Evolving from Rule-based Classifier: Machine Learning Powered Auto Remediation in Netflix Data Platform

netflixtechblog.com/evolving-from-rule-based-classifier-machine-learning-powered-auto-remediation-in-netflix-data-039d5efd115b

Netflix Technology Blog March 4, 2024

Learn about Netflix's world class engineering efforts, company culture, product developments and more.

by Binbing Hou, Stephanie Vezich Tamayo, Xiao Chen, Liang Tian, Troy Ristow, Haoyuan Wang, Snehal Chennuru, Pawan Dixit

This is the first of the series of our work at Netflix on leveraging data insights and Machine Learning (ML) to improve the operational automation around the performance and cost efficiency of big data jobs. Operational automation–including but not limited to, auto diagnosis, auto remediation, auto configuration, auto tuning, auto scaling, auto debugging, and auto testing–is key to the success of modern data platforms. In this blog post, we present our project on Auto Remediation, which integrates the currently used rule-based classifier with an ML service and aims to automatically remediate failed jobs without human intervention. We have deployed Auto Remediation in production for handling memory configuration errors and unclassified errors of Spark jobs and observed its efficiency and effectiveness (e.g., automatically remediating 56% of memory configuration errors and saving 50% of the monetary costs caused by all errors) and great potential for further improvements.

At Netflix, hundreds of thousands of workflows and millions of jobs are running per day across multiple layers of the big data platform. Given the extensive scope and intricate complexity inherent to such a distributed, large-scale system, even if the failed jobs account for a tiny portion of the total workload, diagnosing and remediating job failures can cause considerable operational burdens.

For efficient error handling, Netflix developed an error classification service, called Pensive, which leverages a rule-based classifier for error classification. The rule-based classifier classifies job errors based on a set of predefined rules and provides insights for schedulers to decide whether to retry the job and for engineers to diagnose and remediate the job failure.

However, as the system has increased in scale and complexity, the rule-based classifier has been facing challenges due to its limited support for operational automation, especially for handling memory configuration errors and unclassified errors. Therefore, the operational cost increases linearly with the number of failed jobs. In some cases–for

example, diagnosing and remediating job failures caused by Out-Of-Memory (OOM) errors–joint effort across teams is required, involving not only the users themselves, but also the support engineers and domain experts.

To address these challenges, we have developed a new feature, called *Auto Remediation*, which integrates the rule-based classifier with an ML service. Based on the classification from the rule-based classifier, it uses an ML service to predict retry success probability and retry cost and selects the best candidate configuration as recommendations; and a configuration service to automatically apply the recommendations. Its major advantages are below:

- Instead of completely deprecating the current rule-based classifier, Auto Remediation integrates the classifier with an ML service so that it can leverage the merits of both: the rule-based classifier provides static, deterministic classification results per error class, which is based on the context of domain experts; the ML service provides performance- and cost-aware recommendations per job, which leverages the power of ML. With the integrated intelligence, we can properly meet the requirements of remediating different errors.
- The pipeline of classifying errors, getting recommendations, and applying recommendations is fully automated. It provides the recommendations together with the retry decision to the scheduler, and particularly uses an online configuration service to store and apply recommended configurations. In this way, no human intervention is required in the remediation process.
- Auto Remediation generates recommendations by considering both performance (i.e., the retry success probability) and compute cost efficiency (i.e., the monetary costs of running the job) to avoid blindly recommending configurations with excessive resource consumption. For example, for memory configuration errors, it searches multiple parameters related to the memory usage of job execution and recommends the combination that minimizes a linear combination of failure probability and compute cost.

These advantages have been verified by the production deployment for remediating Spark jobs' failures. Our observations indicate that Auto Remediationcan successfully remediate about 56% of all memory configuration errors by applying the recommended memory configurations online without human intervention; and meanwhile reduce the cost of about 50% due to its ability to recommend new configurations to make memory configurations successful and disable unnecessary retries for unclassified errors. We have also noted a great potential for further improvement by model tuning (see the section of Rollout in Production).

Rule-based Classifier: Basics and Challenges

Basics

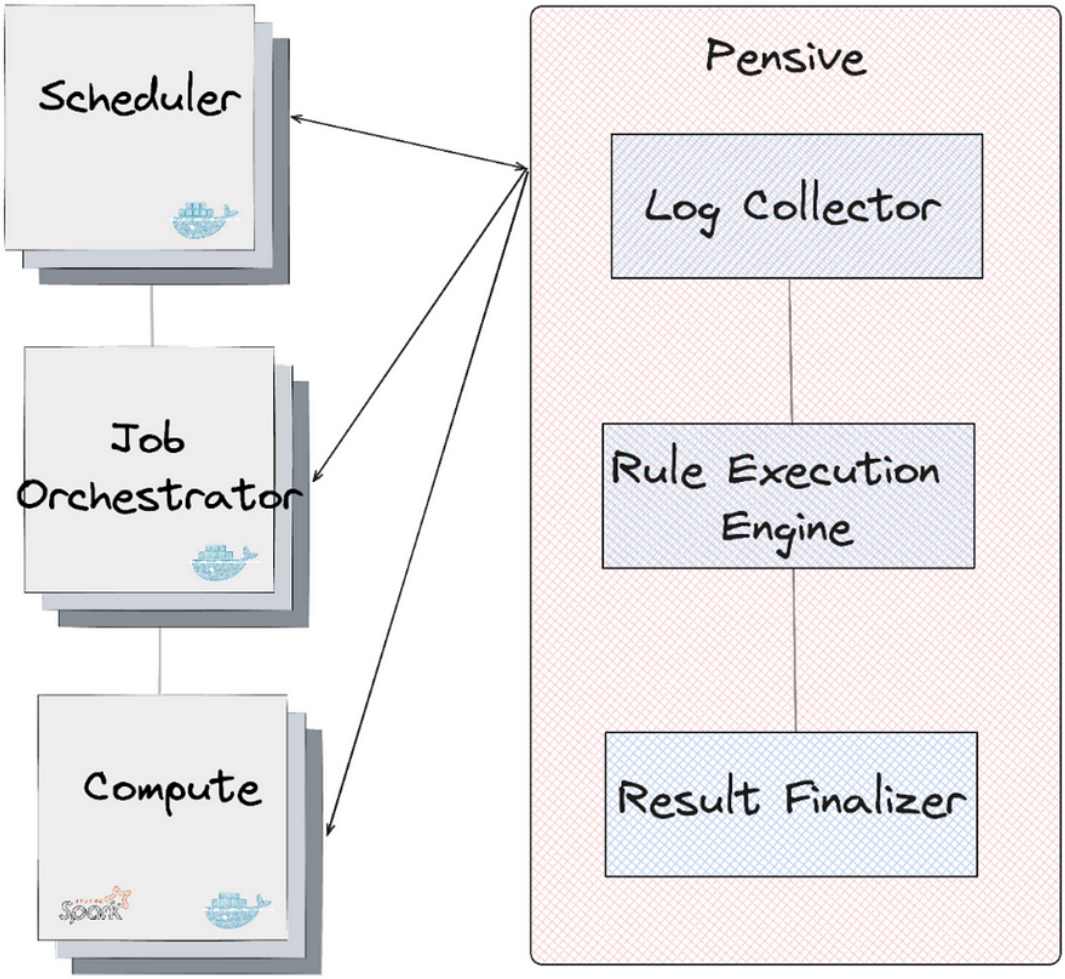


Figure 1. Pensive in Netflix Data Platform

Figure 1 illustrates the error classification service, i.e., Pensive, in the data platform. It leverages the rule-based classifier and is composed of three components:

- is responsible for pulling logs from different platform layers for error classification (e.g., the scheduler, job orchestrator, and compute clusters).
- is responsible for matching the collected logs against a set of predefined rules. A rule includes (1) the name, source, log, and summary, of the error and whether the error is restartable; and (2) the regex to identify the error from the log. For example, the rule with the name SparkDriverOOM includes the information indicating that if the stdout log of a Spark job can match the regex , then this error is classified to be a user error, not restartable.
- is responsible for finalizing the error classification result based on the matched rules. If one or multiple rules are matched, then the classification of the first matched rule determines the final classification result (the rule priority is determined by the rule ordering, and the first rule has the highest priority). On the other hand, if no rules are matched, then this error will be considered unclassified.

Challenges

While the rule-based classifier is simple and has been effective, it is facing challenges due to its limited ability to handle the errors caused by misconfigurations and classify new errors:

- The rules-based classifier provides error classification results indicating whether to restart the job; however, for non-transient errors, it still relies on engineers to manually remediate the job. The most notable example is memory configuration errors. Such errors are generally caused by the misconfiguration of job memory. Setting an excessively small memory can result in Out-Of-Memory (OOM) errors while setting an excessively large memory can waste cluster memory resources. What's more challenging is that some memory configuration errors require changing the configurations of multiple parameters. Thus, setting a proper memory configuration requires not only the manual operation but also the expertise of Spark job execution. In addition, even if a job's memory configuration is initially well tuned, changes such as data size and job definition can cause performance to degrade. Given that about 600 memory configuration errors per month are observed in the data platform, timely remediation of memory configuration errors alone requires non-trivial engineering efforts.
- The rule-based classifier relies on data platform engineers to manually add rules for recognizing errors based on the known context; otherwise, the errors will be unclassified. Due to the migrations of different layers of the data platform and the diversity of applications, existing rules can be invalid, and adding new rules requires engineering efforts and also depends on the deployment cycle. More than 300 rules have been added to the classifier, yet about 50% of all failures remain unclassified. For unclassified errors, the job may be retried multiple times with the default retry policy. If the error is non-transient, these failed retries incur unnecessary job running costs.

Evolving to Auto Remediation: Service Architecture

Methodology

To address the above-mentioned challenges, our basic methodology is to integrate the rule-based classifier with an ML service to generate recommendations, and use a configuration service to apply the recommendations automatically:

- We use the rule-based classifier as the first pass to classify all errors based on predefined rules, and the ML service as the second pass to provide recommendations for memory configuration errors and unclassified errors.
- We use an online configuration service to store and apply the recommended configurations. The pipeline is fully automated, and the services used to generate and apply recommendations are decoupled.

