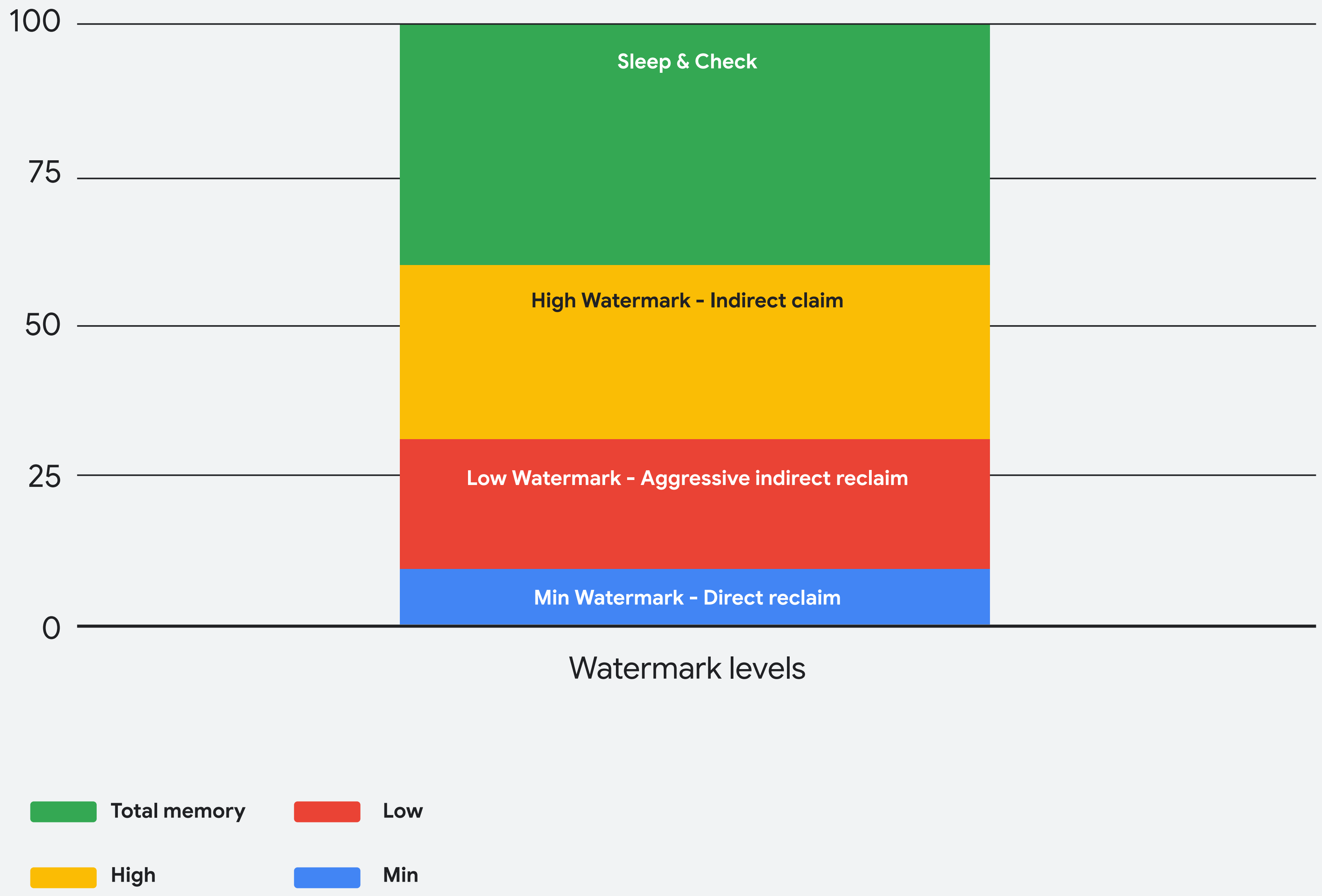
# Kernel kswapd

- kswapd is a Kernel task to manage available free memory.
- Kernel uses 3 watermarks per memory zone to track pressure
  - Min, Low, and High
- When free memory <= Low and > Min
  - kswapd performs asynchronous/indirect memory reclaim.
- When free memory <= Min
  - kswapd performs synchronous/direct memory reclaim.
  - System becomes unstable
- When free memory >= High
  - kswapd temporarily enters sleep state.
  - Periodically checks the memory pressure.

