Real World JFR

Experiences building and deploying a continuous profiler at scale

Jean-Philippe Bempel
@jpbempel

DATADOG

Agenda

- Quick introduction to JDK Flight Recorder (JFR)
- JFR at Datadog
- Lessons learnt

Quick Intro to the JDK Flight Recorder (JFR)

- Data Flight Recorder for the JVM
- Records information about the JVM and the application
- Low overhead
- Powerful APIs and Tooling
- Can be used to solve a range of different problems

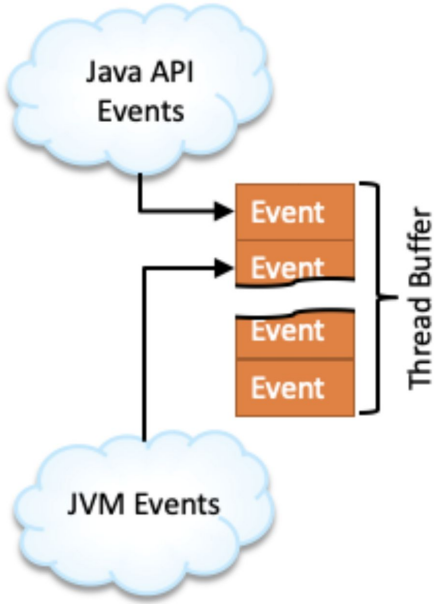
DATADOG

Availability of JFR

- OpenJDK/HotSpot
- Open sourced since JDK 11
- Backported to OpenJDK 8u262/272

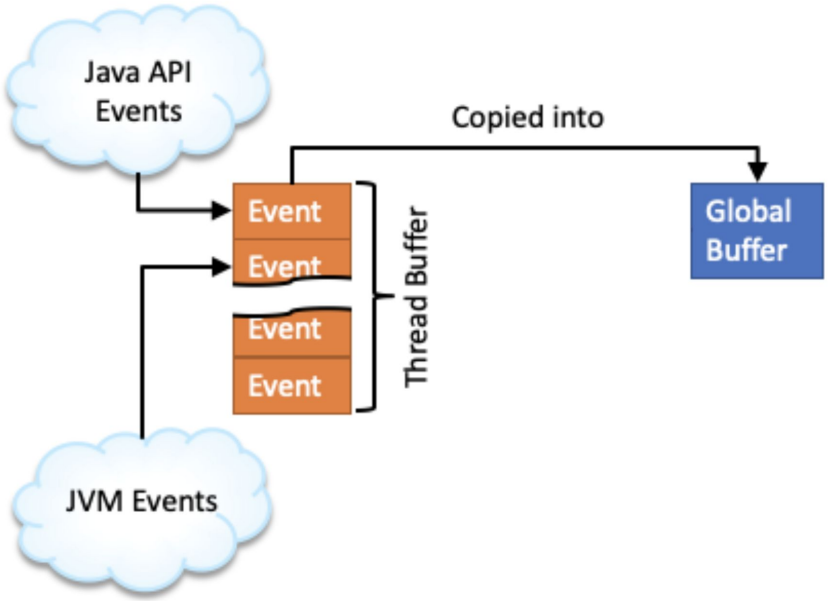
JFR Inner Workings

- Events recorded into thread buffers



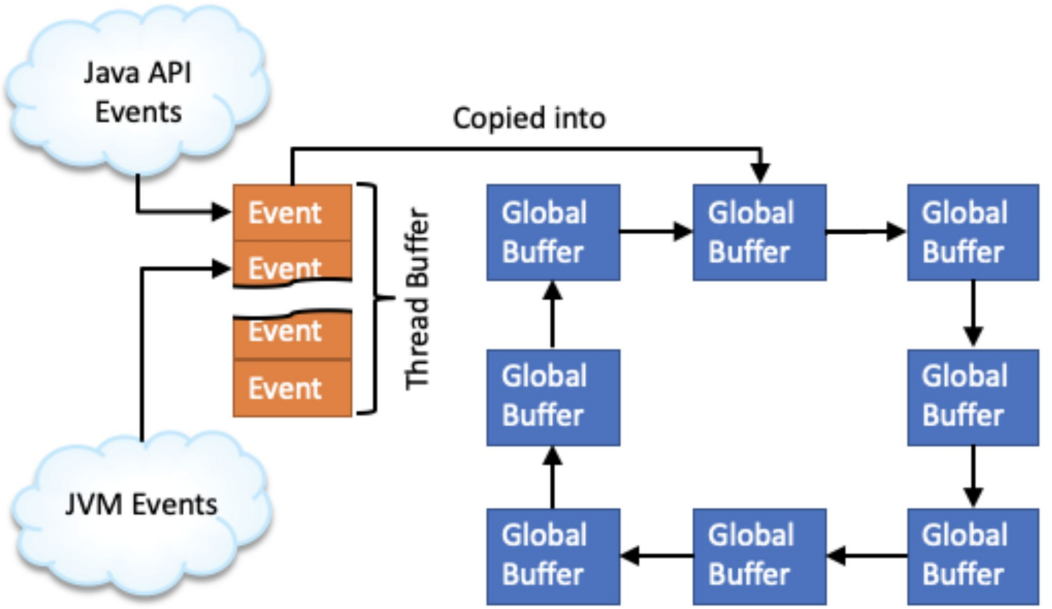
JFR Inner Workings

- Events recorded into thread buffers
- When full, copied into global buffer



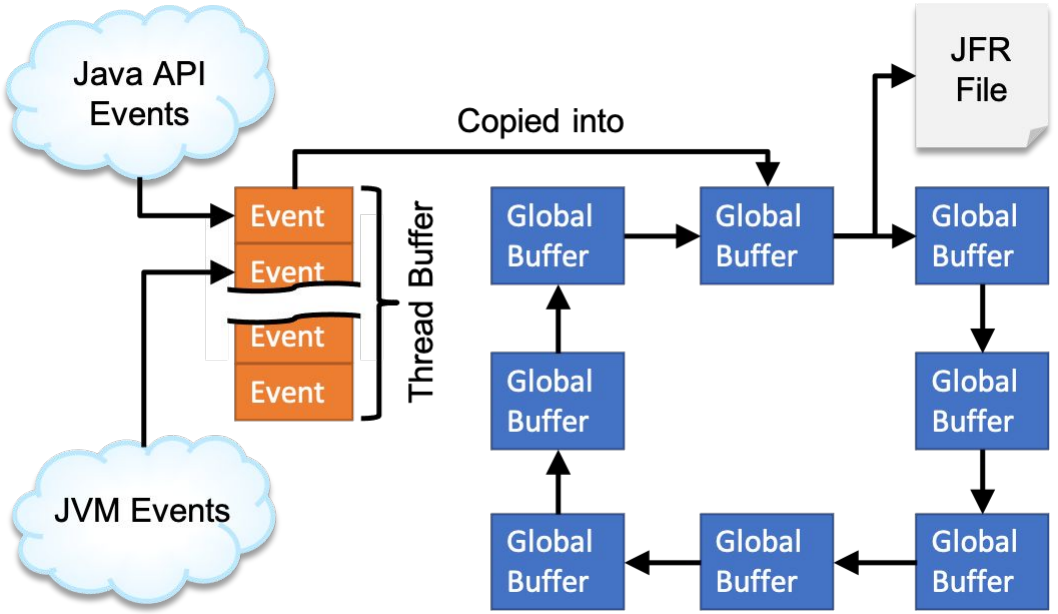
JFR Inner Workings

- Events recorded into thread buffers
- When full, copied into global buffer
- Can be configured to keep on overwriting/reusing buffer



JFR Inner Workings

- Events recorded into thread buffers
- When full, copied into global buffer
- Can be configured to keep on overwriting/reusing buffer
- … or emit to disk

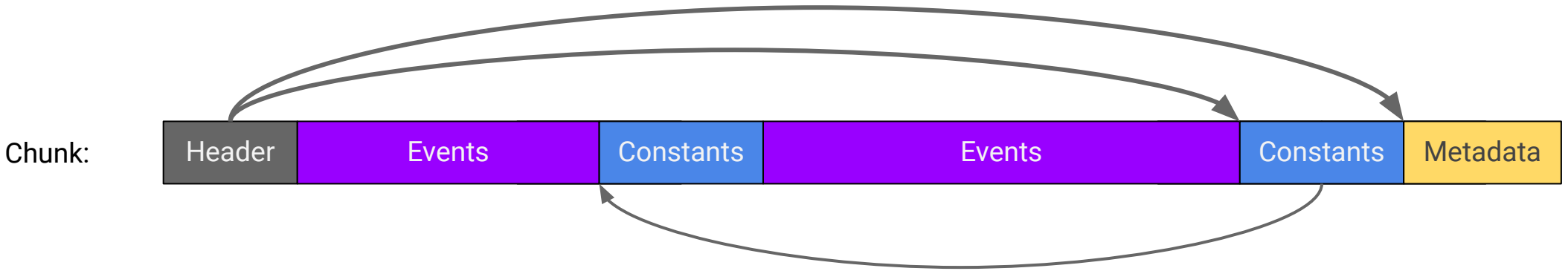


Low Overhead

- High performance flight recording engine
  - Invariant TSC for time stamping
  - Thread local native buffers
  - Efficient format for low overhead event emission
- High performance data collection
  - Access to data already collected in the Java runtime
  - Built into the JVM/JDK - skip abstractions
- Trying hard not to change runtime characteristics

Other Properties of JFR

- Self describing chunks of information
- Self contained chunks
- Chunk rotations will happen when
  - Start / Stop recording
  - Create a snapshot



Recording:

Quick Demo JFR

JFR at Datadog

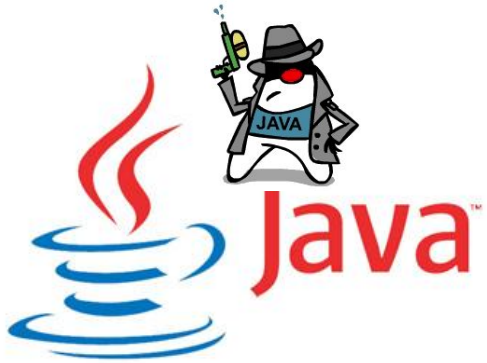
DATADOG

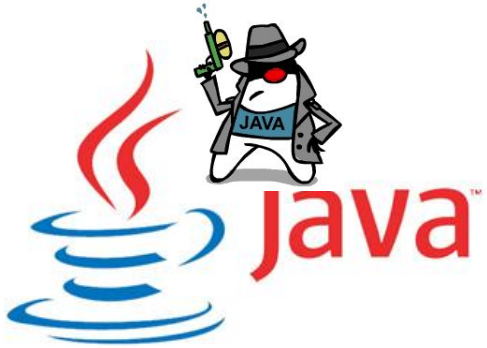
JFR at Datadog

- Continuous capture offers a lot of interesting capabilities
  - Always data available when things go bad
  - Possibility to break down profiling data
    - Time/Thread
    - Context
  - Continuous stream of data for analysis and statistics
- Is continuous capturing economically feasible?
  - What is the data rate?
  - What is the actual performance overhead?