Service Integrations

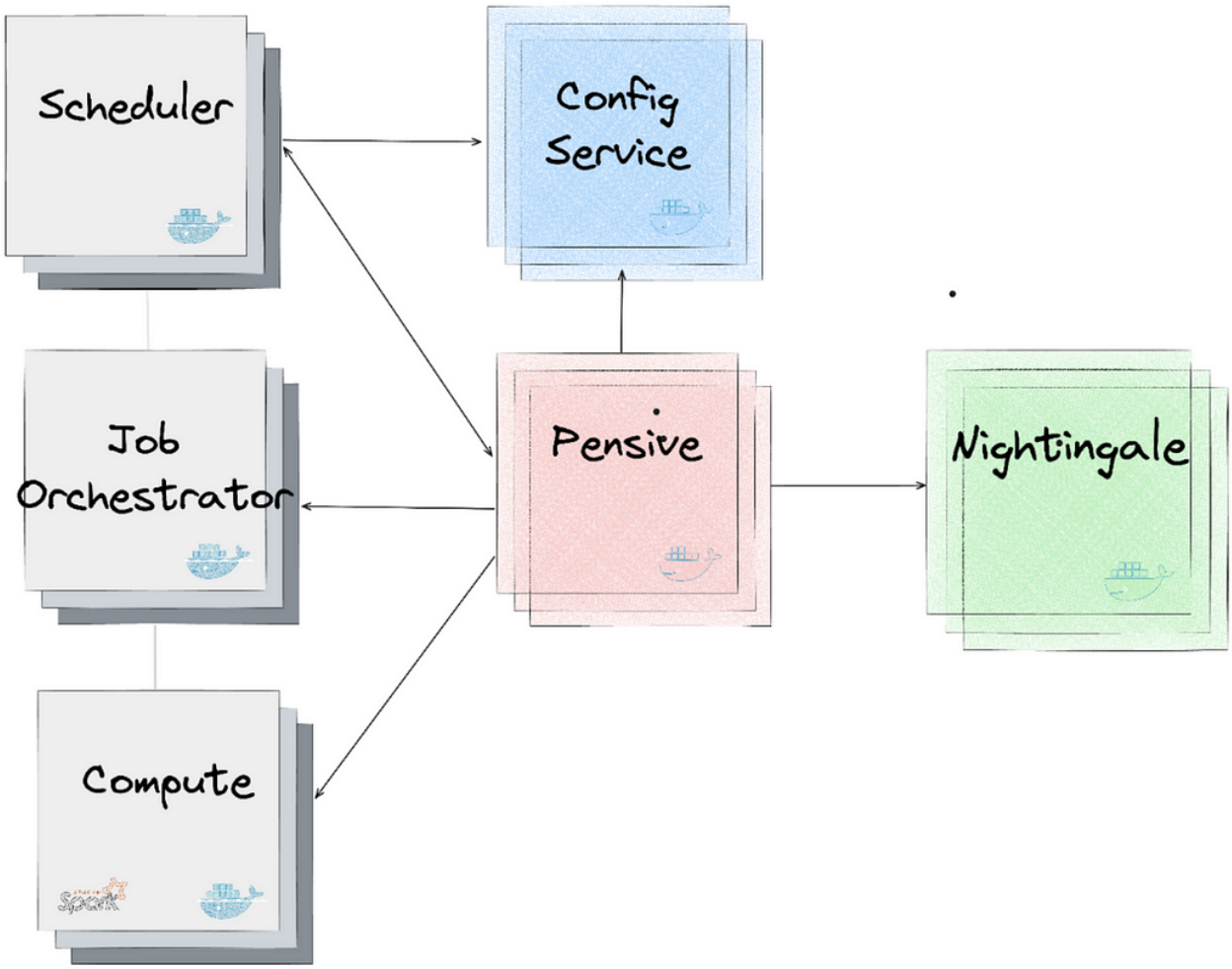


Figure 2. Auto Remediation in Netflix Data Platform

Figure 2 illustrates the integration of the services generating and applying the recommendations in the data platform. The major services are as follows:

- is a service running the ML model trained using and is responsible for generating a retry recommendation. The recommendation includes (1) whether the error is restartable; and (2) if so, the recommended configurations to restart the job.
- is an online configuration service. The recommended configurations are saved in as a JSON patch with a scope defined to specify the jobs that can use the recommended configurations. When calls to get recommended configurations, passes the original configurations to and returns the mutated configurations by applying the JSON patch to the original configurations. can then restart the job with the mutated configurations (including the recommended configurations).
- is an error classification service that leverages the rule-based classifier. It calls to get recommendations and stores the recommendations to so that it can be picked up by to restart the job.
- is the service scheduling jobs (our current implementation is with). Each time when a job fails, it calls to get the error classification to decide whether to restart a job and calls to get the recommended configurations for restarting the job.

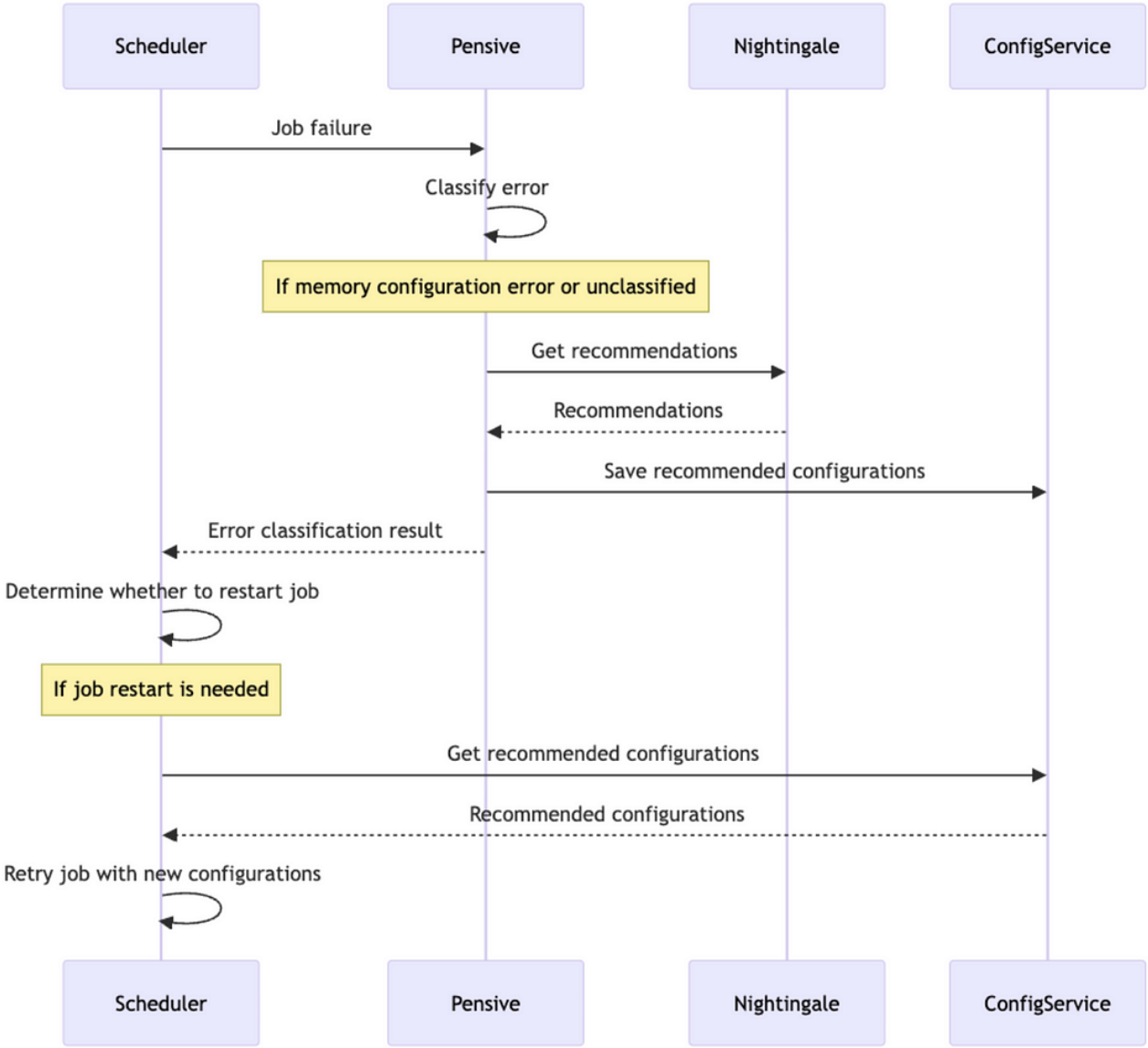


Figure 3. Sequence Diagram of Service Calls with Auto Remediation

Figure 3 illustrates the sequence of service calls with Auto Remediation:

1. Upon a job failure, calls to get the error classification.
2. classifies the error based on the rule-based classifier. If the error is identified to be a memory configuration error or an unclassified error, it calls to get recommendations.
3. With the obtained recommendations, updates the error classification result and saves the recommended configurations to ; and then returns the error classification result to .
4. Based on the error classification result received from determines whether to restart the job.
5. Before restarting the job, calls to get the recommended configuration and retries the job with the new configuration.

Evolving to Auto Remediation: ML Service

Overview

The ML service, i.e., Nightingale, aims to generate a retry policy for a failed job that trades off between retry success probability and job running costs. It consists of two major components:

- that jointly estimates a) probability of retry success, and b) retry cost in dollars, conditional on properties of the retry.
- which explores the Spark configuration parameter space to recommend a configuration which minimizes a linear combination of retry failure probability and cost.

The prediction model is retrained offline daily, and is called by the optimizer to evaluate each candidate set of configuration parameter values. The optimizer runs in a RESTful service which is called upon job failure. If there is a feasible configuration solution from the optimization, the response includes this recommendation, which ConfigService uses to mutate the configuration for the retry. If there is no feasible solution–in other words, it is unlikely the retry will succeed by changing Spark configuration parameters alone–the response includes a flag to disable retries and thus eliminate wasted compute cost.

Prediction Model

Given that we want to explore how retry success and retry cost might change under different configuration scenarios, we need some way to predict these two values using the information we have about the job. Data Platform logs both retry success outcome and execution cost, giving us reliable labels to work with. Since we use a shared feature set to predict both targets, have good labels, and need to run inference quickly online to meet SLOs, we decided to formulate the problem as a multi-output supervised learning task. In particular, we use a simple Feedforward Multilayer Perceptron (MLP) with two heads, one to predict each outcome.

Training: Each record in the training set represents a potential retry which previously failed due to memory configuration errors or unclassified errors. The labels are: a) did retry fail, b) retry cost. The raw feature inputs are largely unstructured metadata about the job such as the Spark execution plan, the user who ran it, and the Spark configuration parameters and other job properties. We split these features into those that can be parsed into numeric values (e.g., Spark executor memory parameter) and those that cannot (e.g., user name). We used feature hashing to process the non-numeric values because they come from a high cardinality and dynamic set of values. We then create a lower dimensionality embedding which is concatenated with the normalized numeric values and passed through several more layers.

Inference: Upon passing validation audits, each new model version is stored in Metaflow Hosting, a service provided by our internal ML Platform. The optimizer makes several calls to the model prediction function for each incoming configuration recommendation request, described in more detail below.

Optimizer

When a job attempt fails, it sends a request to Nightingale with a job identifier. From this identifier, the service constructs the feature vector to be used in inference calls. As described previously, some of these features are Spark configuration parameters which are candidates to be mutated (e.g., spark.executor.memory, spark.executor.cores). The set of Spark configuration parameters was based on distilled knowledge of domain experts who work on Spark performance tuning extensively. We use Bayesian Optimization (implemented via Meta's Ax library) to explore the configuration space and generate a recommendation. At each iteration, the optimizer generates a candidate parameter value combination (e.g., spark.executor.memory=7192 mb, spark.executor.cores=8), then evaluates that candidate by calling the prediction model to estimate retry failure probability and cost using the candidate configuration (i.e., mutating their values in the feature vector). After a fixed number of iterations is exhausted, the optimizer returns the "best" configuration solution (i.e., that which minimized the combined retry failure and cost objective) for ConfigService to use if it is feasible. If no feasible solution is found, we disable retries.

One downside of the iterative design of the optimizer is that any bottleneck can block completion and cause a timeout, which we initially observed in a non-trivial number of cases. Upon further profiling, we found that most of the latency came from the candidate generated step (i.e., figuring out which directions to step in the configuration space after the previous iteration's evaluation results). We found that this issue had been raised to Ax library owners, who added GPU acceleration options in their API. Leveraging this option decreased our timeout rate substantially.

Rollout in Production

We have deployed Auto Remediation in production to handle memory configuration errors and unclassified errors for Spark jobs. Besides the retry success probability and cost efficiency, the impact on user experience is the major concern:

- Auto remediation improves user experience because the job retry is rarely successful without a new configuration for memory configuration errors. This means that a successful retry with the recommended configurations can reduce the operational loads and save job running costs, while a failed retry does not make the user experience worse.
- Auto remediation recommends whether to restart the job if the error cannot be classified by existing rules in the rule-based classifier. In particular, if the ML model predicts that the retry is very likely to fail, it will recommend disabling the retry, which can save the job running costs for unnecessary retries. For cases in which the job is business-critical and the user prefers always retrying the job even if the retry success probability is low, we can add a new rule to the rule-based classifier so that the same error will be classified by the rule-based classifier next time, skipping the recommendations of the ML service. This presents the advantages of the integrated intelligence of the rule-based classifier and the ML service.

The deployment in production has demonstrated that Auto Remediationcan provide effective configurations for memory configuration errors, successfully remediating about 56% of all memory configuration without human intervention. It also decreases compute cost of these jobs by about 50% because it can either recommend new configurations to make the retry successful or disable unnecessary retries. As tradeoffs between performance and cost efficiency are tunable, we can decide to achieve a higher success rate or more cost savings by tuning the ML service.

It is worth noting that the ML service is currently adopting a conservative policy to disable retries. As discussed above, this is to avoid the impact on the cases that users prefer always retrying the job upon job failures. Although these cases are expected and can be addressed by adding new rules to the rule-based classifier, we consider tuning the objective function in an incremental manner to gradually disable more retries is helpful to provide desirable user experience. Given the current policy to disable retries is conservative, Auto Remediation presents a great potential to eventually bring much more cost savings without affecting the user experience.