## Reclaim types

- Indirect memory reclaim
  - Increases kswapd CPU usage.
  - May slow down other processes depending on the CPU & memory pressure.
- Direct memory reclaim
  - All new allocations will be blocked until kswapd frees up memory up to min watermark.

### Kswapd watermark levels



# kswapd tuning

Kernel knobs for tuning kswapd behavior:

- **/proc/sys/vm/swappiness** - Defines the aggressiveness of swapping out memory pages of inactive processes.
    - **Range:** 0-100 **Default**: 60
    - High values can cause Kernel to swap out processes even when enough memory is available.
    - Low values can cause Kernel to not swap out processes even when the available memory is low.
    - Recommendation:
        - Devices with high physical memory - use lower swappiness values.
        - Devices with low  physical memory - use higher swappiness values.
- **/proc/sys/vm/watermark_scale_factor** - Used to scale the buffer spaces between memory zone watermarks.
    - **Range:** 0 - 1000 **Default:** 10
        - 10 means buffer space is 0.1% of available memory.
        - 1000 means buffer space is 10% of available memory.
    - Low values can cause too much direct reclaim or kswapd not freeing up enough memory in a single pass.
    - High values can cause kswapd to free up more memory than needed.
- **/proc/sys/vm/min_free_kbytes** - Amount of free memory kept in reserve at all times. Defines min watermark across all memory zones.
    - Recommended value range: 1% - 2% of total system memory.
    - High values can scale up watermark buffer spaces leading to
        - kswapd freeing too much memory than needed.
        - Frequent kswapd invocation causing CPU contention to spike.
    - Low values can cause kswapd to not free up enough memory leading to
        - System slowdown
        - Hangs/crashes
        - Memory fragmentation

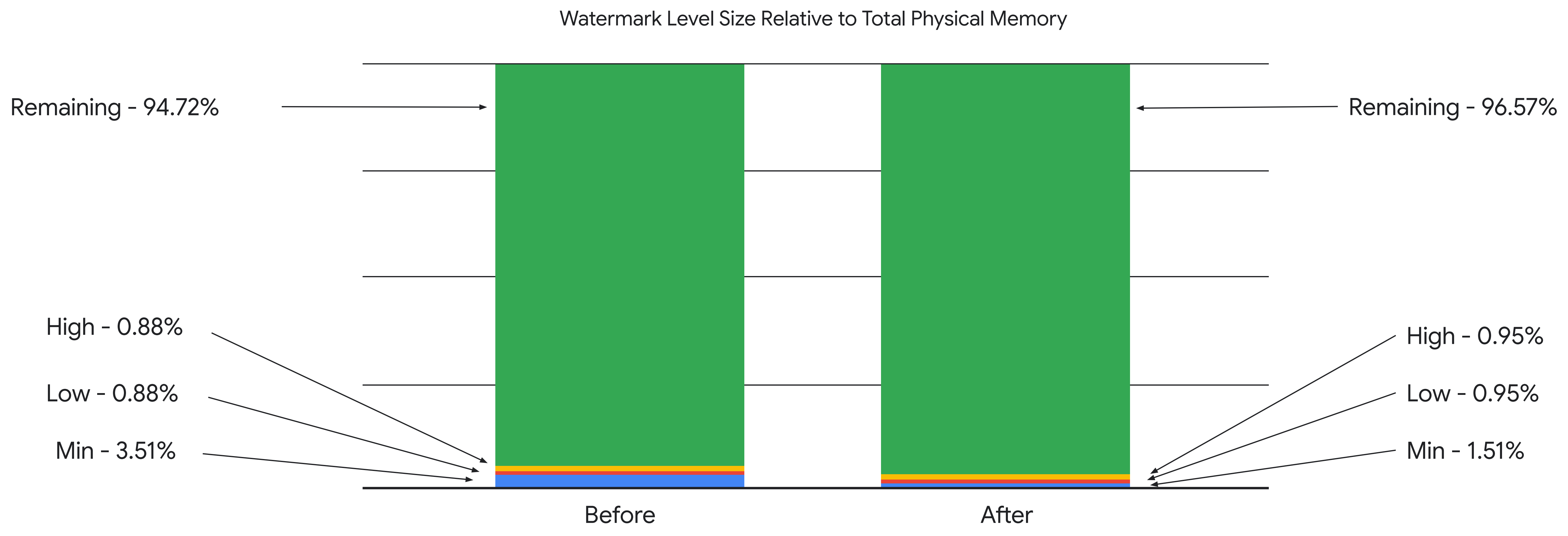
# kswapd tuning example

**Before**

- **/proc/sys/vm/watermark_scale_factor** - 1
- **/proc/sys/vm/min_free_kbytes** - 144 MiB
- **/proc/sys/vm/swappiness** - 60