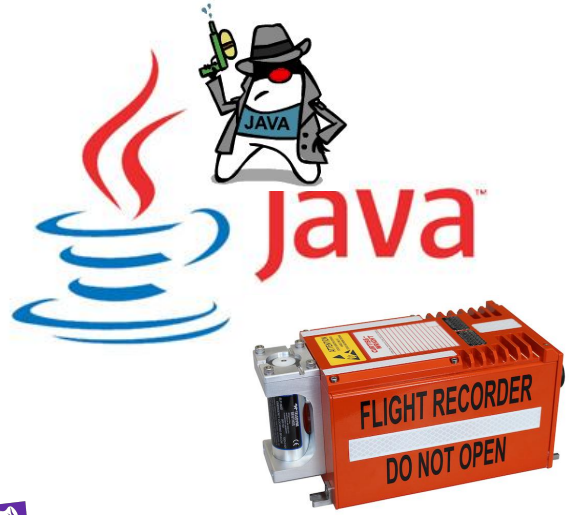
Continuous Profiler Architecture

Continuous Profiler Architecture

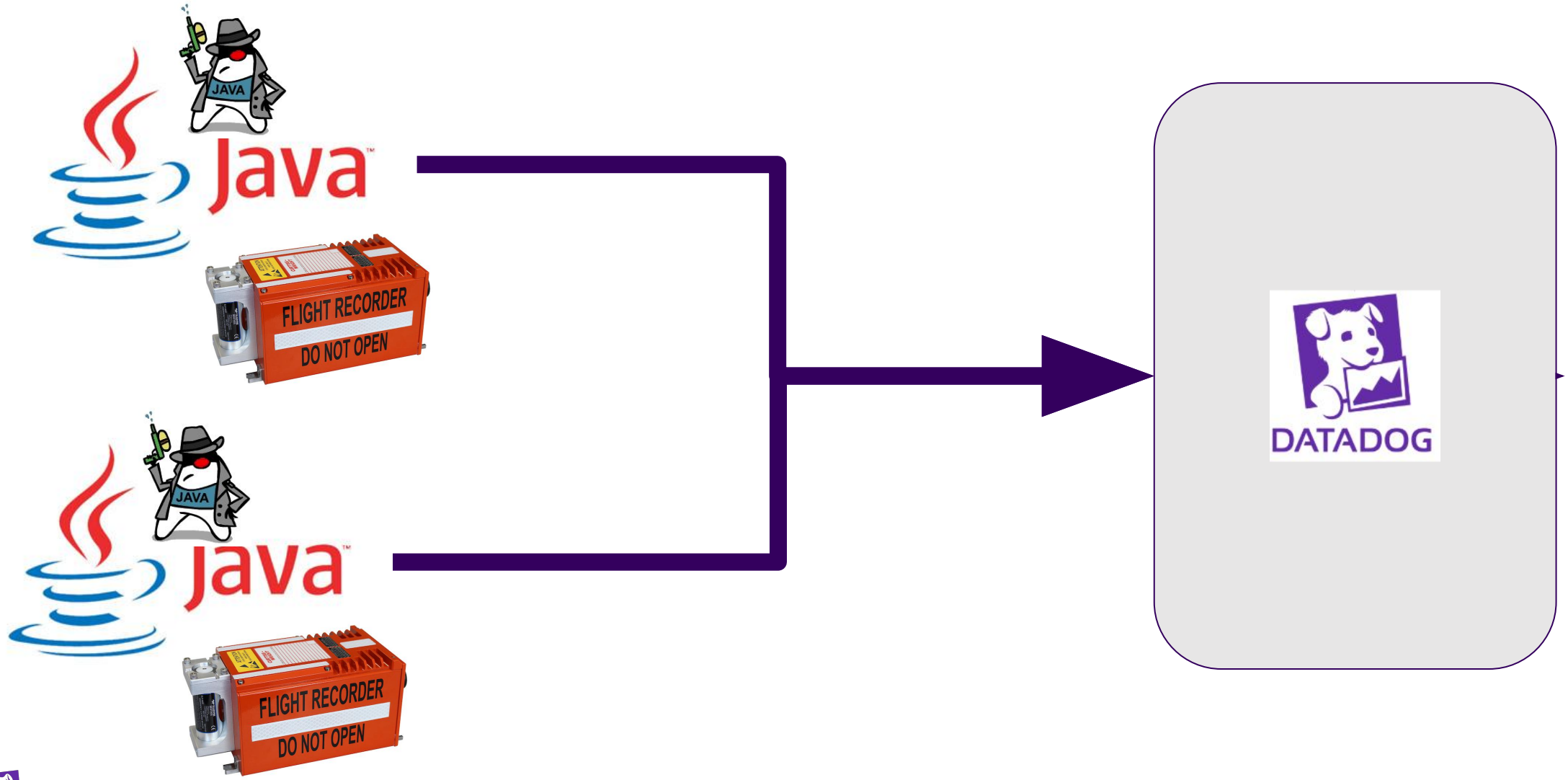




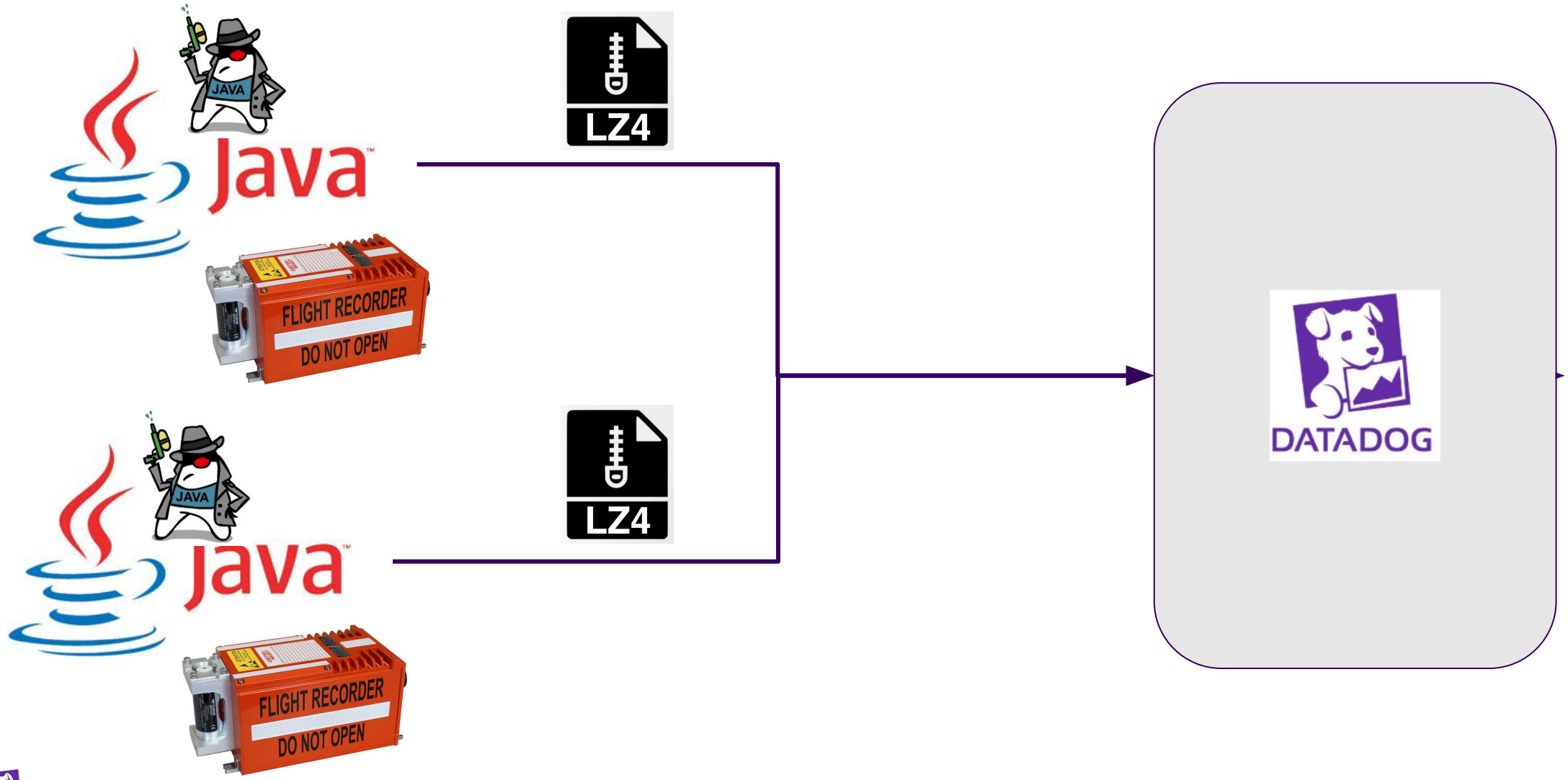
Continuous Profiler Architecture



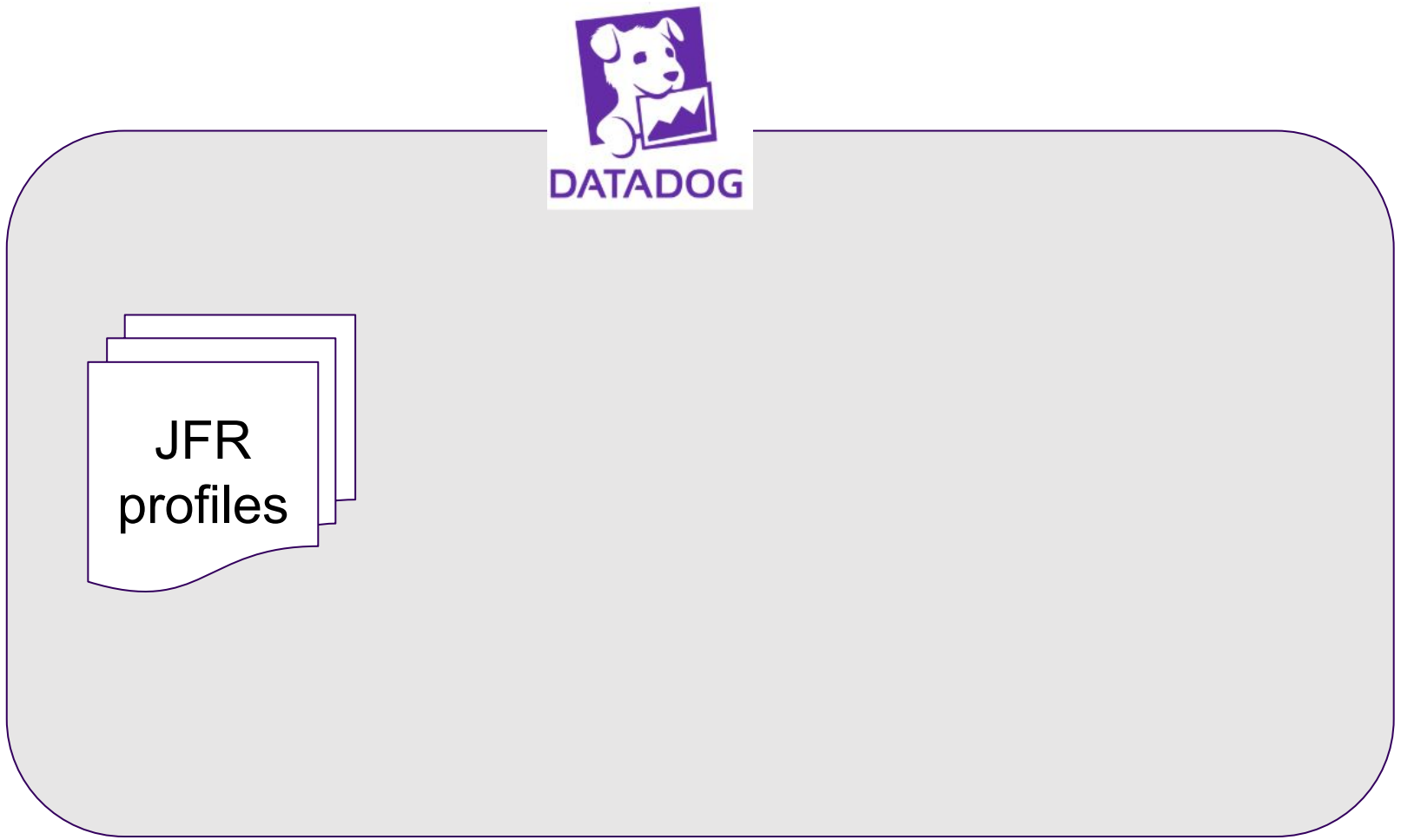
Continuous Profiler Architecture



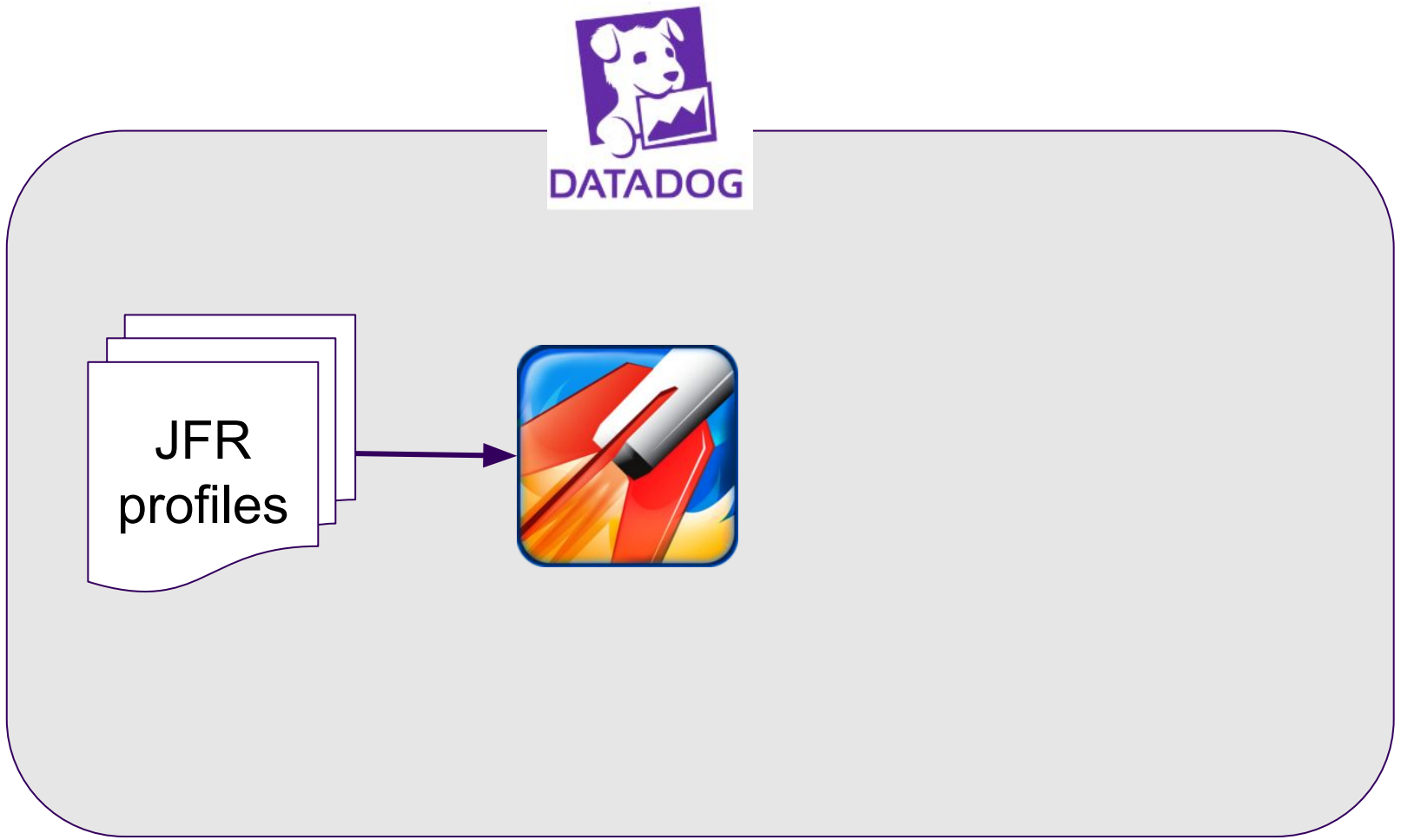
Continuous Profiler Architecture



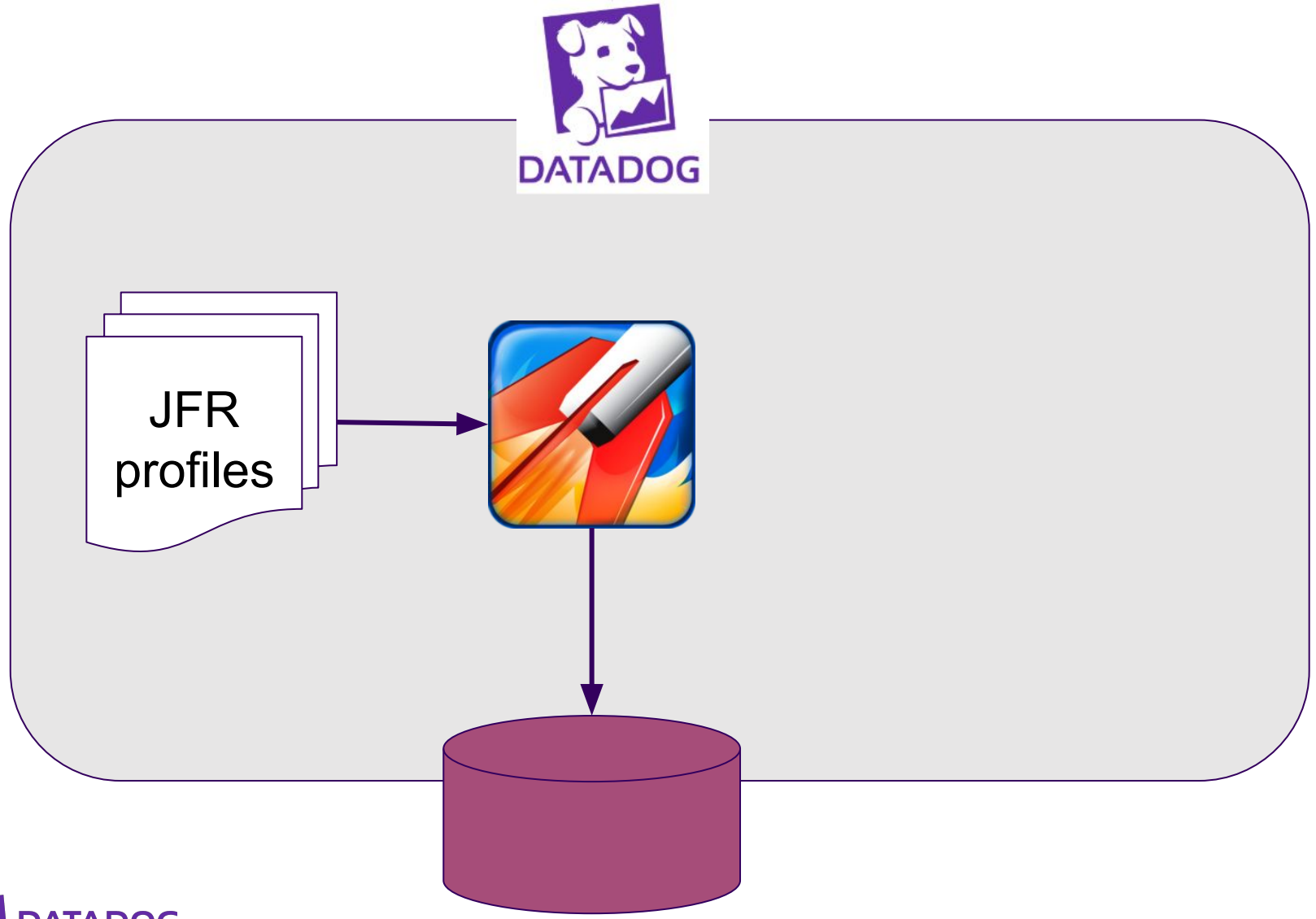
Continuous Profiler Architecture



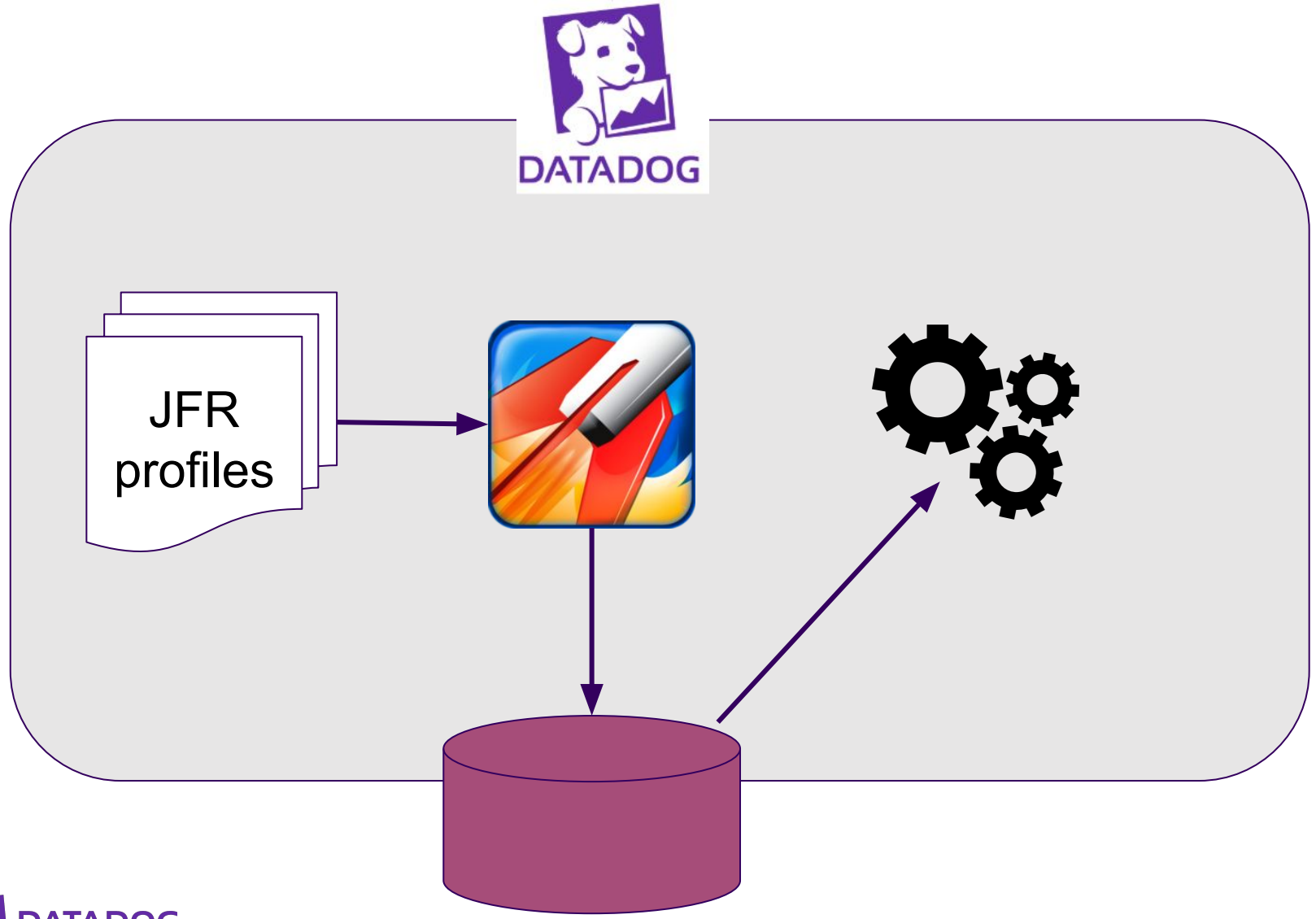
Continuous Profiler Architecture



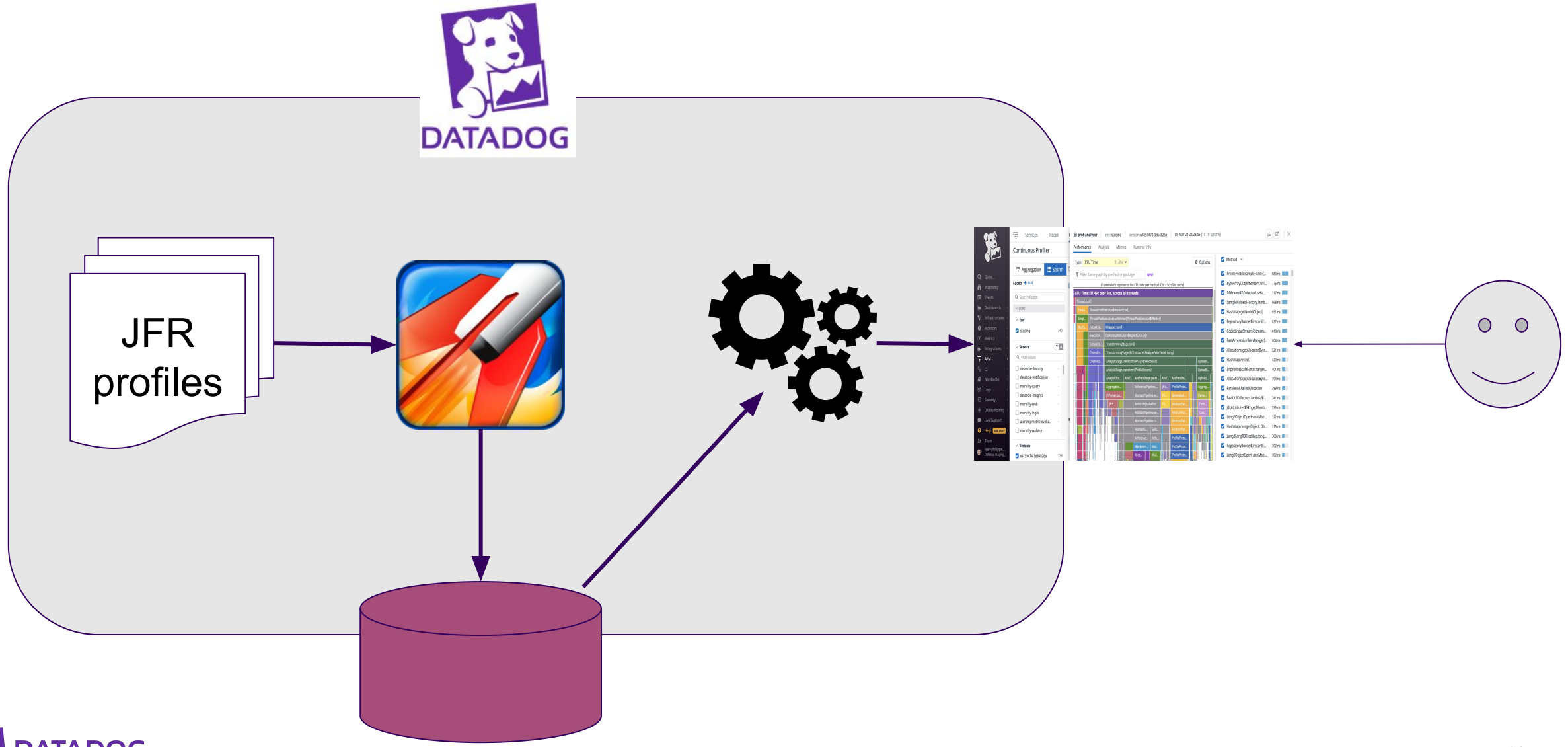
Continuous Profiler Architecture



Continuous Profiler Architecture



Continuous Profiler Architecture



Continuous Profiler figures

- Actually works surprisingly well at incredible scale
  - Datadog is intaking terabytes of JFR data per minute
  - Datadog is intaking all data from every Java process
- Recording data size 5 (2 compressed) MiB per minute
- Corresponds to around 100k events
- Normally one chunk per minute
- Cpu Overhead usually < 2%