Beyond Error Handling: Towards Right Sizing

Auto Remediation is our first step in leveraging data insights and Machine Learning (ML) for improving user experience, reducing the operational burden, and improving cost efficiency of the data platform. It focuses on automating the remediation of failed jobs, but also paves the path to automate operations other than error handling.

One of the initiatives we are taking, called *Right Sizing*, is to reconfigure scheduled big data jobs to request the proper resources for job execution. For example, we have noted that the average requested executor memory of Spark jobs is about four times their max used memory, indicating a significant overprovision. In addition to the configurations of the job itself, the resource overprovision of the container that is requested to execute the job can also be reduced for cost savings. With heuristic- and ML-based methods, we can infer the proper configurations of job execution to minimize resource overprovisions and save millions of dollars per year without affecting the performance. Similar to Auto Remediation, these configurations can be automatically applied via ConfigService without human intervention. Right Sizing is in progress and will be covered with more details in a dedicated technical blog post later. Stay tuned.

Acknowledgements

Auto Remediation is a joint work of the engineers from different teams and organizations. This work would have not been possible without the solid, in-depth collaborations. We would like to appreciate all folks, including Spark experts, data scientists, ML engineers, the scheduler and job orchestrator engineers, data engineers, and support engineers, for sharing the context and providing constructive suggestions and valuable feedback (e.g., John Zhuge, Jun He, Holden Karau, Samarth Jain, Julian Jaffe, Batul Shajapurwala, Michael Sachs, Faisal Siddiqi).

Noisy Neighbor Detection with eBPF

netflixtechblog.com/noisy-neighbor-detection-with-ebpf-64b1f4b3bbdd

Netflix Technology Blog September 10, 2024

Learn about Netflix’s world class engineering efforts, company culture, product developments and more.

By , , ,

The Compute and Performance Engineering teams at Netflix regularly investigate performance issues in our multi-tenant environment. The first step is determining whether the problem originates from the application or the underlying infrastructure. One issue that often complicates this process is the "noisy neighbor" problem. On Titus, our multi-tenant compute platform, a "noisy neighbor" refers to a container or system service that heavily utilizes the server's resources, causing performance degradation in adjacent containers. We usually focus on CPU utilization because it is our workloads’ most frequent source of noisy neighbor issues.

Detecting the effects of noisy neighbors is complex. Traditional performance analysis tools such as perf can introduce significant overhead, risking further performance degradation. Additionally, these tools are typically deployed after the fact, which is too late for effective investigation.Another challenge is that debugging noisy neighbor issues requires significant low-level expertise and specialized tooling. In this blog post, we'll reveal how we leveraged to achieve continuous, low-overhead instrumentation of the Linux scheduler, enabling effective self-serve monitoring of noisy neighbor issues. You’ll learn how Linux kernel instrumentation can improve your infrastructure observability with deeper insights and enhanced monitoring.

To ensure the reliability of our workloads that depend on low latency responses, we instrumented the run queue latency for each container, which measures the time processes spend in the scheduling queue before being dispatched to the CPU. Extended waiting in this queue can be a telltale of performance issues, especially when containers are not utilizing their total CPU allocation. Continuous instrumentation is critical to catching such matters as they emerge, and eBPF, with its hooks into the Linux scheduler with minimal overhead, enabled us to monitor run queue latency efficiently.

To emit a run queue latency metric, we leveraged three eBPF hooks: `sched_wakeup`, `sched_wakeup_new`, and `sched_switch`.

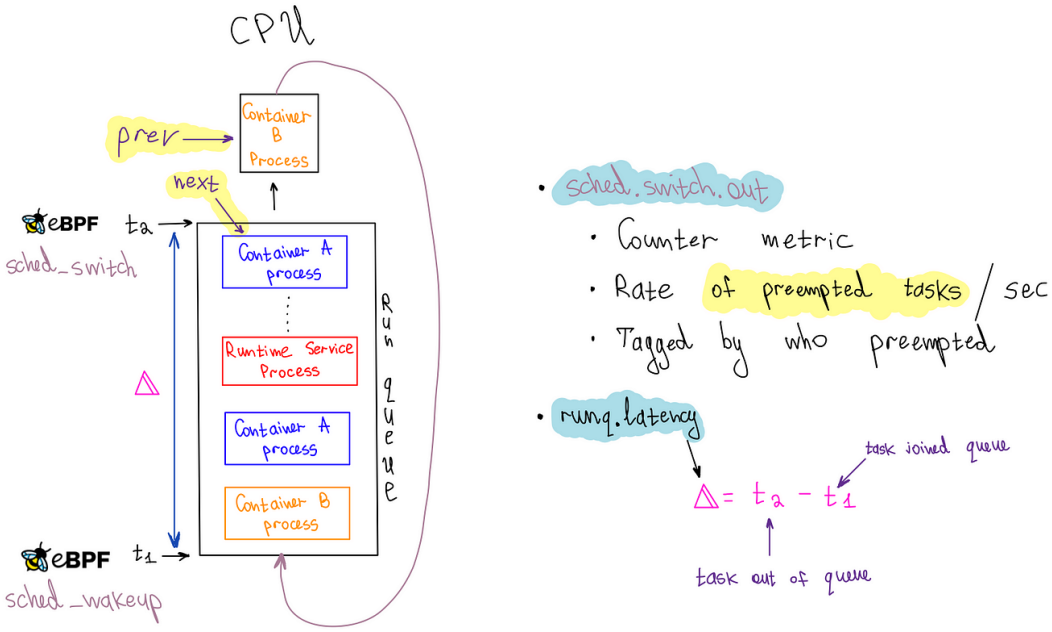


Diagram of how run queue latency is measured and instrumented

The `sched_wakeup`and `sched_wakeup_new` hooks are invoked when a process changes state from 'sleeping' to 'runnable.' They let us identify when a process is ready to run and is waiting for CPU time. During this event, we generate a timestamp and store it in an eBPF hash map using the process ID as the key.

```
        bpf_map_update_elem(&runq_enqueued, &pid, &ts, BPF_NOEXIST);     ;}

    __uint(type, BPF_MAP_TYPE_HASH);    __uint(max_entries, MAX_TASK_ENTRIES);
__uint(key_size, (u32));    __uint(value_size, (u64));} runq_enqueued ;


SEC(){ ( *)ctx[];    u32 pid = task->pid;    u64 ts = bpf_ktime_get_ns();
```

Conversely, the `sched_switch` hook is triggered when the CPU switches between processes. This hook provides pointers to the process currently utilizing the CPU and the process about to take over. We use the upcoming task's process ID (PID) to fetch the timestamp from the eBPF map. This timestamp represents when the process entered the queue, which we had previously stored. We then calculate the run queue latency by simply subtracting the timestamps.

```
                                    bpf_map_delete_elem(&runq_enqueued,
&next_pid);    ....

SEC(){ ( task_struct *)ctx[]; ( task_struct *)ctx[];    u32 prev_pid = prev->pid;
u32 next_pid = next->pid;


    u64 *tsp = bpf_map_lookup_elem(&runq_enqueued, &next_pid); (tsp == ) {;     }


    u64 now = bpf_ktime_get_ns();    u64 runq_lat = now - *tsp;
```

One of the advantages of eBPF is its ability to provide pointers to the actual kernel data structures representing processes or threads, also known as tasks in kernel terminology. This feature enables access to a wealth of information stored about a process. We required the process's cgroup ID to associate it with a container for our specific use case. However, the cgroup information in the process struct is safeguarded by an RCU (Read Copy Update) lock.

To safely access this RCU-protected information, we can leverage kfuncs in eBPF. kfuncs are kernel functions that can be called from eBPF programs. There are kfuncs available to lock and unlock RCU read-side critical sections. These functions ensure that our eBPF program remains safe and efficient while retrieving the cgroup ID from the task struct.

```
  u64 {        u64 cgroup_id;    bpf_rcu_read_lock();    cgroups = task->cgroups;
cgroup_id = cgroups->dfl_cgrp->kn->id;    bpf_rcu_read_unlock();     cgroup_id;}

 __ksym; __ksym;
```

Once the data is ready, we must package it and send it to userspace. For this purpose, we chose the eBPF ring buffer. It is efficient, high-performing, and user-friendly. It can handle variable-length data records and allows data reading without necessitating extra memory copying or syscalls. However, the sheer number of data points was causing the userspace program to use too much CPU, so we implemented a rate limiter in eBPF to sample the data.

```
                                                                    ;}

    __uint(type, BPF_MAP_TYPE_RINGBUF);    __uint(max_entries,
RINGBUF_SIZE_BYTES);} events ;


    __uint(type, BPF_MAP_TYPE_PERCPU_HASH);    __uint(max_entries,
MAX_TASK_ENTRIES);    __uint(key_size, (u64));    __uint(value_size, (u64));}
cgroup_id_to_last_event_ts ;


    u64 prev_cgroup_id;    u64 cgroup_id;    u64 runq_lat;    u64 ts;};


SEC(){


    u64 prev_cgroup_id = get_task_cgroup_id(prev);    u64 cgroup_id =
get_task_cgroup_id(next);


    u64 *last_ts =         bpf_map_lookup_elem(&cgroup_id_to_last_event_ts,
&cgroup_id);    u64 last_ts_val = last_ts ==  ?  : *last_ts;


 (now - last_ts_val < RATE_LIMIT_NS) {;    }


    event = bpf_ringbuf_reserve(&events, (*event), );


 (event) {        event->prev_cgroup_id = prev_cgroup_id;        event->cgroup_id
= cgroup_id;        event->runq_lat = runq_lat;        event->ts = now;
bpf_ringbuf_submit(event, );
bpf_map_update_elem(&cgroup_id_to_last_event_ts, &cgroup_id,            &now,
BPF_ANY);


    }
```

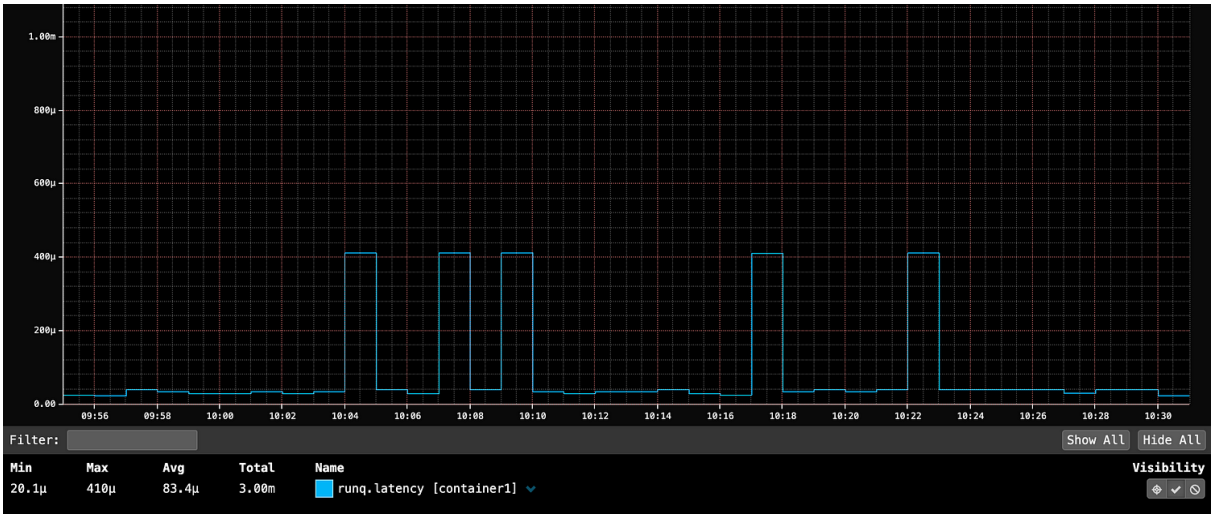
Our userspace application, developed in Go, processes events from the ring buffer to emit metrics to our metrics backend, Atlas. Each event includes a run queue latency sample with a cgroup ID, which we associate with containers running on the host. We categorize it as a system service if no such association is found. When a cgroup ID is associated with a container, we emit a percentile timer Atlas metric (`runq.latency`) for that container. We also increment a counter metric (`sched.switch.out`) to monitor preemptions occurring for the container's processes. Access to the `prev_cgroup_id` of

the preempted process allows us to tag the metric with the cause of the preemption, whether it's due to a process within the same container (or cgroup), a process in another container, or a system service.