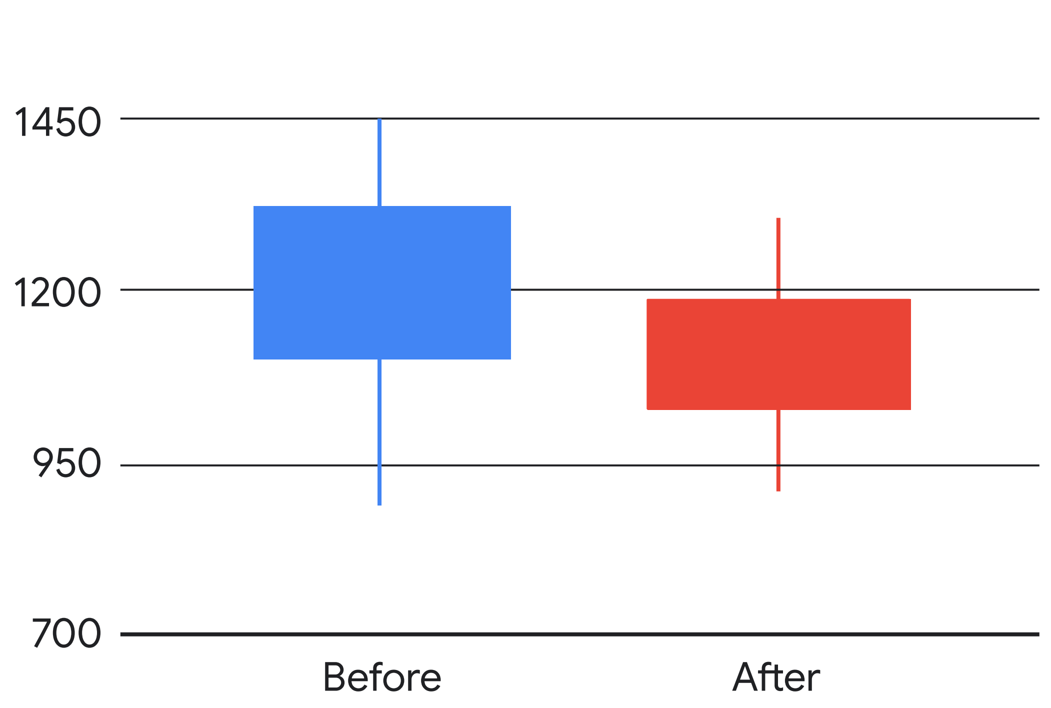
**After**

- **/proc/sys/vm/watermark_scale_factor** - 109
- **/proc/sys/vm/min_free_kbytes** - 60 MiB
- **/proc/sys/vm/swappiness** - 60