- Cost for continuously repeating the metadata ~= 0.5%

JFR template specially crafted

- 2 flavors provided by OpenJDK

- Customize one with:
  - Allocation profiling
  - Exceptions
  - Exec Sampling 20 ->9 ms
  - Thresholds adjusted (VM operation, File IO, Monitors, Threads)

Overhead assessment

- Spring petclinic application
- request processing time too short
- More difficult to assess any overhead statistically significant
- Also not representative to real workload

Overhead assessment

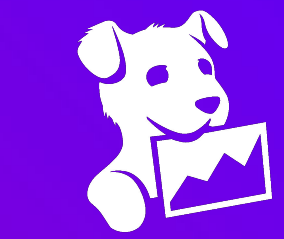
- Custom Spring petclinic
- Increase processing time for 100ms per request
- Increase In-memory database entries

Overhead measurements

- Heap & GC

- CPU from `/proc/<pid>/stat`
  get total cpu ticks since startup

Lessons Learnt



DATADOG

JFR at Scale

- All these observations are from using JFR at scale
- Very varying kinds of loads and applications
  - “Typical” long running Java microservices
  - Scala / Akka / high throughput messaging
  - Async / Reactive
- From casual use of the JDK Flight Recorder, you may not encounter any of these problems

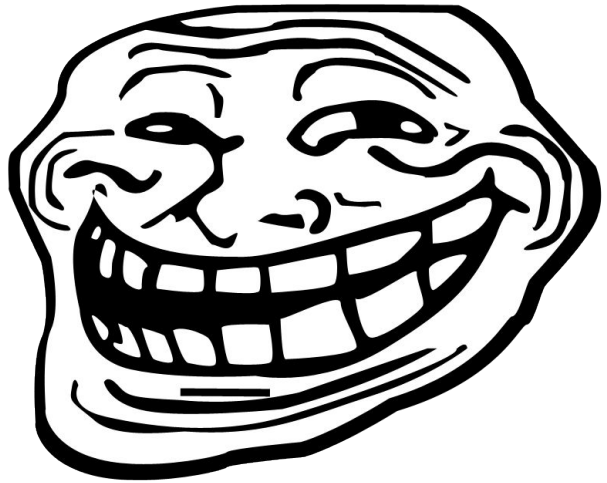
Exception Profiling

- Built in JFR exception profiler can be configured to capture all Exceptions or only Errors.
- Captures all, caught or uncaught
  (event generated on exception creation)
- Great to at least enable Errors?

"Error is the superclass of all the exceptions from which ordinary programs are not ordinarily expected to recover."

Errors (And Exceptions) Outside the Ivory Tower

- One of the most popular and widely used Java Libraries:
  - JavaCC => Lucene => Elasticsearch
  - Enormous amount of errors
  - Subclass named LookAheadSuccess
  - Used for control flow in a parser

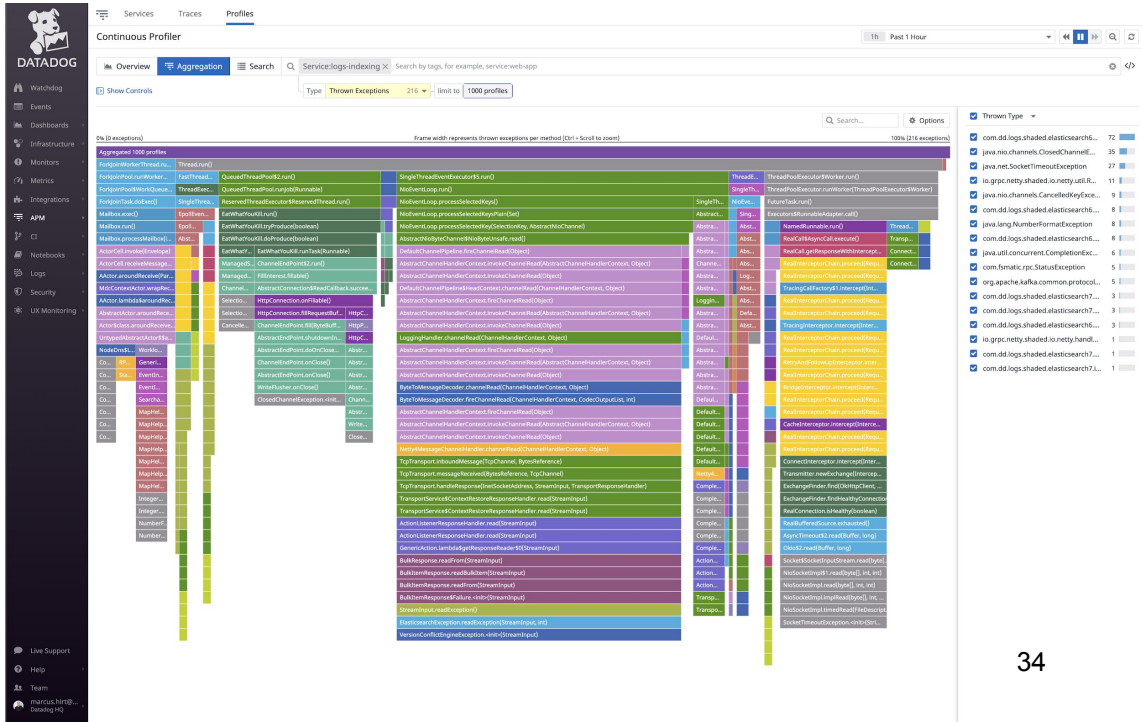


- But exception profiling is great!

  -> Invent new exception profiler

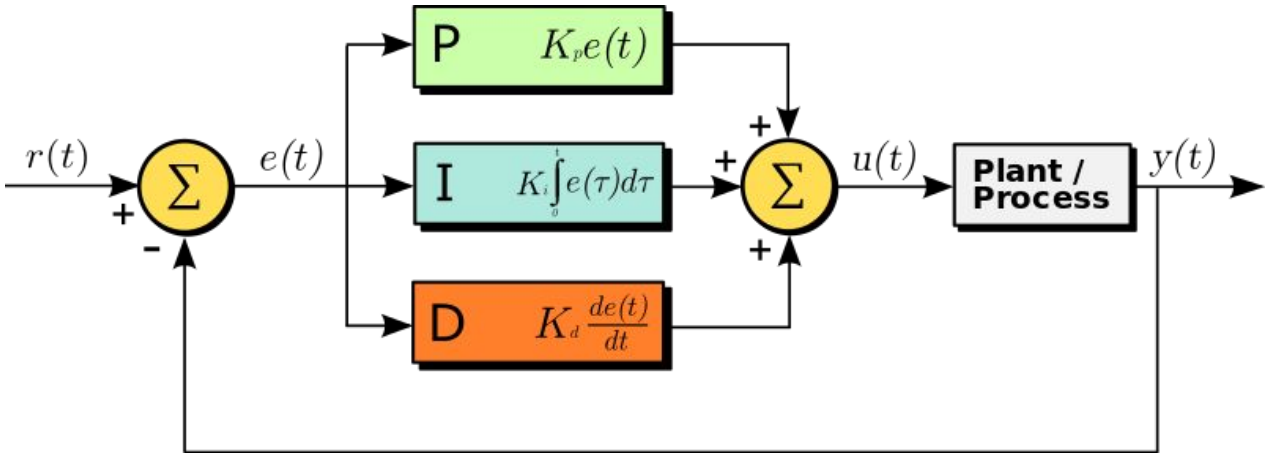
New Exception Profiler

- Get a count of exceptions per type
- Sample the first thrown exception of each type
- Subsample to try to hit a target rate
  - Use inspiration from PID controllers
  - Evenly spread across time