It's important to highlight that both the `runq.latency` metric and the `sched.switch.out` metrics are needed to determine if a container is affected by noisy neighbors, which is the goal we aim to achieve — relying solely on the `runq.latency` metric can lead to misconceptions. For example, if a container is at or over its cgroup CPU limit, the scheduler will throttle it, resulting in an apparent spike in run queue latency due to delays in the queue. If we were only to consider this metric, we might incorrectly attribute the performance degradation to noisy neighbors when it's actually because the container is hitting its CPU quota. However, simultaneous spikes in both metrics, mainly when the cause is a different container or system process, clearly indicate a noisy neighbor issue.

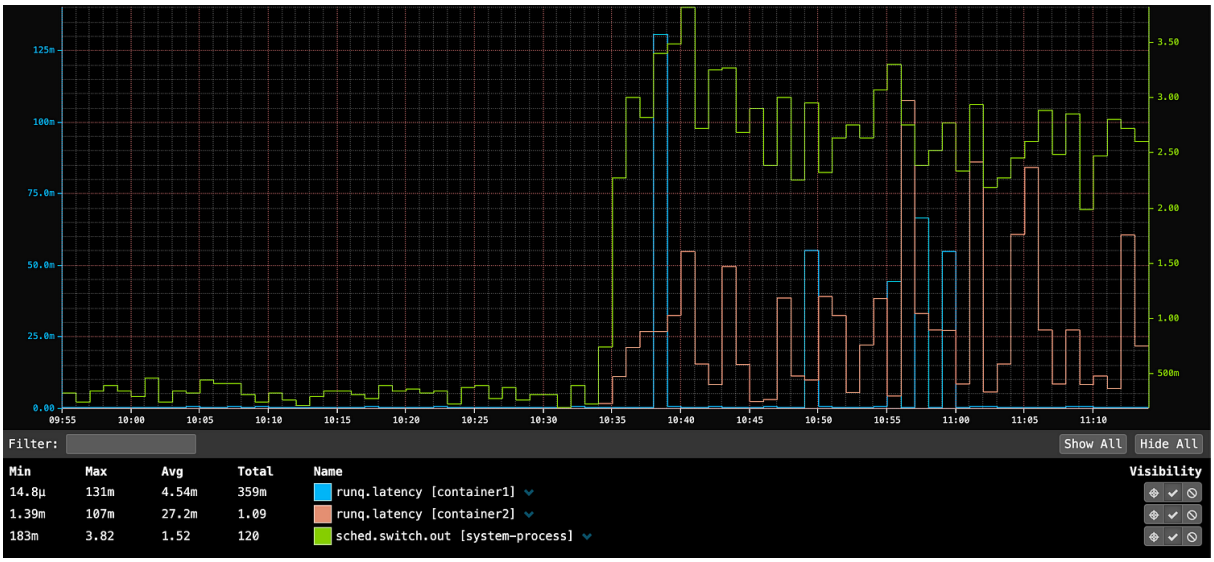
A Noisy Neighbor Story

Below is the `runq.latency` metric for a server running a single container with ample CPU capacity. The 99th percentile averages 83.4µs (microseconds), serving as our baseline. Although there are some spikes reaching 400µs, the latency remains within acceptable parameters.



container1's 99th percentile runq.latency averages 83µs (microseconds), with spikes up to 400µs, without adjacent containers. This serves as our baseline for a container not contending for CPU on a host.

At 10:35, launching `container2`, which fully utilized all CPUs on the host, caused a significant 131-millisecond spike (131,000 microseconds) in `container1`'s P99 run queue latency. This spike would be noticeable in the userspace application if it were serving HTTP traffic. If userspace app owners reported an unexplained latency spike, we could quickly identify the noisy neighbor issue through run queue latency metrics.



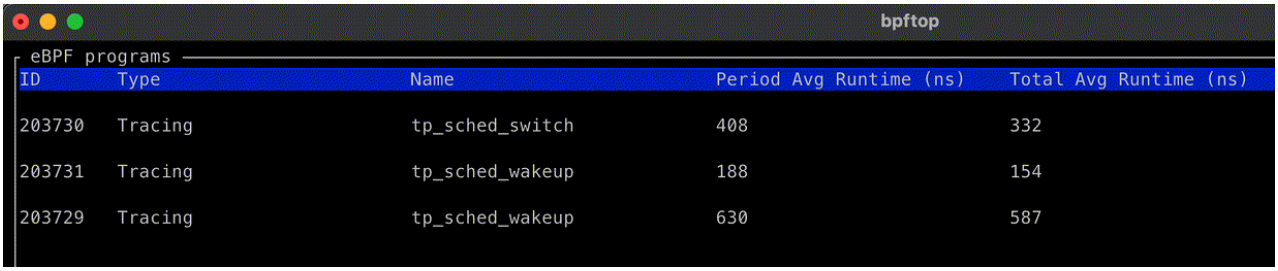
Launching container2 at 10:35, which maxes out all CPUs on the host, due to increased preemptions by system processes. This indicates a noisy neighbor issue, where system services compete for CPU time with containers.

The `sched.switch.out` metric indicates that the spike was due to increased preemptions by system processes, highlighting a noisy neighbor issue where system services compete with containers for CPU time. Our metrics show that the noisy neighbors were actually system processes, likely triggered by `container2` consuming all available CPU capacity.

Optimizing eBPF Code

We developed an open-source eBPF process monitoring tool called bpftop to measure the overhead of eBPF code in this kernel hot path. Our profiling with `bpftop` shows that the instrumentation adds less than 600 nanoseconds to each `sched_*` hook. We conducted a performance analysis on a Java service running in a container, and the instrumentation did not introduce significant overhead. The performance variance with the run queue profiling code active versus inactive was not measurable in milliseconds.

During our research on how eBPF statistics are measured in the kernel, we identified an opportunity to improve the calculation. We submitted this patch, which was included in the Linux kernel 6.10 release.



Through trial and error and using `bpftop`, we identified several optimizations that helped maintain low overhead for our eBPF code:

- We found that `BPF_MAP_TYPE_HASH` was the most performant for storing enqueued timestamps. Using `BPF_MAP_TYPE_TASK_STORAGE` resulted in nearly a twofold performance decline. `BPF_MAP_TYPE_PERCPU_HASH` was slightly less performant than `BPF_MAP_TYPE_HASH`, which was unexpected and requires further investigation.
- `BPF_MAP_TYPE_LRU_HASH` maps are 40–50 nanoseconds slower per operation than regular hash maps. Due to space concerns from PID churn, we initially used them for enqueued timestamps. Ultimately, we settled on `BPF_MAP_TYPE_HASH` with an increased size to mitigate this risk.
- The `BPF_CORE_READ` helper adds 20–30 nanoseconds per invocation. In the case of raw tracepoints, specifically those that are "BTF-enabled" (`tp_btf/*`), it is safe and more efficient to access the task struct members directly. Andrii Nakryiko recommends this approach in this .
- The `sched_switch`, `sched_wakeup`, and `sched_wakeup_new` are all triggered for kernel tasks, which are identifiable by their PID of 0. We found monitoring these tasks unnecessary, so we implemented several early exit conditions and conditional logic to prevent executing costly operations, such as accessing BPF maps, when dealing with a kernel task. Notably, kernel tasks operate through the scheduler queue like any regular process.

Conclusion

Our findings highlight the value of low-overhead continuous instrumentation of the Linux kernel with eBPF. We have integrated these metrics into customer dashboards, enabling actionable insights and guiding multitenancy performance discussions. We can also now use these metrics to refine CPU isolation strategies to minimize the impact of noisy neighbors. Additionally, thanks to these metrics, we've gained deeper insights into the Linux scheduler.

This work has also deepened our understanding of eBPF technology and underscored the importance of tools like `bpftop` for optimizing eBPF code. As eBPF adoption increases, we foresee more infrastructure observability and business logic shifting to it. One promising project in this space is sched_ext, which has the potential to revolutionize how scheduling decisions are made and tailored to specific workload needs.

Investigation of a Workbench UI Latency Issue

netflixtechblog.com/investigation-of-a-workbench-ui-latency-issue-faa017b4653d

Netflix Technology Blog October 14, 2024

Learn about Netflix's world class engineering efforts, company culture, product developments and more.

By: Hechao Li and Marcelo Mayworm

With special thanks to our stunning colleagues Amer Ather, Itay Dafna, Luca Pozzi, Matheus Leão, and Ye Ji.

At Netflix, the Analytics and Developer Experience organization, part of the Data Platform, offers a product called Workbench. Workbench is a remote development workspace based on Titus that allows data practitioners to work with big data and machine learning use cases at scale. A common use case for Workbench is running JupyterLab Notebooks.

Recently, several users reported that their JupyterLab UI becomes slow and unresponsive when running certain notebooks. This document details the intriguing process of debugging this issue, all the way from the UI down to the Linux kernel.

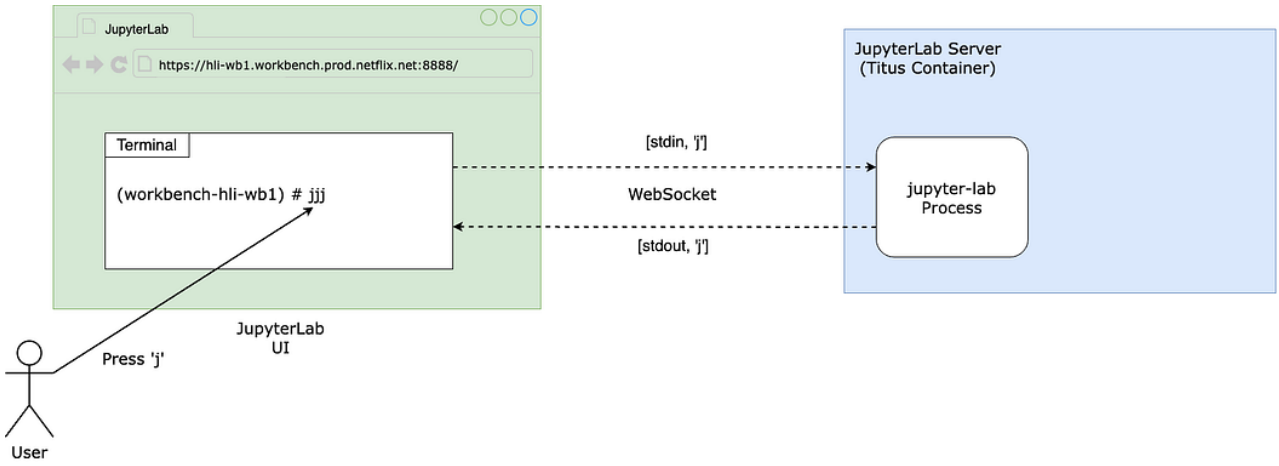
Symptom

Machine Learning engineer Luca Pozzi reported to our Data Platform team that their **JupyterLab UI on their workbench becomes slow and unresponsive when running some of their Notebooks.** Restarting the *ipykernel* process, which runs the Notebook, might temporarily alleviate the problem, but the frustration persists as more notebooks are run.

Quantify the Slowness

While we observed the issue firsthand, the term "UI being slow" is subjective and difficult to measure. To investigate this issue, **we needed a quantitative analysis of the slowness**.

Itay Dafna devised an effective and simple method to quantify the UI slowness. Specifically, we opened a terminal via JupyterLab and held down a key (e.g., "j") for 15 seconds while running the user's notebook. The input to stdin is sent to the backend (i.e., JupyterLab) via a WebSocket, and the output to stdout is sent back from the backend and displayed on the UI. We then exported the *.har* file recording all communications from the browser and loaded it into a Notebook for analysis.



Using this approach, we observed latencies ranging from 1 to 10 seconds, averaging 7.4 seconds.

```
[8]: analyze_har("workbench-websocket-data-repro.har")
Last executed at 2024-07-19 22:46:05 in 7.00s
Arithmetic mean (in seconds):  7.42267880761013
Stdev:  15.5084540925444
Median:  3.127
```

Blame The Notebook

Now that we have an objective metric for the slowness, let's officially start our investigation. If you have read the symptom carefully, you must have noticed that the slowness only occurs when the user runs **certain** notebooks but not others.