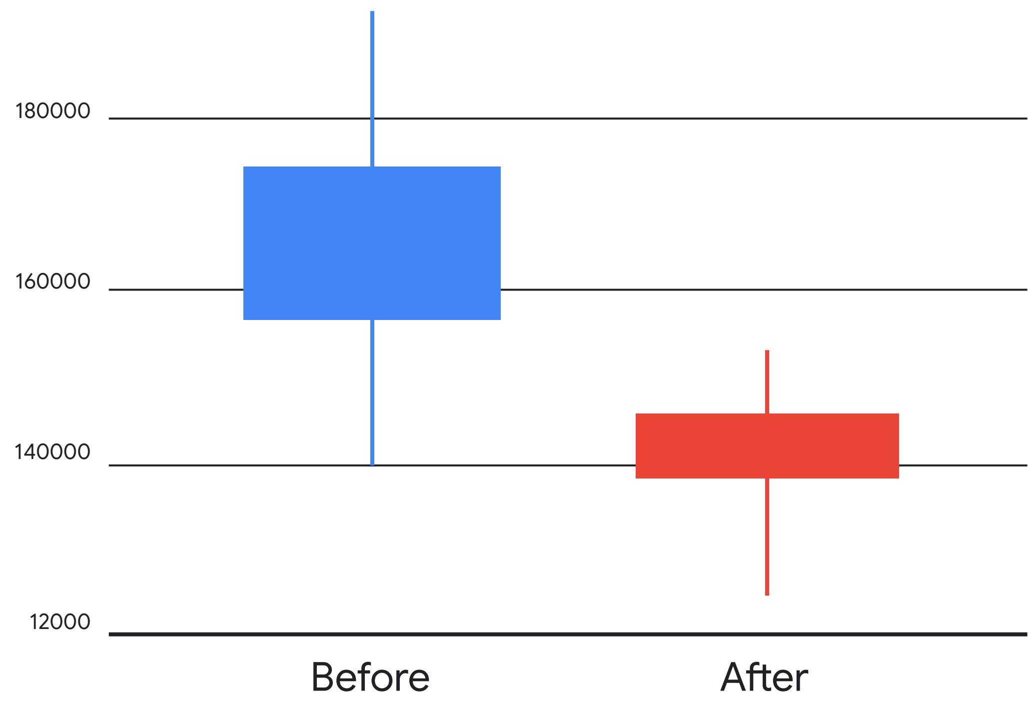


# kswapd tuning example

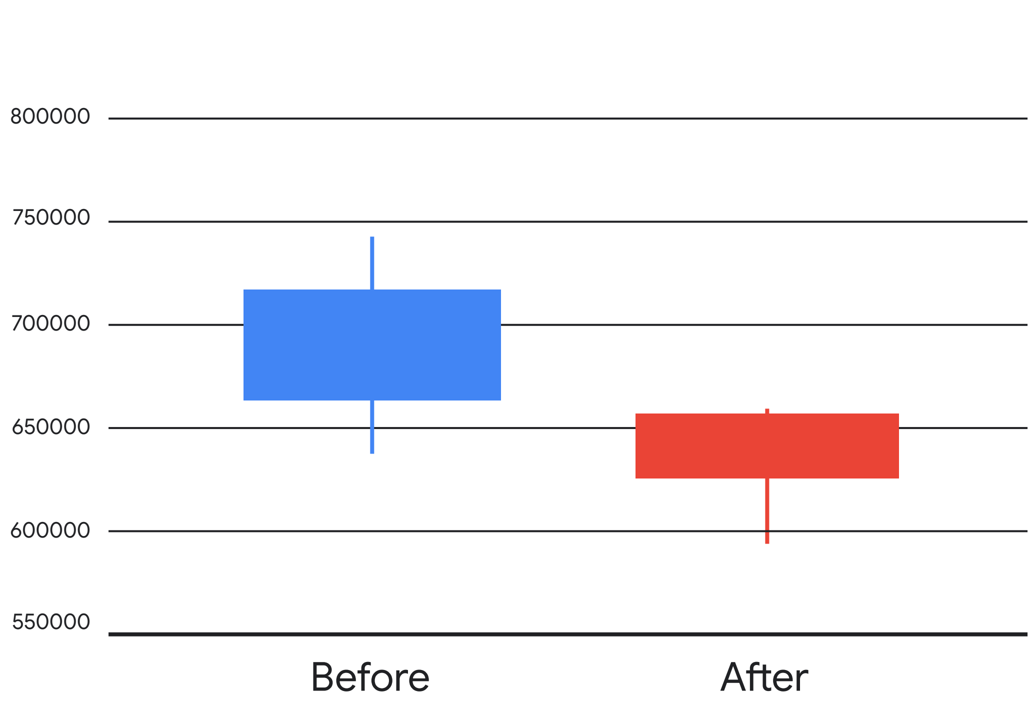
### Kswapd cpu time millis




### Working set refault file




### Page stolen by kswapd



# Dex Optimization

# Dex Optimization

## What is Dex Optimization?

By default in Android, apps are executed in interpreted mode. Dex optimization allows for compilation of selected code paths to machine code, which accelerates code execution.

## Why is this Important?

Background dex optimization already occurs regularly as users interact with their apps. However, after initial installation, boot time performance may be degraded if all apps run in interpreted mode. There is an opportunity to compile key apps ahead of time to greatly reduce boot times.