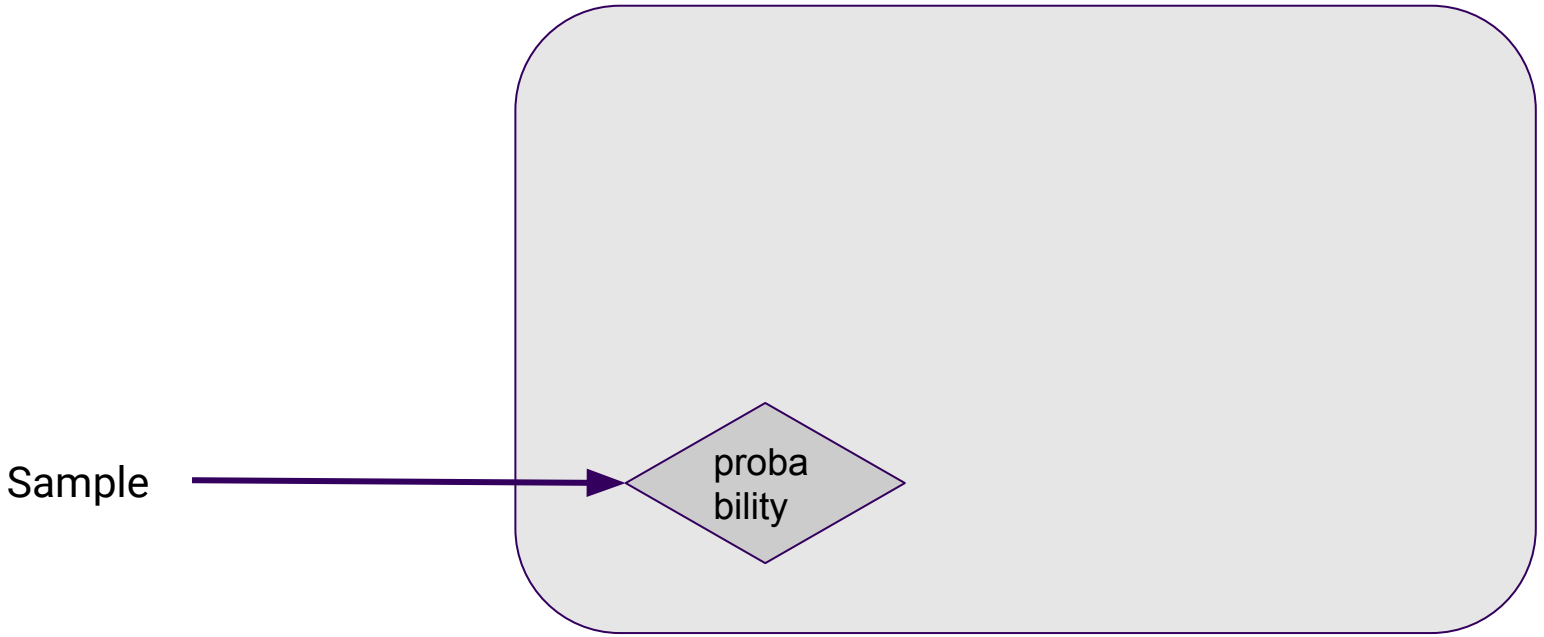


PID controller

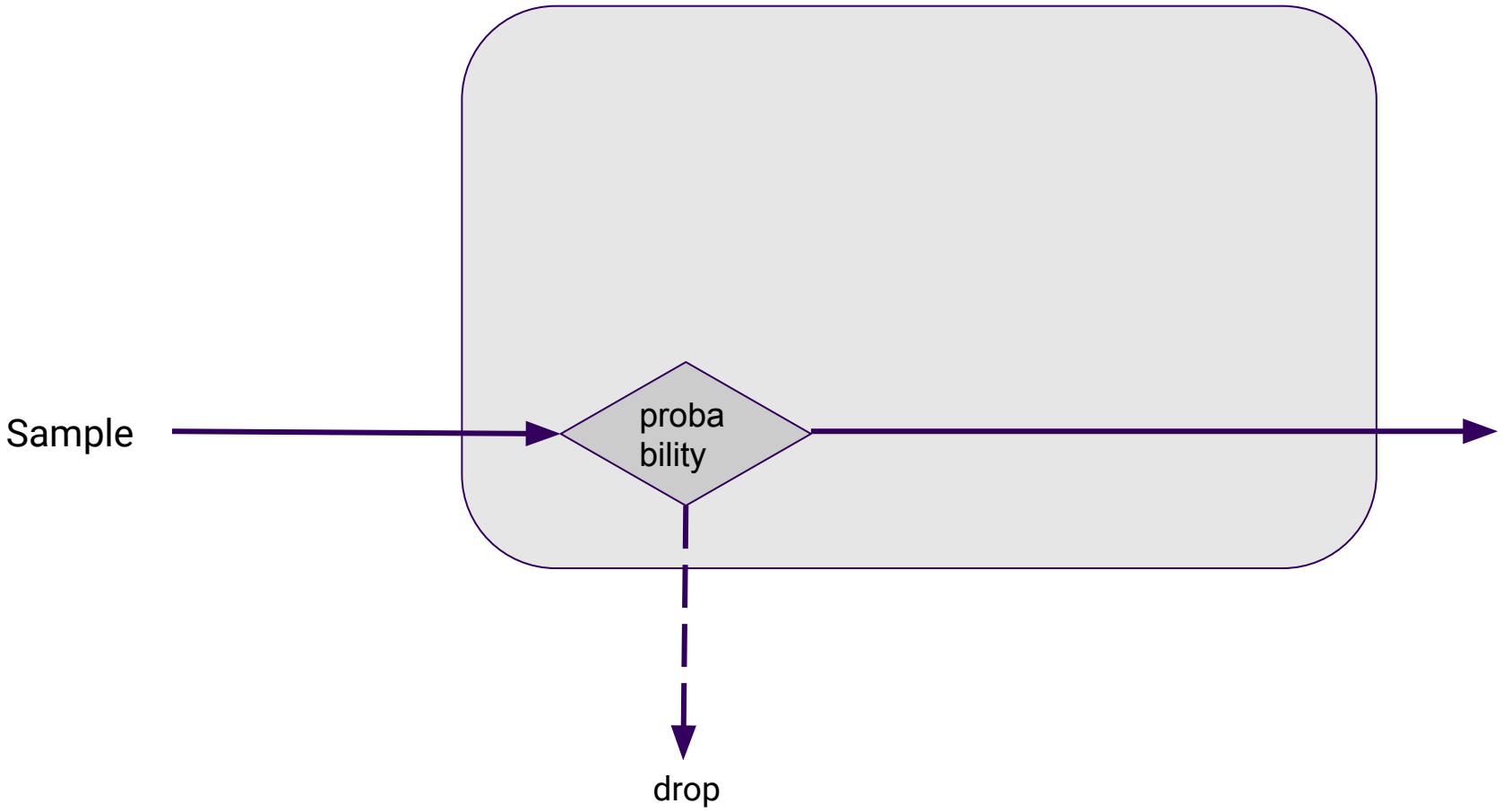
- Control loop mechanism
- using feedback loop
- apply correction based on error with P, I D terms