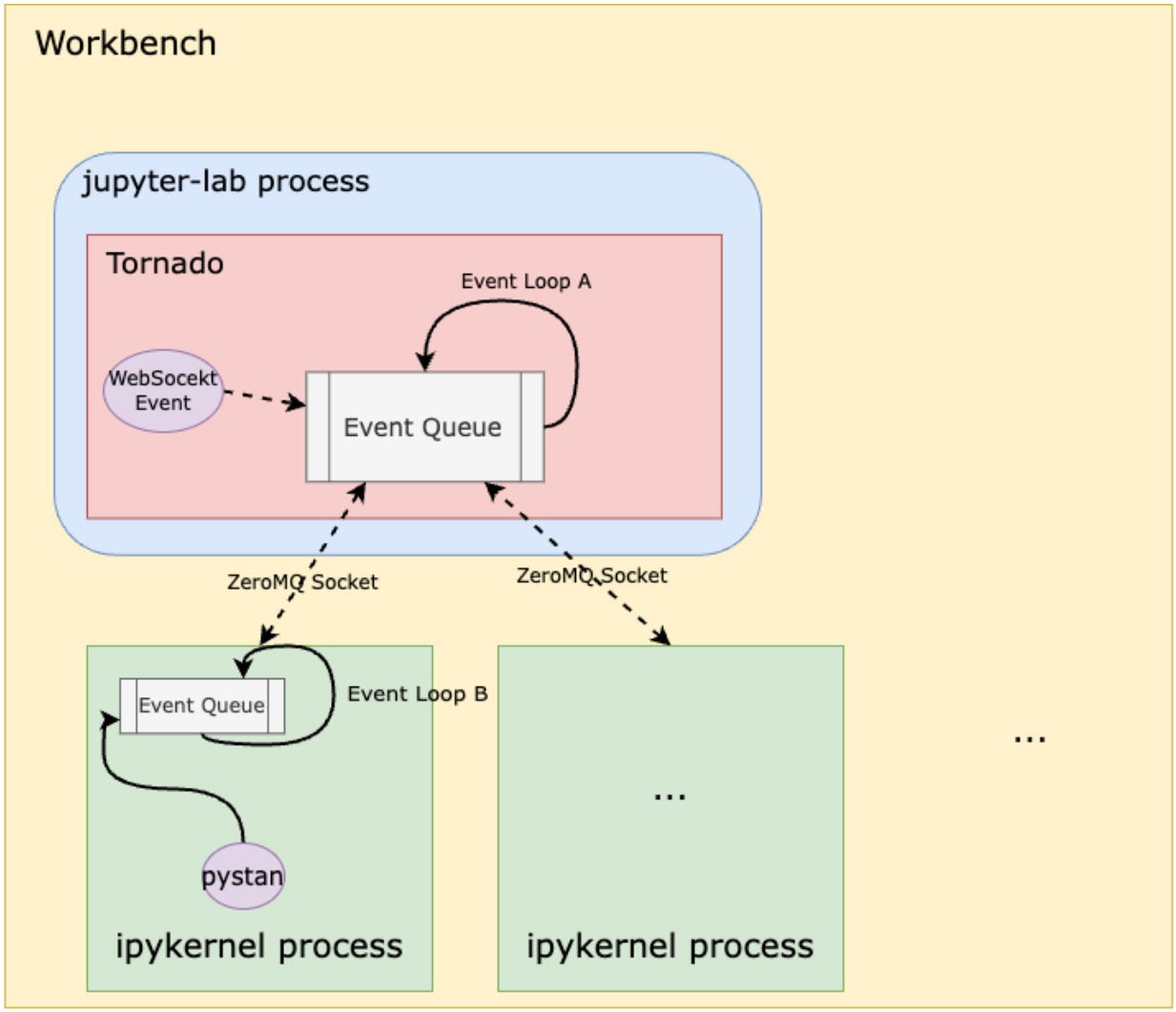
Therefore, the first step is scrutinizing the specific Notebook experiencing the issue. Why does the UI always slow down after running this particular Notebook? Naturally, you would think that there must be something wrong with the code running in it.

Upon closely examining the user's Notebook, we noticed a library called *pystan* , which provides Python bindings to a native C++ library called stan, looked suspicious. Specifically, *pystan* uses *asyncio*. However, **because there is already an existing event loop running in the Notebook process and cannot be nested by design, in order for to work, the authors of injecting into the existing event loop by using a package called** , a library that became unmaintained because the author unfortunately passed away.

Given this seemingly hacky usage, we naturally suspected that the events injected by *pystan* into the event loop were blocking the handling of the WebSocket messages used to communicate with the JupyterLab UI. This reasoning sounds very plausible. However, **the user claimed that there were cases when a Notebook not using runs, the UI also became slow**.

Moreover, after several rounds of discussion with ChatGPT, we learned more about the architecture and realized that, in theory, **the usage of and should not cause the slowness in handling the UI WebSocket** for the following reasons:

Even though *pystan* uses *nest_asyncio* to inject itself into the main event loop, **the Notebook runs on a child process (i.e., the process) of the server process**, which means the main event loop being injected by *pystan* is that of the *ipykernel* process, not the *jupyter-server* process. Therefore, even if *pystan* blocks the event loop, it shouldn't impact the *jupyter-lab* main event loop that is used for UI websocket communication. See the diagram below:



In other words, **events are injected to the event loop B in this diagram instead of event loop A**. So, it shouldn't block the UI WebSocket events.

You might also think that because event loop A handles both the WebSocket events from the UI and the ZeroMQ socket events from the *ipykernel* process, a high volume of ZeroMQ events generated by the notebook could block the WebSocket. However, **when we captured packets on the ZeroMQ socket while reproducing the issue, we didn't observe heavy traffic on this socket that could cause such blocking**.

A stronger piece of evidence to rule out *pystan* was that we were ultimately able to reproduce the issue even without it, which I'll dive into later.

Blame Noisy Neighbors

The Workbench instance runs as a Titus container. To efficiently utilize our compute resources, **Titus employs a CPU oversubscription feature**, meaning the combined virtual CPUs allocated to containers exceed the number of available physical CPUs on a Titus agent. **If a container is unfortunate enough to be scheduled alongside other "noisy" containers — those that consume a lot of CPU resources — it could suffer from CPU deficiency.**

However, after examining the CPU utilization of neighboring containers on the same Titus agent as the Workbench instance, as well as the overall CPU utilization of the Titus agent, we quickly ruled out this hypothesis. Using the top command on the Workbench, we observed that when running the Notebook, **the Workbench instance uses only 4 out of the 64 CPUs allocated to it**. Simply put, **this workload is not CPU-bound.**

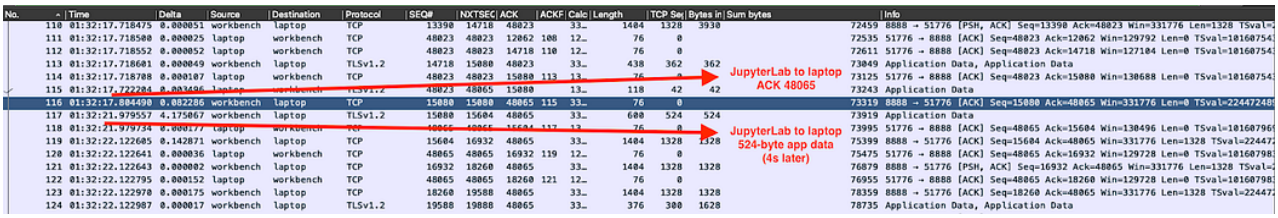
```
top - 17:39:45 up 4 days, 14:23,  0 users,  load average: 19.82, 17.12, 18.11
Tasks: 198 total,   5 running, 192 sleeping,   0 stopped,   1 zombie
%Cpu(s): 21.1 us,  2.2 sy,  0.0 ni, 75.6 id,  0.9 wa,  0.0 hi,  0.2 si,  0.0 st
MiB Mem : 491520.0 total, 458180.3 free,  17764.6 used,  15575.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used. 473759.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
3756800 root      20   0   13.9g   2.3g   7680 R 100.0   0.5   0:09.00 python3.10
3756806 root      20   0   13.9g   2.3g   7680 R 100.0   0.5   0:08.90 python3.10
3756808 root      20   0   13.9g   2.3g   7680 R 100.0   0.5   0:08.58 python3.10
3756807 root      20   0   13.9g   2.3g   7680 R  99.7   0.5   0:08.77 python3.10
2119005 root      20   0   14.0g   2.4g 129792 S   2.3   0.5   1:04.44 python3.10
 183086 root      20   0 5644032 274176  41472 S   2.0   0.1 106:19.11 jupyter-lab
     14 root      20   0 1007396  28368   9216 S   0.7   0.0  12:17.42 spectatord
3756461 root      20   0   10632   3840   3072 R   0.7   0.0   0:00.06 top
```

Blame The Network

The next theory was that the network between the web browser UI (on the laptop) and the JupyterLab server was slow. To investigate, we **captured all the packets between the laptop and the server** while running the Notebook and continuously pressing 'j' in the terminal.

When the UI experienced delays, we observed a 5-second pause in packet transmission from server port 8888 to the laptop. Meanwhile, **traffic from other ports, such as port 22 for SSH, remained unaffected**. This led us to conclude that the pause was caused by the application running on port 8888 (i.e., the JupyterLab process) rather than the network.



The Minimal Reproduction

As previously mentioned, another strong piece of evidence proving the innocence of pystan was that we could reproduce the issue without it. By gradually stripping down the “bad” Notebook, we eventually arrived at a minimal snippet of code that reproduces the issue without any third-party dependencies or complex logic:

```
                p  processes:      p.join()

 time os multiprocessing  Process


N = os.cpu_count()


():  time.sleep()


 __name__ == :(, )  file:    data = file.read()    processes = [] i (N):      p =
Process(target=launch_worker, args=(i,))      processes.append(p)      p.start()
```

The code does only two things:

1. Read a 2GB file into memory (the Workbench instance has 480G memory in total so this memory usage is almost negligible).
2. Start N processes where N is the number of CPUs. The N processes do nothing but sleep.

There is no doubt that this is the most silly piece of code I’ve ever written. It is neither CPU bound nor memory bound. Yet **it can cause the JupyterLab UI to stall for as many as 10 seconds!**

Questions

There are a couple of interesting observations that raise several questions:

- We noticed that . If you don’t read the 2GB file (that is not even used!), the issue is not reproducible.
- , hinting at contention on the single-threaded event loop in this process (event loop A in the diagram before).
- The code runs in a Notebook, which means it runs in the process, that is a child process of the process.