## Which Apps should be Dex Pre-Optimized?

When apps run in interpreted mode, they will also undergo JIT compilation. Post-boot, a high level of JIT compilation will lead to a high degree of CPU contention, further exacerbating degraded startup times. Identifying processes with high JIT CPU time will indicate apps that can be dex pre-optimized, forgoing JIT entirely.

# Dex Optimization

## Heavy JIT Compilation in Perfetto

JIT compilation activity can be visualized via Perfetto.

Within a specific process, one can identify the JIT thread pool tracks. In this example, Car Assistant is displaying a large number of JIT compilation events.

# Dex Optimization

## Perfetto Query for Top Processes with the Most JIT CPU Time

Using the query will allow one to obtain a table as shown below:

| total_duration | instances | prefix_name | |
|---|---|---|---|
| 5936808097 | 12608 | JIT | com.google.android.carassistant |
| 4889113085 | 9984 | JIT | com.google.android.apps.map |
| 4597115883 | 6785 | JIT | com.android.vendin |
| 2342370393 | 3474 | JIT | com.google.android.apps.geo.automotive.adas |
| 1440327723 | 2422 | JIT | com.android.vending |
| 1078105791 | 2169 | JIT | com.google.android.tt |

```
INCLUDE PERFETTO MODULE slices.slices;

DROP VIEW IF EXISTS interesting_slices_d0;
CREATE VIEW interesting_slices_d0 AS
select id as slice_id, ts, dur, name, track_id, track_name, thread_name, utid, tid,
process_name, upid, pid from _slice_with_thread_and_process_info where
depth=0;

DROP TABLE IF EXISTS slice_thread_state_breakdown;
CREATE VIRTUAL TABLE slice_thread_state_breakdown
USING SPAN_LEFT_JOIN(
interesting_slices_d0 PARTITIONED utid,
thread_state PARTITIONED utid
);

SELECT sum(dur) total_duration, count(*) instances, substr(name, 0,
IIF(instr(name, ' ') > 0, instr(name, ' '), IIF(instr(name, ',') > 0, instr(name, ','),
length(name)))) as prefix_name,
substr(process_name, 0, IIF(instr(process_name, ':') > 0, instr(process_name, ':'),
length(process_name))) as process_name_prefix FROM
slice_thread_state_breakdown
WHERE state = 'Running' and prefix_name = "JIT"
group by prefix_name, process_name_prefix
order by total_duration desc;
```

# Dex Optimization Configuration

## How to Configure Dex Pre-Optimization

For more details refer to https://source.android.com/docs/core/runtime/configure#build_options.

Add packages to the following makefile configuration:

```
PRODUCT_DEXPREOPT_SPEED_APPS += \
    MapsCarPrebuilt \
```

# Dex Optimization Configuration

## How to Verify that an App is Dex Pre-Opted

Run the following ADB command:

```
$ adb shell pm art dump com.google.android.apps.maps
# Older releases may need to use this command instead:
$ adb shell dumpsys package dexopt | grep -i
com.google.android.apps.maps -A 2
```

The following output indicates Google Maps is executed in interpreted mode:

```
[com.google.android.apps.maps]
path:
/system/product/priv-app/MapsCarPrebuilt/MapsCarPrebuilt.apk
x86_64: [status=verify] [reason=prebuilt]
```

The following output indicates that Google Maps was dex pre-opted:

```
[com.google.android.apps.maps]
  path: /product/priv-app/MapsCarPrebuilt/MapsCarPrebuilt.apk
    x86_64: [status=speed] [reason=prebuilt] [primary-abi]
```

# Further Materials / Important Links

## Summary of useful information per section:

**Trace Configuration:**

https://perfetto.dev/docs/reference/trace-config-proto

**How to Collect a Perfetto Trace:**

https://perfetto.dev/docs/quickstart/android-tracing

**Android Boot Tracing:**

https://perfetto.dev/docs/case-studies/android-boot-tracing

**CPU Tracks:**

https://perfetto.dev/docs/data-sources/cpu-scheduling

**Memory Tracks:**

https://perfetto.dev/docs/data-sources/memory-counters

**Atrace Logging:**

https://perfetto.dev/docs/data-sources/atrace

# Thank you

Google
Automotive
Partner
Bootcamp