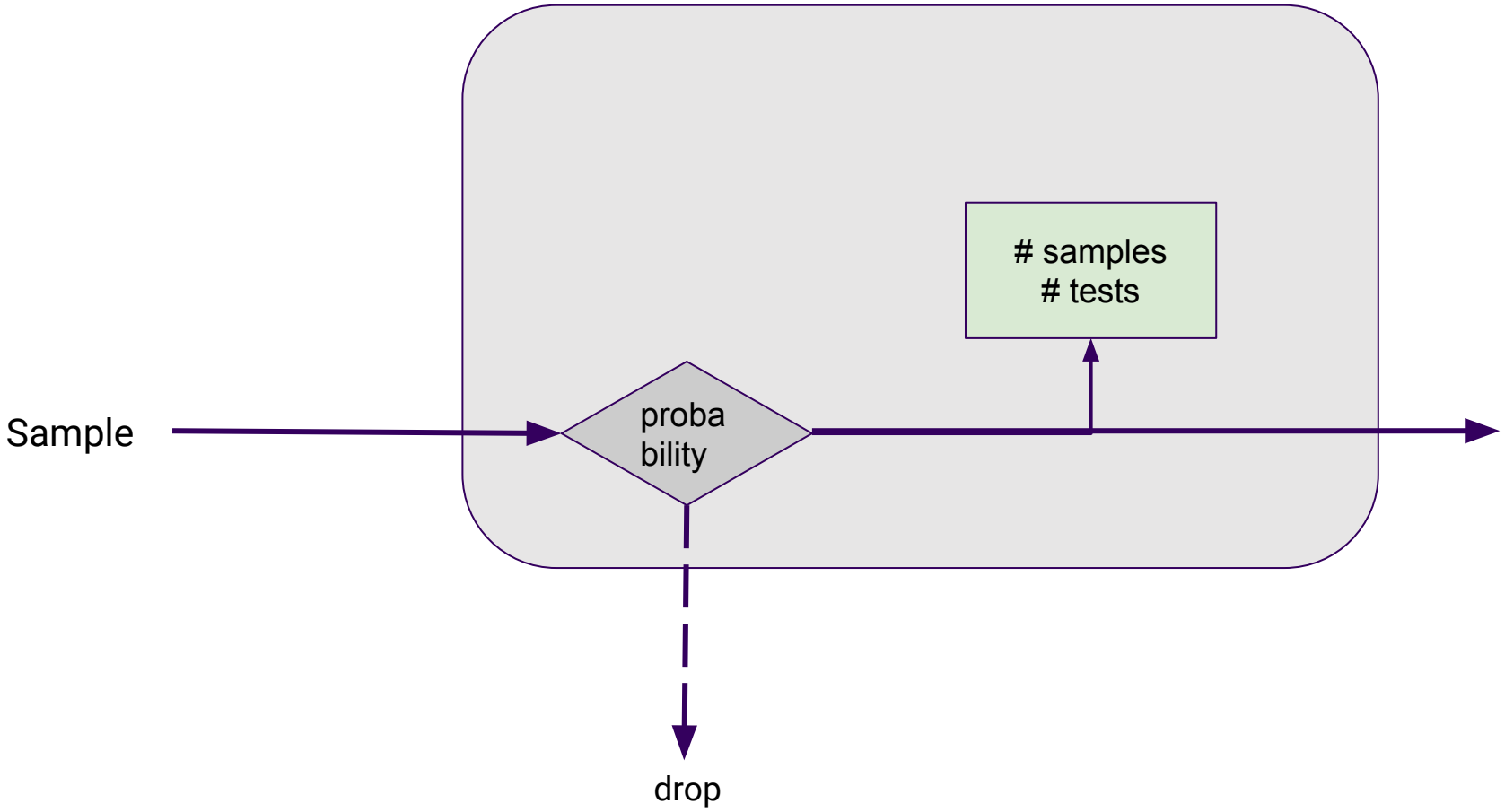
Adaptive Sampler



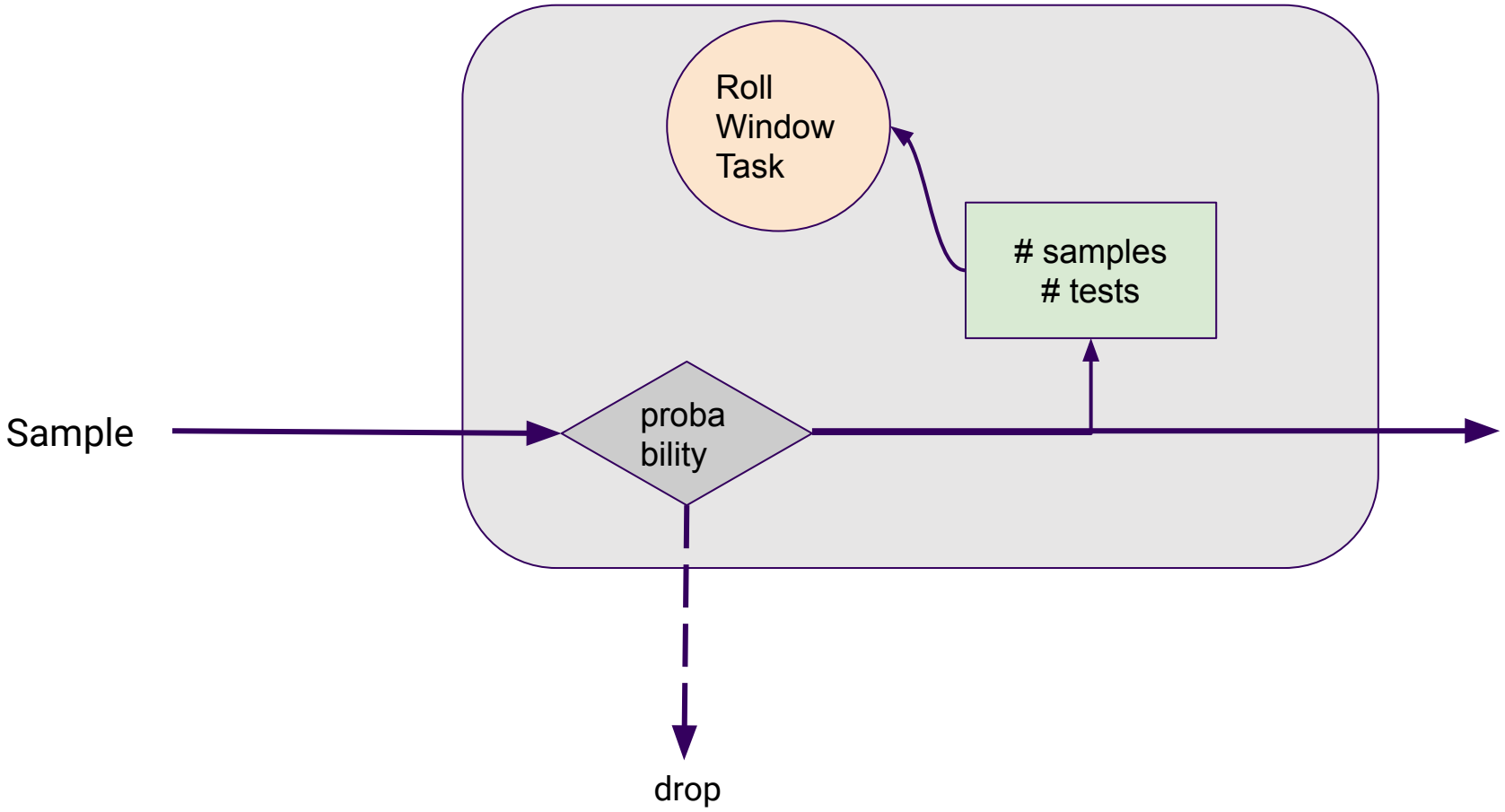
Adaptive Sampler



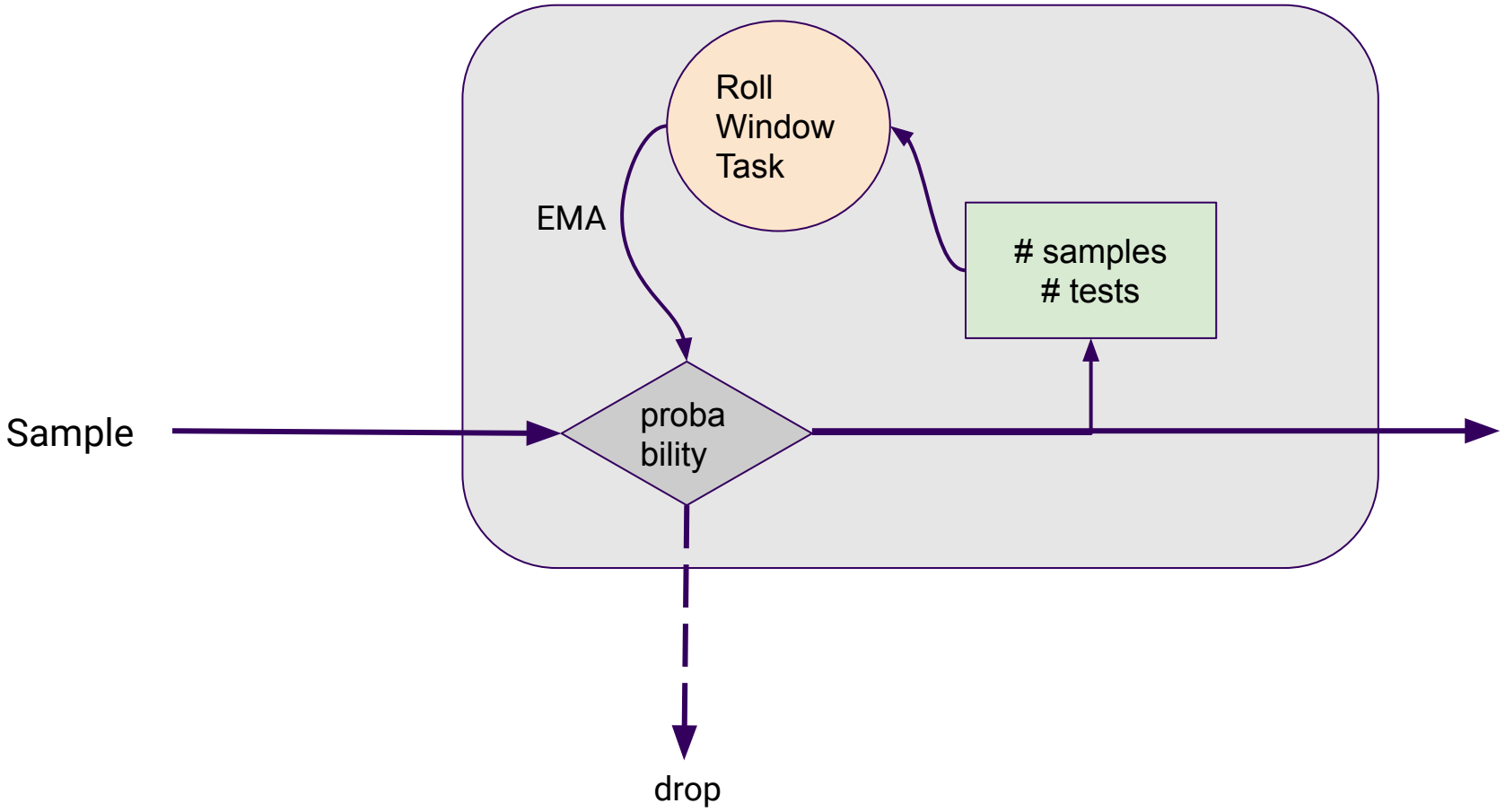
Adaptive Sampler



Adaptive Sampler



Adaptive Sampler

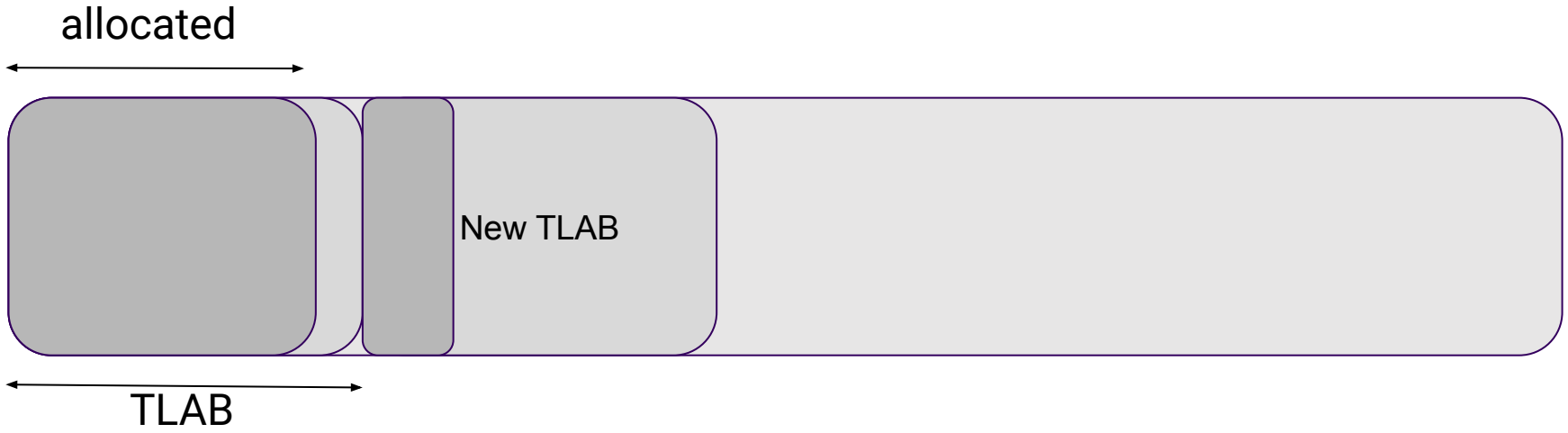


Allocation Profiling in JFR

- Allocation profiling introduced in JFR
  - Introduced in 7u40 (2013)
  - Has two paths/events - New TLAB / outside TLAB

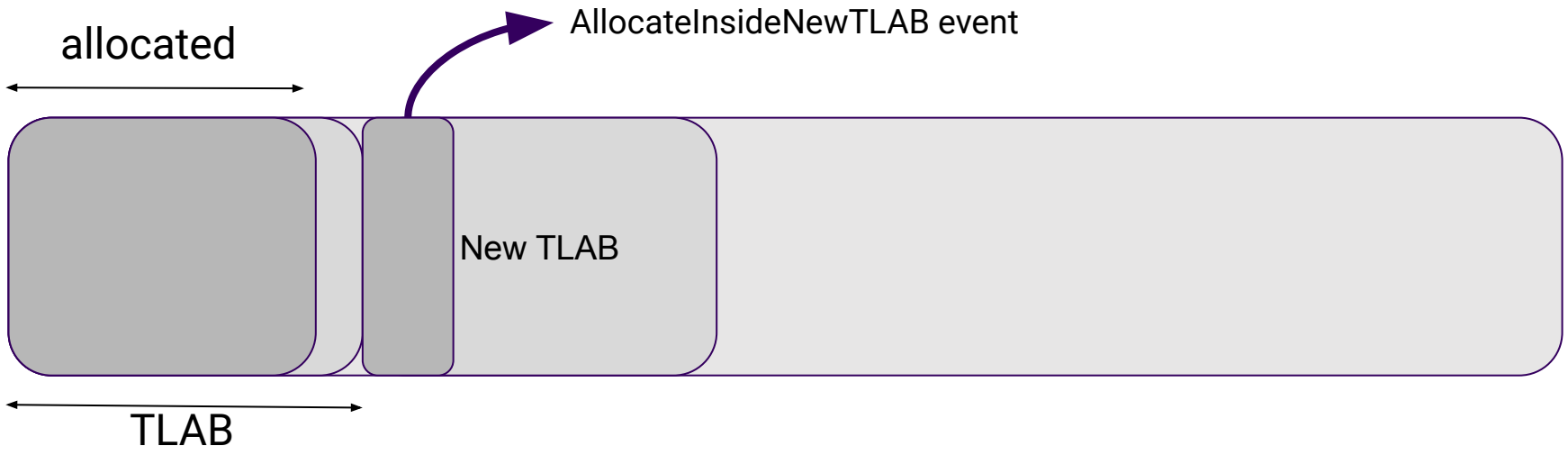
Allocation Profiling in JFR

- Allocation profiling introduced in JFR
  - Introduced in 7u40 (2013)
  - Has two paths/events - New TLAB / outside TLAB



Allocation Profiling in JFR

- Allocation profiling introduced in JFR
  - Introduced in 7u40 (2013)
  - Has two paths/events - New TLAB / outside TLAB



The Dangers of Allocation Profiling

- Normally has quite good runtime performance and data production rate
- These days though...
  - 96+ core beasts
  - allocation hungry services (e.g. stream processing beasts)
- Event rate depends on factors like
  - number of threads
  - size and number of allocations

Allocation Profiling Performance Problems

```
jfr summary recording.jfr
```

```
 Version: 2.1
 Chunks: 2
 Start: 2020-07-22 07:23:25 (UTC)
 Duration: 60 s

 Event Type                                          Count  Size (bytes)
=================================================================================
 jdk.ObjectAllocationInNewTLAB                      437808      10011153
 jdk.ZThreadPhase                                    25780       1142331
 jdk.ThreadPark                                      18614        802953
 jdk.ExecutionSample                                 13509        233323
 jdk.ObjectAllocationOutsideTLAB                     10620        231453
 datadog.ExceptionSample                              9421        348824
 jdk.JavaMonitorWait                                  5199        171686
 jdk.NativeMethodSample                               4293         75900
 jdk.ThreadSleep                                      3244         68124
 jdk.ClassLoaderStatistics                            1468         43716
 jdk.BooleanFlag                                      1236         43028
 jdk.ThreadAllocationStatistics                        809         13859
 jdk.ThreadCPULoad                                     619         13449
```