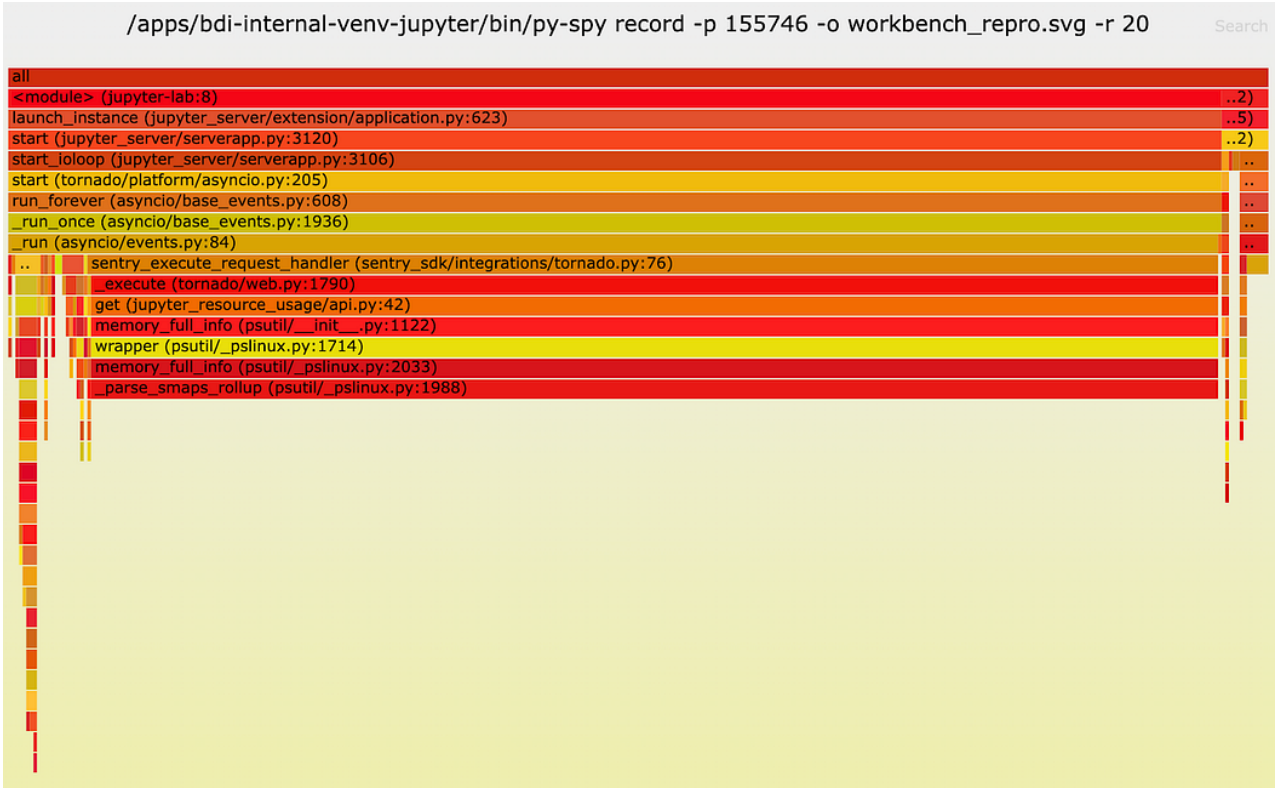
- The workbench has 64CPUs. But when we printed , the output was 96. That means .

Let's answer the last question first. In fact, if you run *lscpu* and *nproc* commands inside a Titus container, you will also see different results — the former gives you 96, which is the number of physical CPUs on the Titus agent, whereas the latter gives you 64, which is the number of virtual CPUs allocated to the container. This discrepancy is due to the lack of a "CPU namespace" in the Linux kernel, causing the number of physical CPUs to be leaked to the container when calling certain functions to get the CPU count. The assumption here is that Python **uses the same function as the command, causing it to get the CPU count of the host instead of the container**. Python 3.13 has a new call that can be used to get the accurate CPU count, but it's not GA'ed yet.

It will be proven later that this inaccurate number of CPUs can be a contributing factor to the slowness.

More Clues

Next, we used *py-spy* to do a profiling of the *jupyter-lab* process. Note that we profiled the parent *jupyter-lab* process, **not** the *ipykernel* child process that runs the reproduction code. The profiling result is as follows:



As one can see, **a lot of CPU time (89%!!) is spent on a function called** . In comparison, the terminal handler used only 0.47% CPU time. From the stack trace, we see that **this function is inside the event loop A**, **so it can definitely cause the UI WebSocket events to be delayed**.

The stack trace also shows that this function is ultimately called by a function used by a Jupyter lab extension called *jupyter_resource_usage*. **We then disabled this extension and restarted the process. As you may have guessed, we could no longer reproduce the slowness!**

But our puzzle is not solved yet. Why does this extension cause the UI to slow down? Let's keep digging.

Root Cause Analysis

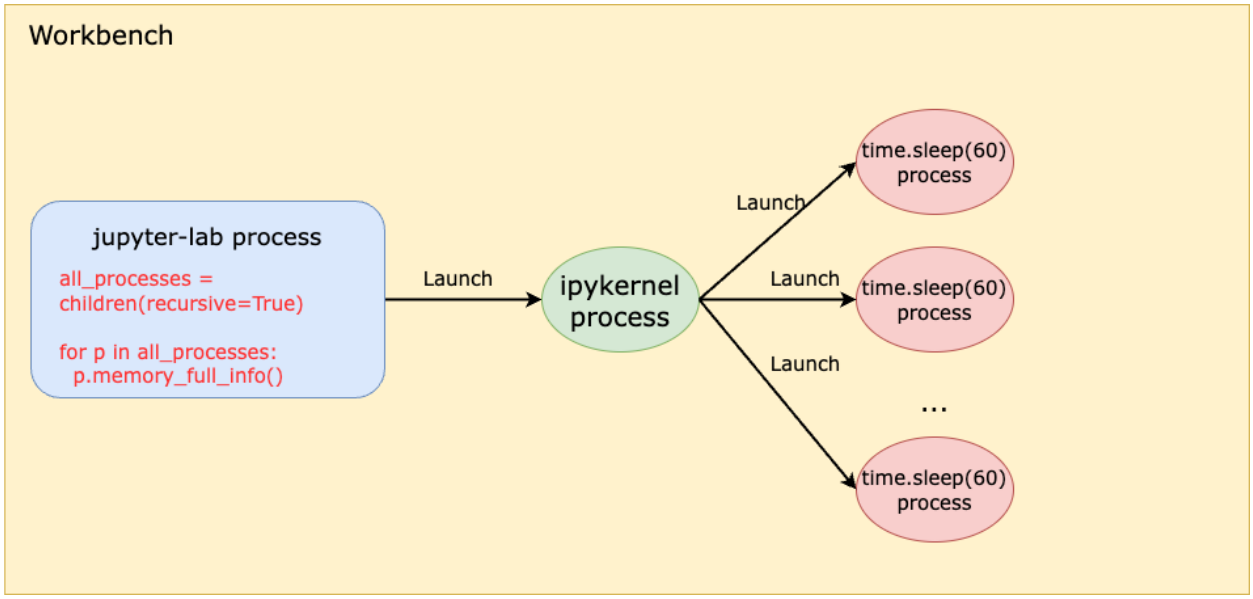
From the name of the extension and the names of the other functions it calls, we can infer that this extension is used to get resources such as CPU and memory usage information. Examining the code, we see that this function call stack is triggered when an API endpoint */metrics/v1* is called from the UI. **The UI apparently calls this function periodically**, according to the network traffic tab in Chrome's Developer Tools.

Now let's look at the implementation starting from the call *get(jupter_resource_usage/api.py:42)* . The full code is here and the key lines are shown below:

```
 p  all_processes:  info = p.memory_full_info()

cur_process = psutil.Process()all_processes = [cur_process] +
cur_process.children(recursive=)
```

Basically, it gets all children processes of the *jupyter-lab* process recursively, including both the *ipykernel* Notebook process and all processes created by the Notebook. Obviously, **the cost of this function is linear to the number of all children processes**. In the reproduction code, we create 96 processes. So here we will have at least 96 (sleep processes) + 1 (*ipykernel* process) + 1 (*jupyter-lab* process) = 98 processes when it should actually be 64 (allocated CPUs) + 1 (*ipykernel* process) + 1 *(jupyter-lab* process) = 66 processes, because the number of CPUs allocated to the container is, in fact, 64.



This is truly ironic. **The more CPUs we have, the slower we are!**

At this point, we have answered one question: **Why does starting many grandchildren processes in the child process cause the parent process to be slow?** Because the parent process runs a function that's linear to the number all children process recursively.

However, this solves only half of the puzzle. If you remember the previous analysis, **starting many child processes ALONE doesn't reproduce the issue**. If we don't read the 2GB file, even if we create 2x more processes, we can't reproduce the slowness.

So now we must answer the next question: **Why does reading a 2GB file in the child process affect the parent process performance,** especially when the workbench has as much as 480GB memory in total?

To answer this question, let's look closely at the function ***__parse_smaps_rollup***. As the name implies, this function parses the file */proc/<pid>/smaps_rollup*.

```
 ():  uss = pss = swap =    open_binary(.(self._procfs_path, self.pid))  f:   line
f:     line.startswith(b"Private_"):          s uss += (line.split()[]) *
line.startswith(b"Pss:"):      pss = (line.split()[]) *
line.startswith(b"Swap:"):      swap = (line.split()[]) *  (uss, pss, swap)
```

Naturally, you might think that when memory usage increases, this file becomes larger in size, causing the function to take longer to parse. Unfortunately, this is not the answer because:

- First, .
- Second, instead of a regular file on disk. In other words, .

This file was introduced in this commit in 2017, with the purpose of improving the performance of user programs that determine aggregate memory statistics. Let's first focus on the handler of syscall on this */proc/<pid>/smaps_rollup*.

```
static int smaps_rollup_open(struct inode *inode, struct file *file)
{
        int ret;
        struct proc_maps_private *priv;

        priv = kzalloc(sizeof(*priv), GFP_KERNEL_ACCOUNT);
        if (!priv)
                return -ENOMEM;

        ret = single_open(file, show_smaps_rollup, priv);
        if (ret)
                goto out_free;

        priv->inode = inode;
        priv->mm = proc_mem_open(inode, PTRACE_MODE_READ);
        if (IS_ERR(priv->mm)) {
                ret = PTR_ERR(priv->mm);

                single_release(inode, file);
                goto out_free;
        }

        return 0;

out_free:
        kfree(priv);
        return ret;
}
```

Following through the *single_open* function, we will find that it uses the function *show_smaps_rollup* for the show operation, which can translate to the *read* system call on the file. Next, we look at the *show_smaps_rollup* implementation. You will notice **a do-while loop that is linear to the virtual memory area**.

```
   {  …  vma_start = vma->vm_start;   {    smap_gather_stats(vma, &mss, );
last_vma_end = vma->vm_end;    …  } for_each_vma(vmi, vma);  …}
```

This perfectly **explains why the function gets slower when a 2GB file is read into memory**. **Because the handler of reading the file now takes longer to run the while loop**. Basically, even though already improved the performance of getting memory information compared to the old method of parsing the */proc/<pid>/smaps* file, **it is still linear to the virtual memory used**.

More Quantitative Analysis

Even though at this point the puzzle is solved, let's conduct a more quantitative analysis. How much is the time difference when reading the *smaps_rollup* file with small versus large virtual memory utilization? Let's write some simple benchmark code like below:

```
                    read_smaps_rollup(pid)

 os


():(.(pid), )  f: line  f:


 __name__ == “__main__”:  pid = os.getpid()


  read_smaps_rollup(pid)


(“/root/2G_file”, “rb”)  f:    data = f.read()
```

This program performs the following steps:

1. Reads the file of the current process.
2. Reads a 2GB file into memory.
3. Repeats step 1.

We then use *strace* to find the accurate time of reading the *smaps_rollup* file.

```
openat(AT_FDCWD, “/proc/3107492/smaps_rollup”, O_RDONLY|O_CLOEXEC) = 3 <0.000023>
(3, “560b42ed4000-7ffdadcef000 — -p 0”…, 1024) = 670 <0.000259>...openat(AT_FDCWD,
“/proc/3107492/smaps_rollup”, O_RDONLY|O_CLOEXEC) = 3 <0.000029>(3, “560b42ed4000-
7ffdadcef000 — -p 0”…, 1024) = 670 <0.027698>
```

```
$ sudo strace -T -e trace=openat, python3 benchmark.py 2>&1 | grep “smaps_rollup”
-A 1
```

As you can see, both times, the read *syscall* returned 670, meaning the file size remained the same at 670 bytes. However, **the time it took the second time (i.e.**, **0.027698 seconds) is 100x the time it took the first time (i.e.**, **0.000259 seconds)**! This means that if there are 98 processes, the time spent on reading this file alone will be 98 * 0.027698 = 2.7 seconds! Such a delay can significantly affect the UI experience.

Solution

This extension is used to display the CPU and memory usage of the notebook process on the bar at the bottom of the Notebook:

We confirmed with the user that disabling the *jupyter-resource-usage* extension meets their requirements for UI responsiveness, and that this extension is not critical to their use case. Therefore, we provided a way for them to disable the extension.

Summary

This was such a challenging issue that required debugging from the UI all the way down to the Linux kernel. It is fascinating that the problem is linear to both the number of CPUs and the virtual memory size — two dimensions that are generally viewed separately.

Overall, we hope you enjoyed the irony of:

1. The extension used to monitor CPU usage causing CPU contention.
2. An interesting case where the more CPUs you have, the slower you get!

If you're excited by tackling such technical challenges and have the opportunity to solve complex technical challenges and drive innovation, consider joining our Data Platform teams. Be part of shaping the future of Data Security and Infrastructure, Data Developer Experience, Analytics Infrastructure and Enablement, and more. Explore the impact you can make with us!

Java 21 Virtual Threads - Dude, Where's My Lock?

netflixtechblog.com/java-21-virtual-threads-dude-wheres-my-lock-3052540e231d

Netflix Technology Blog July 29, 2024

Learn about Netflix's world class engineering efforts, company culture, product developments and more.

By Vadim Filanovsky, Mike Huang, Danny Thomas and Martin Chalupa

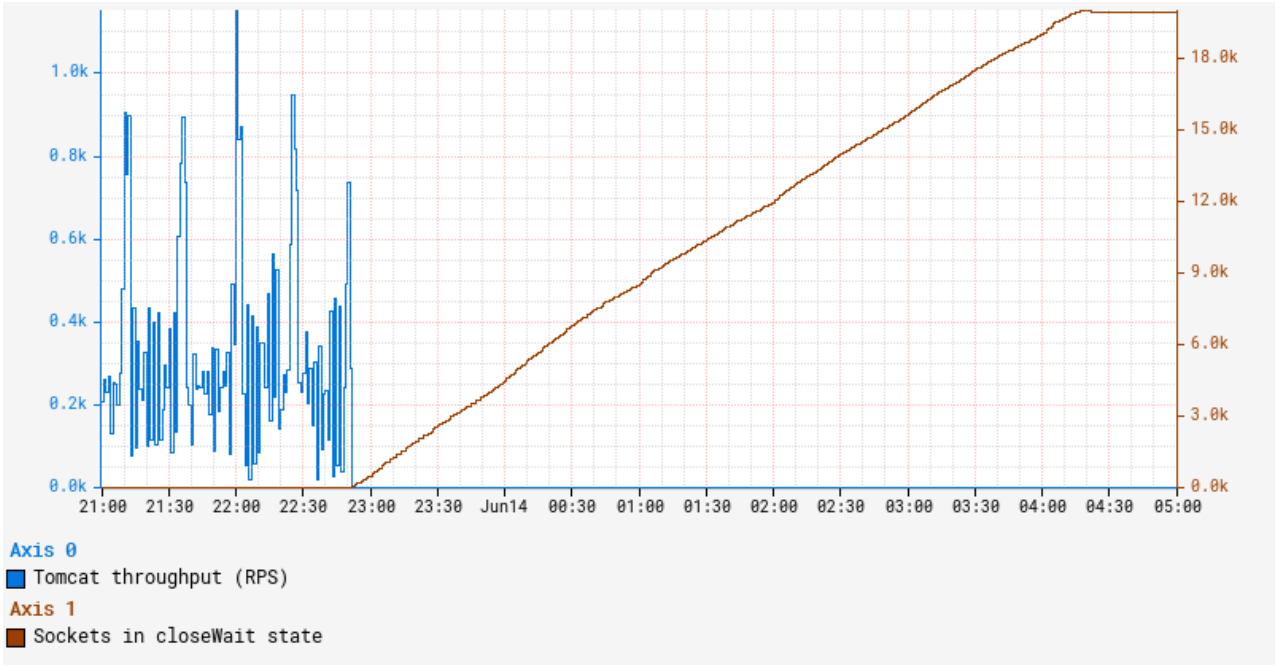
Netflix has an extensive history of using Java as our primary programming language across our vast fleet of microservices. As we pick up newer versions of Java, our JVM Ecosystem team seeks out new language features that can improve the ergonomics and performance of our systems. In a recent article, we detailed how our workloads benefited from switching to generational ZGC as our default garbage collector when we migrated to Java 21. Virtual threads is another feature we are excited to adopt as part of this migration.

For those new to virtual threads, they are described as "lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications." Their power comes from their ability to be suspended and resumed automatically via continuations when blocking operations occur, thus freeing the underlying operating system threads to be reused for other operations. Leveraging virtual threads can unlock higher performance when utilized in the appropriate context.

In this article we discuss one of the peculiar cases that we encountered along our path to deploying virtual threads on Java 21.

The problem

Netflix engineers raised several independent reports of intermittent timeouts and hung instances to the Performance Engineering and JVM Ecosystem teams. Upon closer examination, we noticed a set of common traits and symptoms. In all cases, the apps affected ran on Java 21 with SpringBoot 3 and embedded Tomcat serving traffic on REST endpoints. The instances that experienced the issue simply stopped serving traffic even though the JVM on those instances remained up and running. One clear symptom characterizing the onset of this issue is a persistent increase in the number of sockets in `closeWait` state as illustrated by the graph below:



Collected diagnostics

Sockets remaining in `closeWait` state indicate that the remote peer closed the socket, but it was never closed on the local instance, presumably because the application failed to do so. This can often indicate that the application is hanging in an abnormal state, in which case application thread dumps may reveal additional insight.

In order to troubleshoot this issue, we first leveraged our alerts system to catch an instance in this state. Since we periodically collect and persist thread dumps for all JVM workloads, we can often retroactively piece together the behavior by examining these thread dumps from an instance. However, we were surprised to find that all our thread dumps show a perfectly idle JVM with no clear activity. Reviewing recent changes revealed that these impacted services enabled virtual threads, and we knew that virtual thread call stacks do not show up in `jstack`-generated thread dumps. To obtain a more complete thread dump containing the state of the virtual threads, we used the "`jcmd Thread.dump_to_file`" command instead. As a last-ditch effort to introspect the state of JVM, we also collected a heap dump from the instance.

Analysis

Thread dumps revealed thousands of "blank" virtual threads:

```
#119822 "" virtual...

#119821 "" virtual


#119820 "" virtual


#119823 "" virtual


#120847 "" virtual
```

These are the VTs (virtual threads) for which a thread object is created, but has not started running, and as such, has no stack trace. In fact, there were approximately the same number of blank VTs as the number of sockets in closeWait state. To make sense of what we were seeing, we need to first understand how VTs operate.

A virtual thread is not mapped 1:1 to a dedicated OS-level thread. Rather, we can think of it as a task that is scheduled to a fork-join thread pool. When a virtual thread enters a blocking call, like waiting for a `Future`, it relinquishes the OS thread it occupies and simply remains in memory until it is ready to resume. In the meantime, the OS thread can be reassigned to execute other VTs in the same fork-join pool. This allows us to multiplex a lot of VTs to just a handful of underlying OS threads. In JVM terminology, the underlying OS thread is referred to as the "carrier thread" to which a virtual thread can be "mounted" while it executes and "unmounted" while it waits. A great in-depth description of virtual thread is available in JEP 444.

In our environment, we utilize a blocking model for Tomcat, which in effect holds a worker thread for the lifespan of a request. By enabling virtual threads, Tomcat switches to virtual execution. Each incoming request creates a new virtual thread that is simply scheduled as a task on a Virtual Thread Executor. We can see Tomcat creates a `VirtualThreadExecutor` here.

Tying this information back to our problem, the symptoms correspond to a state when Tomcat keeps creating a new web worker VT for each incoming request, but there are no available OS threads to mount them onto.

Why is Tomcat stuck?

What happened to our OS threads and what are they busy with? As described here, a VT will be pinned to the underlying OS thread if it performs a blocking operation while inside a `synchronized` block or method. This is exactly what is happening here. Here is a relevant snippet from a thread dump obtained from the stuck instance:

```
#119515 "" virtual      java.base/jdk.internal.misc.Unsafe.park(Native Method)
java.base/java.lang.VirtualThread.parkOnCarrierThread(VirtualThread.java:661)
java.base/java.lang.VirtualThread.park(VirtualThread.java:593)
java.base/java.lang.System$2.parkVirtualThread(System.java:2643)
java.base/jdk.internal.misc.VirtualThreads.park(VirtualThreads.java:54)
java.base/java.util.concurrent.locks.LockSupport.park(LockSupport.java:219)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQue

java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQue

java.base/java.util.concurrent.locks.ReentrantLock$Sync.lock(ReentrantLock.java:153

java.base/java.util.concurrent.locks.ReentrantLock.lock(ReentrantLock.java:322)
zipkin2.reporter.internal.CountBoundedQueue.offer(CountBoundedQueue.java:54)
zipkin2.reporter.internal.AsyncReporter$BoundedAsyncReporter.report(AsyncReporter.j

zipkin2.reporter.brave.AsyncZipkinSpanHandler.end(AsyncZipkinSpanHandler.java:214)
brave.internal.handler.NoopAwareSpanHandler$CompositeSpanHandler.end(NoopAwareSpanH

brave.internal.handler.NoopAwareSpanHandler.end(NoopAwareSpanHandler.java:48)
brave.internal.recorder.PendingSpans.finish(PendingSpans.java:116)
brave.RealSpan.finish(RealSpan.java:134)
brave.RealSpan.finish(RealSpan.java:129)
io.micrometer.tracing.brave.bridge.BraveSpan.end(BraveSpan.java:117)
io.micrometer.tracing.annotation.AbstractMethodInvocationProcessor.after(AbstractMe

io.micrometer.tracing.annotation.ImperativeMethodInvocationProcessor.proceedUnderSy

io.micrometer.tracing.annotation.ImperativeMethodInvocationProcessor.process(Impera

io.micrometer.tracing.annotation.SpanAspect.newSpanMethod(SpanAspect.java:59)
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandle
      java.base/java.lang.reflect.Method.invoke(Method.java:580)
org.springframework.aop.aspectj.AbstractAspectJAdvice.invokeAdviceMethodWithGivenAr
```

In this stack trace, we enter the synchronization in `brave.RealSpan.finish(RealSpan.java:134)`. This virtual thread is effectively pinned — it is mounted to an actual OS thread even while it waits to acquire a reentrant lock. There are 3 VTs in this exact state and another VT identified as "`<redacted> @DefaultExecutor - 46542`" that also follows the same code path. These 4 virtual threads are pinned while waiting to acquire a lock. Because the app is deployed on an instance with 4 vCPUs, the fork-join pool that underpins VT execution also contains 4 OS threads. Now that we have exhausted all of them, no other virtual thread can make any progress. This explains why Tomcat stopped processing the requests and why the number of sockets in `closeWait` state keeps climbing. Indeed, Tomcat accepts a connection on a socket, creates a request along with a virtual thread, and passes this request/thread to the executor for processing. However, the newly created VT cannot be scheduled because all of the OS threads in the fork-join pool are pinned and never released. So these newly created VTs are stuck in the queue, while still holding the socket.

Who has the lock?

Now that we know VTs are waiting to acquire a lock, the next question is: Who holds the lock? Answering this question is key to understanding what triggered this condition in the first place. Usually a thread dump indicates who holds the lock with either "`- locked <0x…> (at …)`" or "`Locked ownable synchronizers`," but neither of these show up in our thread dumps. As a matter of fact, no locking/parking/waiting information is included in the `jcmd`-generated thread dumps. This is a limitation in Java 21 and will be addressed in the future releases. Carefully combing through the thread dump reveals that there are a total of 6 threads contending for the same `ReentrantLock` and associated `Condition`. Four of these six threads are detailed in the previous section. Here is another thread:

```
#119516 "" virtual
java.base/java.lang.VirtualThread.park(VirtualThread.java:582)
java.base/java.lang.System$2.parkVirtualThread(System.java:2643)
java.base/jdk.internal.misc.VirtualThreads.park(VirtualThreads.java:54)
java.base/java.util.concurrent.locks.LockSupport.park(LockSupport.java:219)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQue

java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQue

java.base/java.util.concurrent.locks.ReentrantLock$Sync.lock(ReentrantLock.java:153

java.base/java.util.concurrent.locks.ReentrantLock.lock(ReentrantLock.java:322)
zipkin2.reporter.internal.CountBoundedQueue.offer(CountBoundedQueue.java:54)
zipkin2.reporter.internal.AsyncReporter$BoundedAsyncReporter.report(AsyncReporter.j

zipkin2.reporter.brave.AsyncZipkinSpanHandler.end(AsyncZipkinSpanHandler.java:214)
brave.internal.handler.NoopAwareSpanHandler$CompositeSpanHandler.end(NoopAwareSpanH

brave.internal.handler.NoopAwareSpanHandler.end(NoopAwareSpanHandler.java:48)
brave.internal.recorder.PendingSpans.finish(PendingSpans.java:116)
brave.RealScopedSpan.finish(RealScopedSpan.java:64)      ...
```

Note that while this thread seemingly goes through the same code path for finishing a span, it does not go through a `synchronized` block. Finally here is the 6th thread:

```
#107 "AsyncReporter <redacted>"
java.base/jdk.internal.misc.Unsafe.park(Native Method)
java.base/java.util.concurrent.locks.LockSupport.park(LockSupport.java:221)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQue

java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awa

zipkin2.reporter.internal.CountBoundedQueue.drainTo(CountBoundedQueue.java:81)
zipkin2.reporter.internal.AsyncReporter$BoundedAsyncReporter.flush(AsyncReporter.ja
      zipkin2.reporter.internal.AsyncReporter$Flusher.run(AsyncReporter.java:352)
java.base/java.lang.Thread.run(Thread.java:1583)
```

This is actually a normal platform thread, not a virtual thread. Paying particular attention to the line numbers in this stack trace, it is peculiar that the thread seems to be blocked within the internal `acquire()` method *after* completing the wait. In other words, this calling

thread owned the lock upon entering `awaitNanos()`. We know the lock was explicitly acquired here. However, by the time the wait completed, it could not reacquire the lock. Summarizing our thread dump analysis:

There are 5 virtual threads and 1 regular thread waiting for the lock. Out of those 5 VTs, 4 of them are pinned to the OS threads in the fork-join pool. There's still no information on who owns the lock. As there's nothing more we can glean from the thread dump, our next logical step is to peek into the heap dump and introspect the state of the lock.