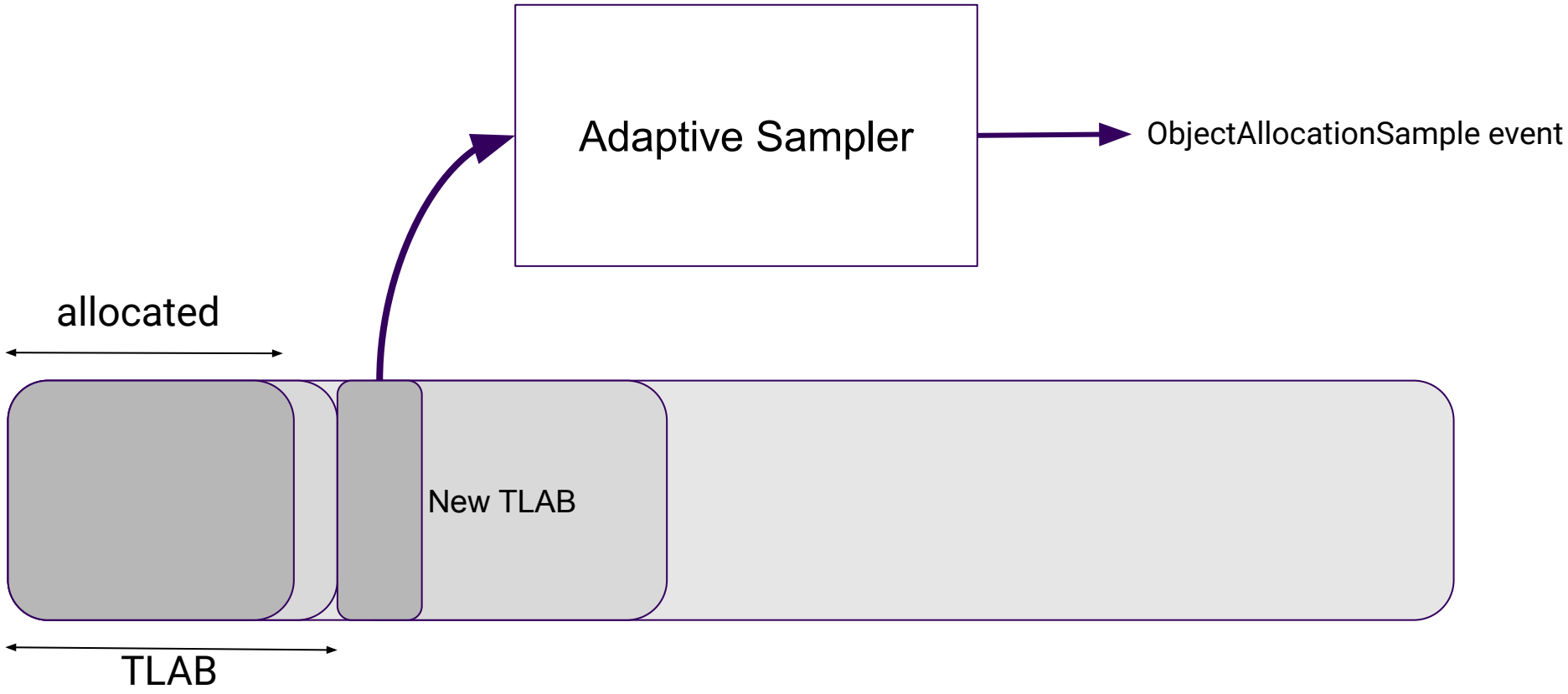
Allocation Profiling Performance Problems

- Hashcode problem - JFR constant pool (solved)
- Still too much data to handle
- Solution -> new allocation profiler in JFR (JDK 16+)

New Allocation Profiler

- Take the idea from the JVMTI (JDK 11+) allocation sampler (i.e. average amount of memory between samples)
- Inspiration from PID controllers - control data production rate
- Many nice qualities:
  - Controllable data budget
  - Actual individual samples (time, thread)
  - Allocation since last sample for weighting (total allocation pressure)

New Allocation Profiler

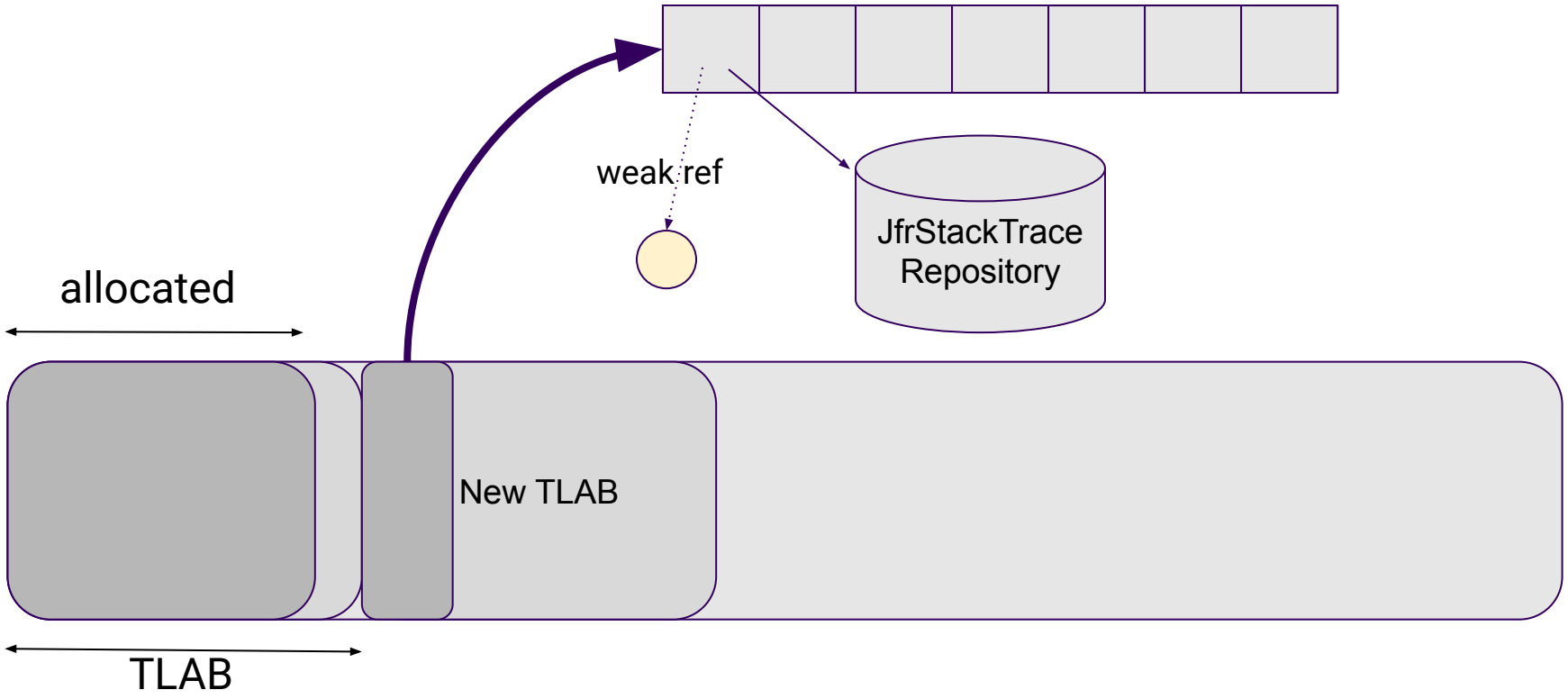


Memory leak profiler: OldObjectSample

- Interesting for solving memory leaks
  - Allocation Stacktrace
  - Allocation Time
  - Type
  - Array size
  - Reference chain