Inspecting the lock

Finding the lock in the heap dump was relatively straightforward. Using the excellent Eclipse MAT tool, we examined the objects on the stack of the `AsyncReporter` non-virtual thread to identify the lock object. Reasoning about the current state of the lock was perhaps the trickiest part of our investigation. Most of the relevant code can be found in the AbstractQueuedSynchronizer.java. While we don't claim to fully understand the inner workings of it, we reverse-engineered enough of it to match against what we see in the heap dump. This diagram illustrates our findings:

First off, the `exclusiveOwnerThread` field is `null` (2), signifying that no one owns the lock. We have an "empty" `ExclusiveNode` (3) at the head of the list (`waiter` is `null` and `status` is cleared) followed by another `ExclusiveNode` with `waiter` pointing to one of the virtual threads contending for the lock — `#119516` (4). The only place we found that clears the `exclusiveOwnerThread` field is within the `ReentrantLock.Sync.tryRelease()` method (source link). There we also set `state = 0` matching the state that we see in the heap dump (1).

With this in mind, we traced the code path to `release()` the lock. After successfully calling `tryRelease()`, the lock-holding thread attempts to signal the next waiter in the list. At this point, the lock-holding thread is still at the head of the list, even though ownership of the lock is *effectively released*. The *next* node in the list points to the thread that is *about to acquire the lock*.

To understand how this signaling works, let's look at the lock acquire path in the `AbstractQueuedSynchronizer.acquire()` method. Grossly oversimplifying, it's an infinite loop, where threads attempt to acquire the lock and then park if the attempt was unsuccessful:

```
while(true) {   if (tryAcquire()) {      return; // lock acquired   }   park();}
```

When the lock-holding thread releases the lock and signals to unpark the next waiter thread, the unparked thread iterates through this loop again, giving it another opportunity to acquire the lock. Indeed, our thread dump indicates that all of our waiter threads are parked on line 754. Once unparked, the thread that managed to acquire the lock should end up in this code block, effectively resetting the head of the list and clearing the reference to the waiter.

To restate this more concisely, the lock-owning thread is referenced by the head node of the list. Releasing the lock notifies the next node in the list while acquiring the lock resets the head of the list to the current node. This means that what we see in the heap dump reflects the state when one thread has already released the lock but the next thread has yet to acquire it. It's a weird in-between state that should be transient, but our JVM is stuck here. We know thread `#119516` was notified and is about to acquire the lock because of the `ExclusiveNode` state we identified at the head of the list. However, thread dumps show that thread `#119516` continues to wait, just like other threads contending for the same lock. How can we reconcile what we see between the thread and heap dumps?

The lock with no place to run

Knowing that thread `#119516` was actually notified, we went back to the thread dump to re-examine the state of the threads. Recall that we have 6 total threads waiting for the lock with 4 of the virtual threads each pinned to an OS thread. These 4 will not yield their OS thread until they acquire the lock and proceed out of the `synchronized` block. `#107 "AsyncReporter <redacted>"` is a regular platform thread, so nothing should prevent it from proceeding if it acquires the lock. This leaves us with the last thread: `#119516`. It is a VT, but it is not pinned to an OS thread. Even if it's notified to be unparked, it cannot proceed because there are no more OS threads left in the fork-join pool to schedule it onto. That's exactly what happens here — although `#119516` is signaled to unpark itself, it cannot leave the parked state because the fork-join pool is occupied by the 4 other VTs waiting to acquire the same lock. None of those pinned VTs can proceed until they acquire the lock. It's a variation of the classic deadlock problem, but instead of 2 locks we have one lock and a semaphore with 4 permits as represented by the fork-join pool.

Now that we know exactly what happened, it was easy to come up with a reproducible test case.

Conclusion

Virtual threads are expected to improve performance by reducing overhead related to thread creation and context switching. Despite some sharp edges as of Java 21, virtual threads largely deliver on their promise. In our quest for more performant Java applications, we see further virtual thread adoption as a key towards unlocking that goal. We look forward to Java 23 and beyond, which brings a wealth of upgrades and hopefully addresses the integration between virtual threads and locking primitives.

This exploration highlights just one type of issue that performance engineers solve at Netflix. We hope this glimpse into our problem-solving approach proves valuable to others in their future investigations.

Announcing bpftop: Streamlining eBPF performance optimization

netflixtechblog.com/announcing-bpftop-streamlining-ebpf-performance-optimization-6a727c1ae2e5

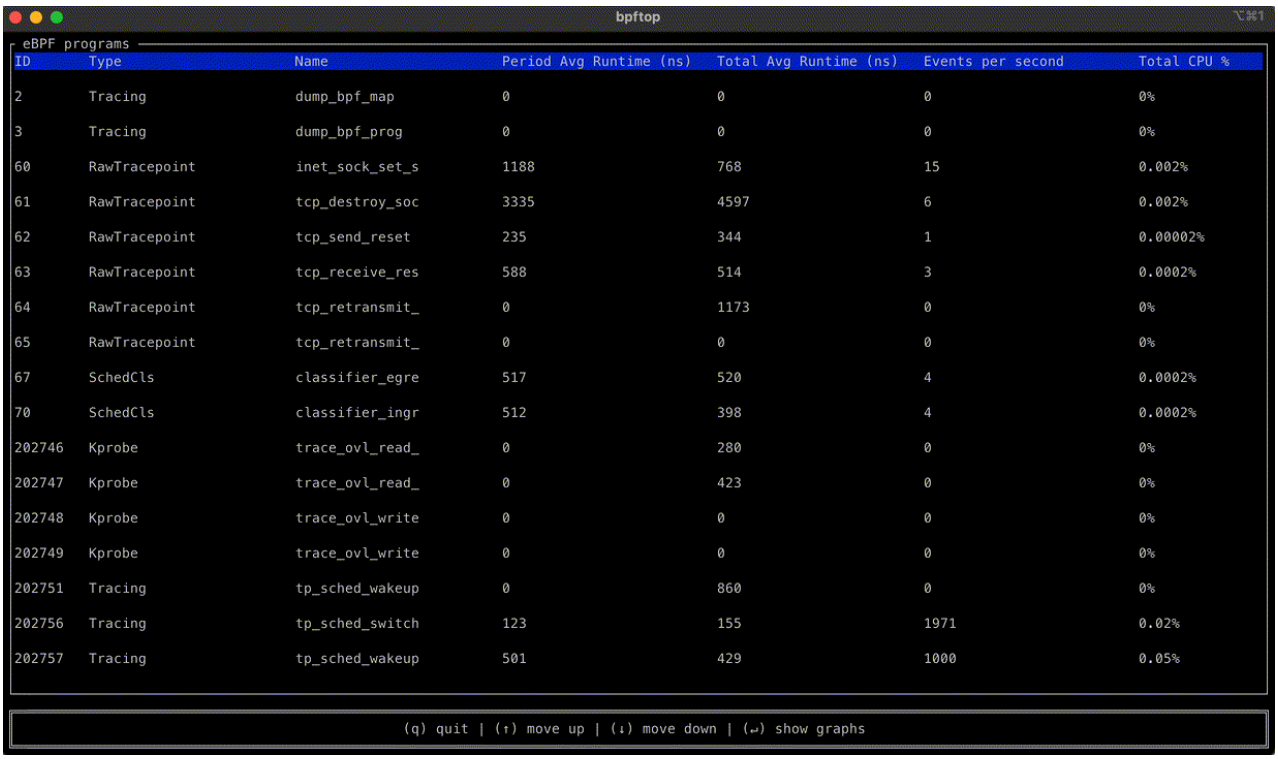
Netflix Technology Blog February 26, 2024

Learn about Netflix's world class engineering efforts, company culture, product developments and more.

By

Today, we are thrilled to announce the release of bpftop, a command-line tool designed to streamline the performance optimization and monitoring of eBPF programs. As Netflix increasingly adopts eBPF [1, 2], applying the same rigor to these applications as we do to other managed services is imperative. Striking a balance between eBPF's benefits and system load is crucial, ensuring it enhances rather than hinders our operational efficiency. This tool enables Netflix to embrace eBPF's potential.

bpftop provides a dynamic real-time view of running eBPF programs. It displays the average execution runtime, events per second, and estimated total CPU % for each program. This tool minimizes overhead by enabling performance statistics only while it is active.



ID	Type	Name	Period Avg Runtime (ns)	Total Avg Runtime (ns)	Events per second	Total CPU %
2	Tracing	dump_bpf_map	0	0	0	0%
3	Tracing	dump_bpf_prog	0	0	0	0%
60	RawTracepoint	inet_sock_set_s	1188	768	15	0.002%
61	RawTracepoint	tcp_destroy_soc	3335	4597	6	0.002%
62	RawTracepoint	tcp_send_reset	235	344	1	0.00002%
63	RawTracepoint	tcp_receive_res	588	514	3	0.0002%
64	RawTracepoint	tcp_retransmit_	0	1173	0	0%
65	RawTracepoint	tcp_retransmit_	0	0	0	0%
67	SchedCls	classifier_egre	517	520	4	0.0002%
70	SchedCls	classifier_ingr	512	398	4	0.0002%
202746	Kprobe	trace_ovl_read_	0	280	0	0%
202747	Kprobe	trace_ovl_read_	0	423	0	0%
202748	Kprobe	trace_ovl_write	0	0	0	0%
202749	Kprobe	trace_ovl_write	0	0	0	0%
202751	Tracing	tp_sched_wakeup	0	860	0	0%
202756	Tracing	tp_sched_switch	123	155	1971	0.02%
202757	Tracing	tp_sched_wakeup	501	429	1000	0.05%

bpftop simplifies the performance optimization process for eBPF programs by enabling an efficient cycle of benchmarking, code refinement, and immediate feedback. Without bpftop, optimization efforts would require manual calculations, adding unnecessary complexity to the process. With bpftop, users can quickly establish a baseline, implement improvements, and verify enhancements, streamlining the process.

A standout feature of this tool is its ability to display the statistics in time series graphs. This approach can uncover patterns and trends that could be missed otherwise.

How it works

bpftop uses the BPF_ENABLE_STATS syscall command to enable global eBPF runtime statistics gathering, which is disabled by default to reduce performance overhead. It collects these statistics every second, calculating the average runtime, events per second, and estimated CPU utilization for each eBPF program within that sample period. This information is displayed in a top-like tabular format or a time series graph over a 10s moving window. Once bpftop terminates, it turns off the statistics-gathering function. The tool is written in Rust, leveraging the libbpf-rs and ratatui crates.

Getting started

Visit the project's GitHub page to learn more about using the tool. We've open-sourced bpftop under the Apache 2 license and look forward to contributions from the community.