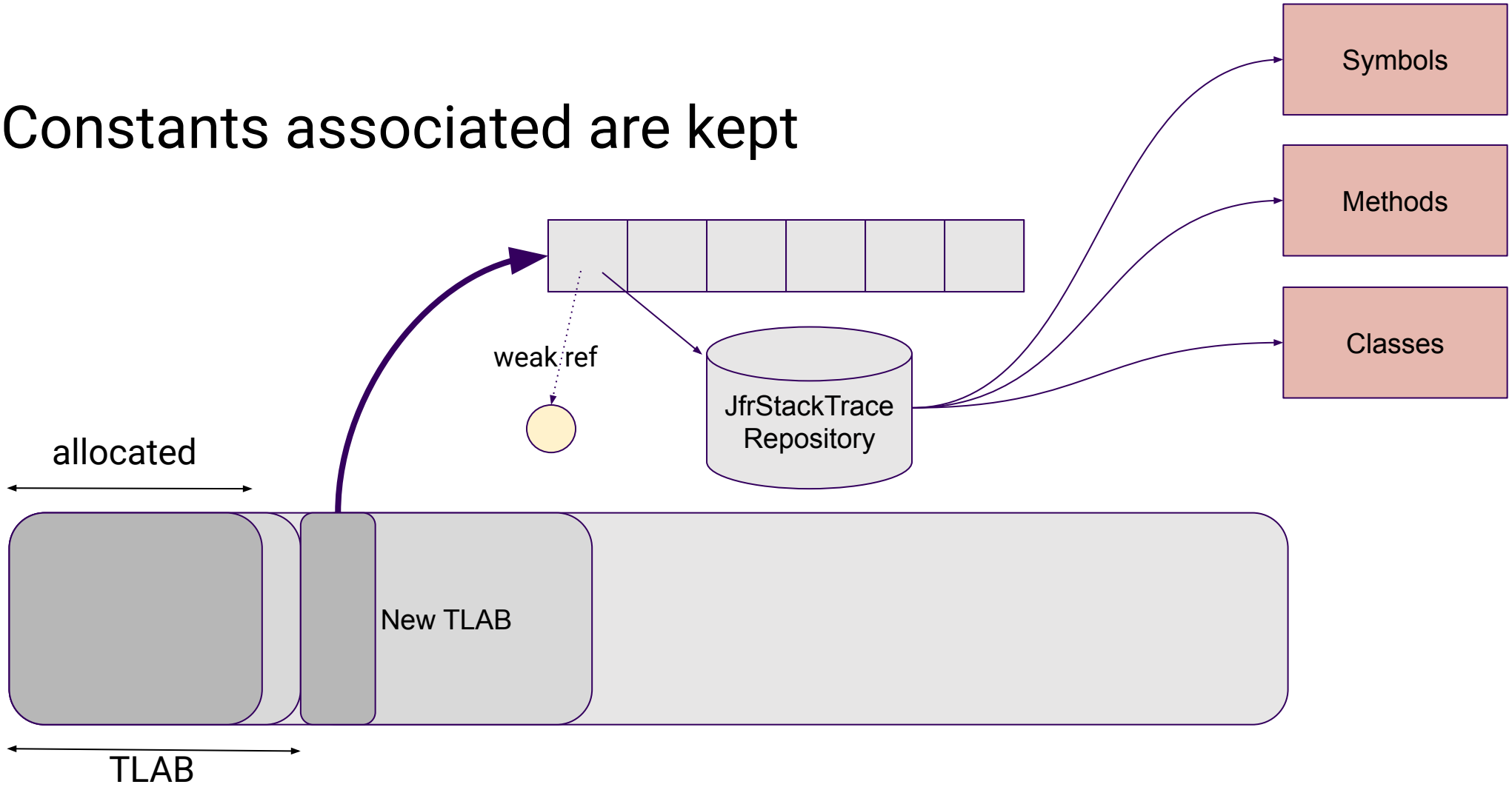
OldObjectSample Problems

- Allocation sample are kept into a queue



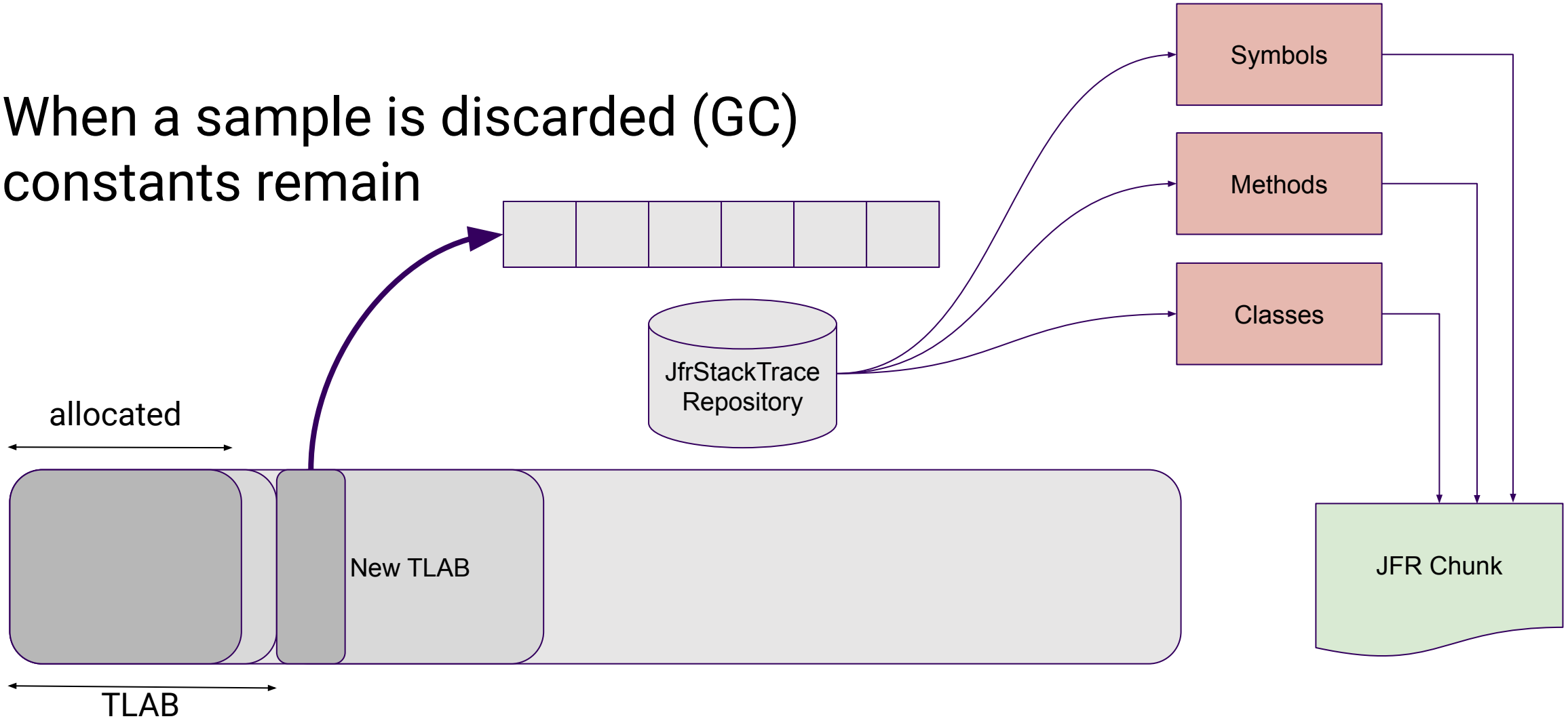
OldObjectSample Problems

- Constants associated are kept

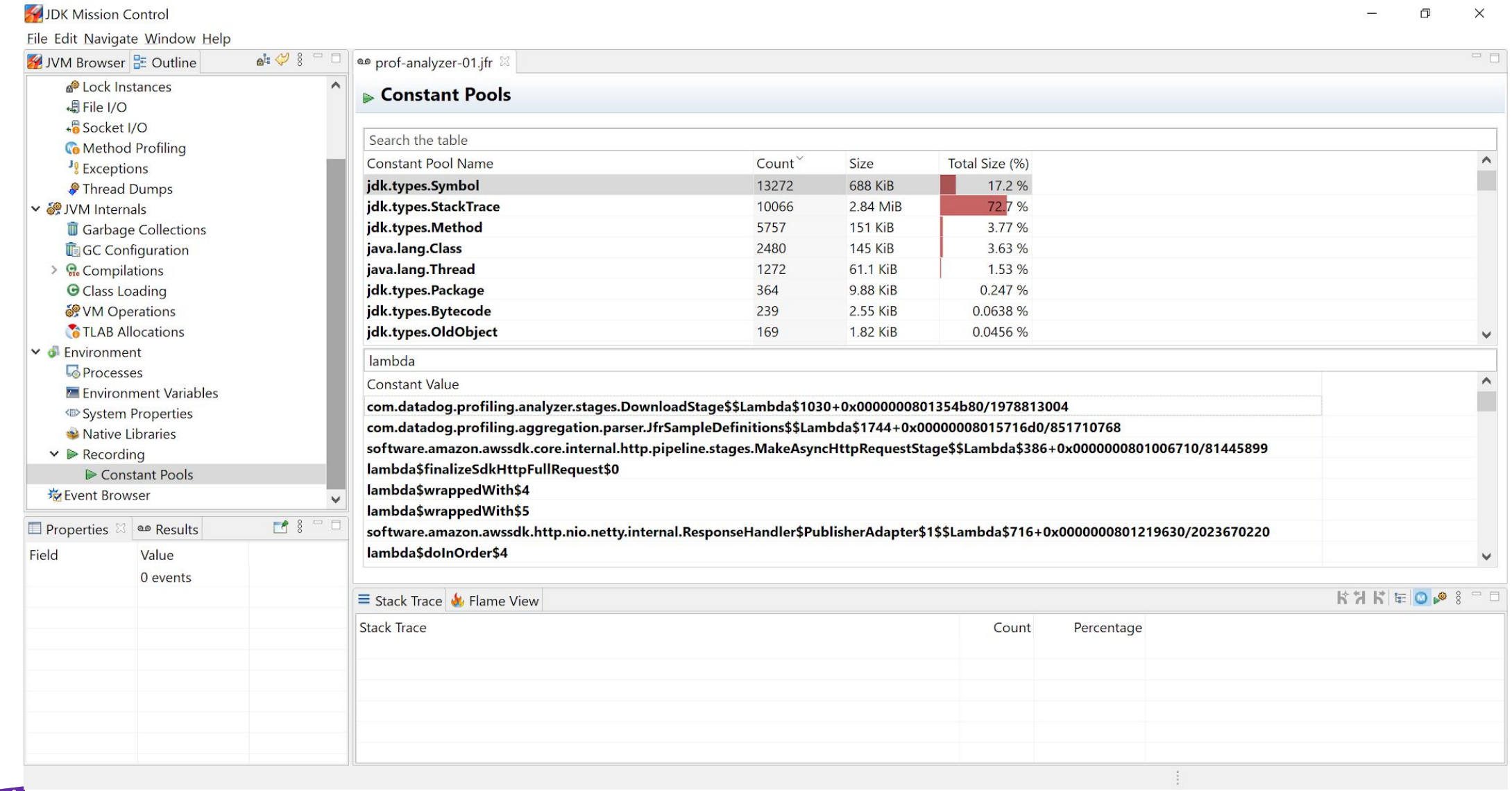


OldObjectSample Problems

- When a sample is discarded (GC) constants remain

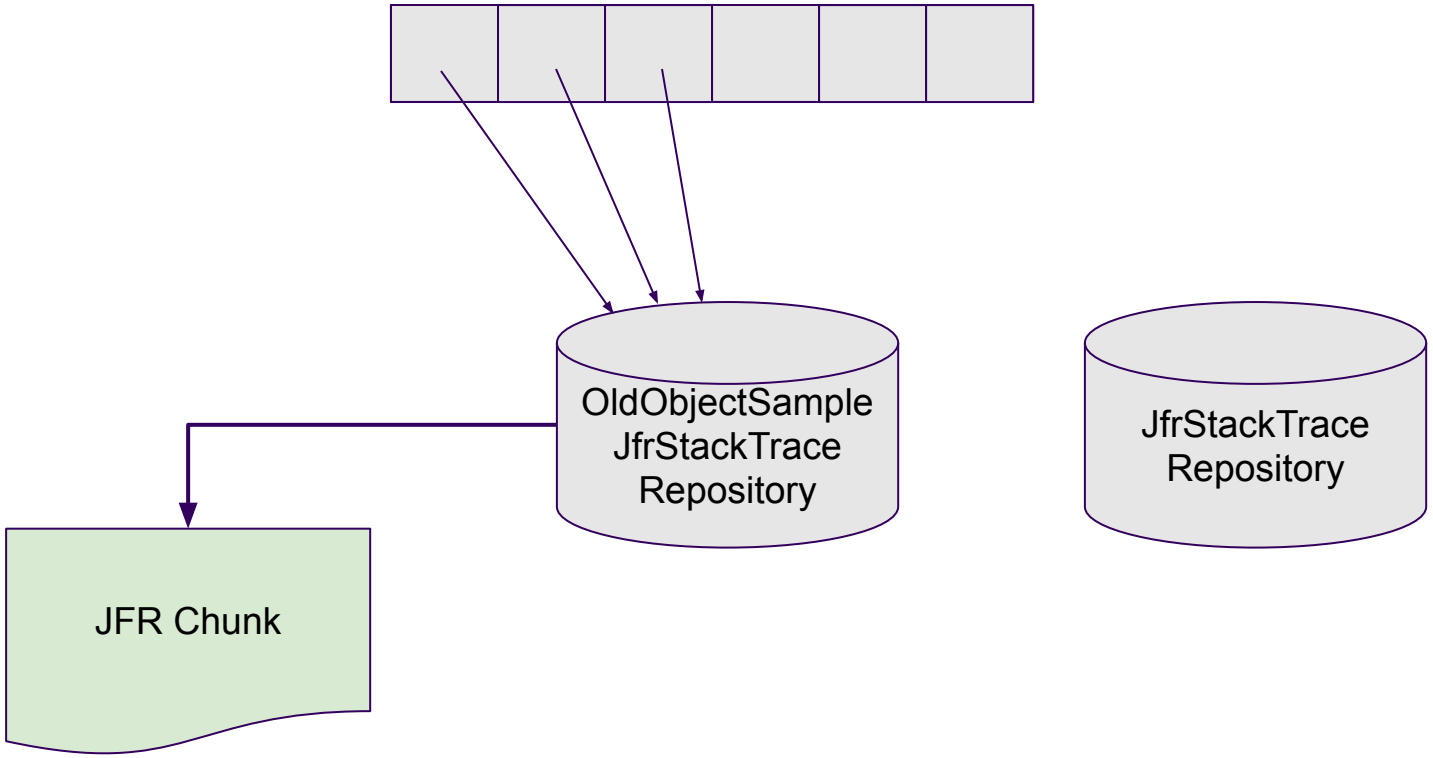


ConstantPool explorer



OldObjectSample Solutions

- Having a second stacktrace repository specific for the event



CPU Profiling

Execution Sample events

- Pros
  - Very cheap - both in memory and overhead
  - Pretty much constant overhead
  - Not safepoint biased (like AsyncProfiler)
- Cons
  - Not sampling all threads
    (e.g. JVM native threads/native library threads)

-> Compensate unaccounted CPU time using other events

CPU Profiling

- JFR would do well with a proper CPU profiler
  - Sample taken when certain CPU time elapsed...
  - ...no matter the thread
  - Nice APIs available today... (e.g. perf_event_open)
  - ...backed by PMU (not in containers though)

Latency outliers vs Wall-Clock

- JFR has events for thread halts
  - Provide more than just a stacktrace
  - Provide exact wall-clock timing of the halts
- Happen too often
  - Must be limited to outliers only
  - Uses thresholding to keep volume down
- Thresholding is problematic
  - Edge cases
  - Statistical skew
  - Hard to know if you've missed somethings

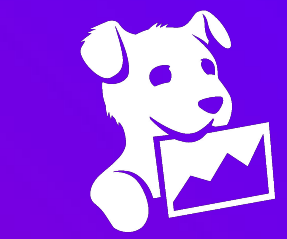
Best of Both Worlds?

- Subsample / Rate limit the events (PID thinking FTW)
- Add a proper wall clock profiler for JFR...

How about adding Event#commit(Thread)?

(Events might also want to add an annotation whether or not the thread state should be captured or not.)

Wrap Up



DATADOG

Summary

- JFR is more than a profiler and can be used in production
- Exception, allocation and leak profiler can increase overhead
- Solutions to this are coming to your JDK near you, soon!

JMC 8 is Released!

- Tutorial:
  https://github.com/thegreystone/jmc-tutorial
  (Feel free to fork and do pull requests for the Tutorial! :))

- JShell for JMC-core (jmc-jshell):
  https://github.com/thegreystone/jmc-jshell

References

- JDK Mission Control GitHub Repo
- Marcus Hirt's Blog
- Continuous Profiling Blog
- Improved JFR Allocation profiling in JDK 16
- Adaptive Sampler: Java impl & JFR impl
- OldObjectSample PR

Q&A

@jpbempel