
KubeCon
CloudNativeCon
North America 2024

KubeCon

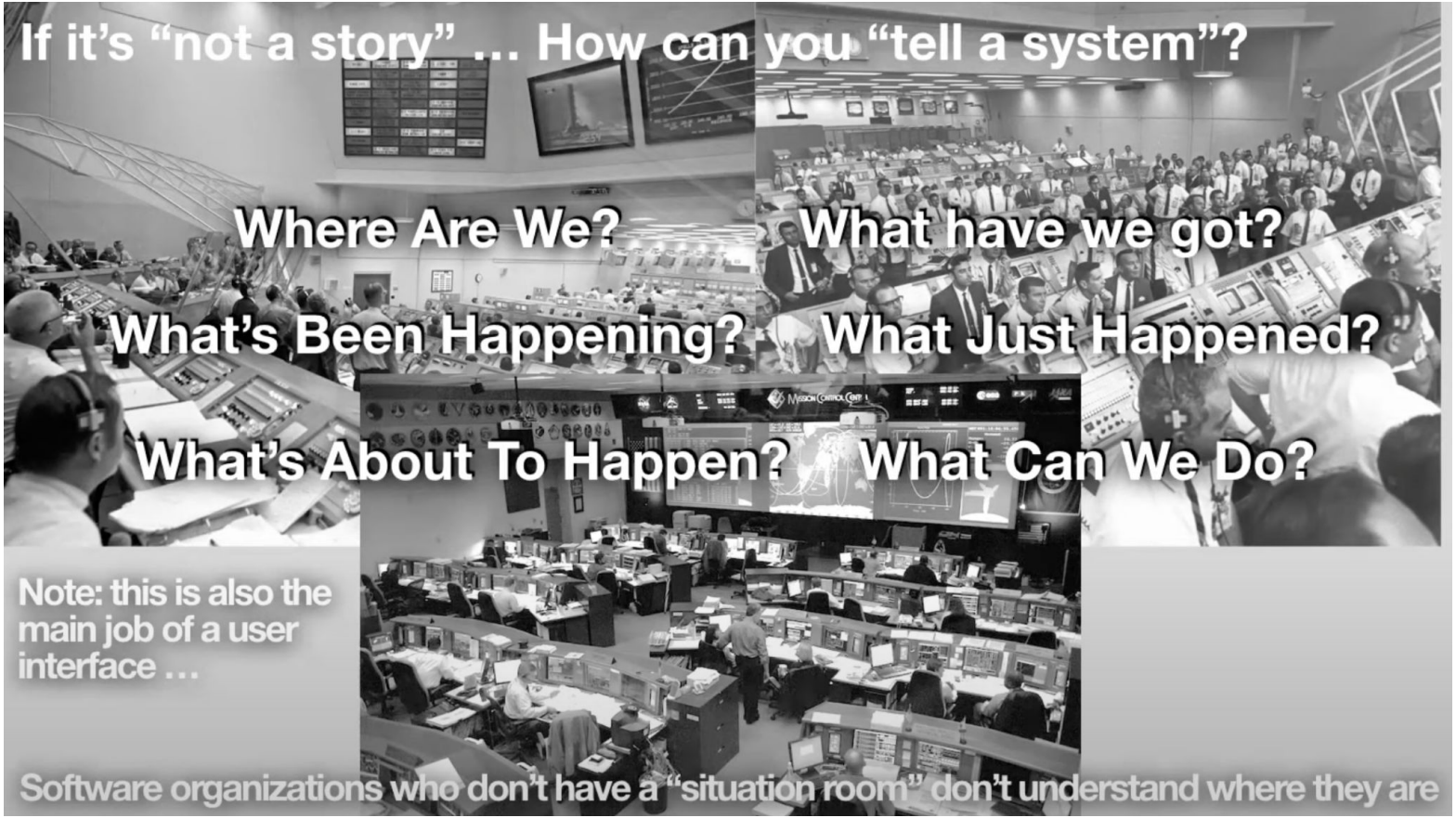
CloudNativeCon

North America 2024

Low-Overhead Zero-Instrumentation Continuous Profiling in OTel

Christos Kalkanis - Elastic

Continuous Profiling



Alan Kay - Making Progress: https://www.youtube.com/watch?v=9MqVfzxAp6A

History

- 2021: Optimyze.cloud launches low-overhead zero-instrumentation multi-runtime profiler
- Acquired by Elastic soon after
- 2024: Elastic donates profiling agent to OpenTelemetry
- Complements traditional observability solutions
- Continued development and evolution

Unobtrusive Frictionless Deployment

Powered by eBPF. Requires zero-instrumentation, no code changes or app restarts. Gain faster ROI.

Whole-System Visibility

Discover unknown-unknowns - from the kernel through userspace into high-level code, across multi-cloud workloads.

Polyglot Visibility

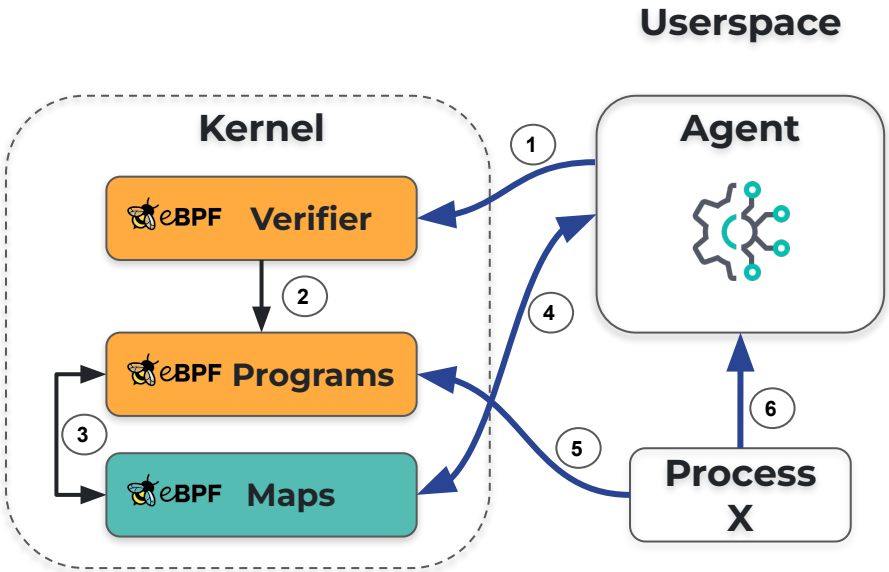
C/C++, Rust & Go (without debug symbols on host). Multiple High Level Language runtimes.

Extremely Low Overhead

Continuous profiling in production with negligible overhead.

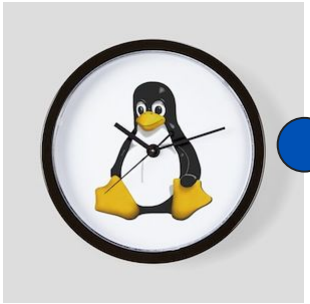
Aiming for: < 1% system CPU, ~250MB of RAM

Profiling Agent Architecture



1. eBPF unwinder programs are sent to the kernel.
2. Kernel verifies that eBPF program is safe, attaches program to event trigger.
3. The eBPF programs pass data to agent via maps.
4. The agent reads collected data from maps. The agent also writes process-specific data to maps to help eBPF unwinder programs perform unwinding.
5. eBPF unwinder programs read target process memory space during unwinding.
6. The agent reads target process memory space during HLL symbolization.

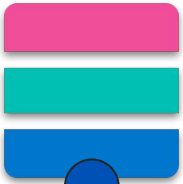
Sampling Profiling with eBPF



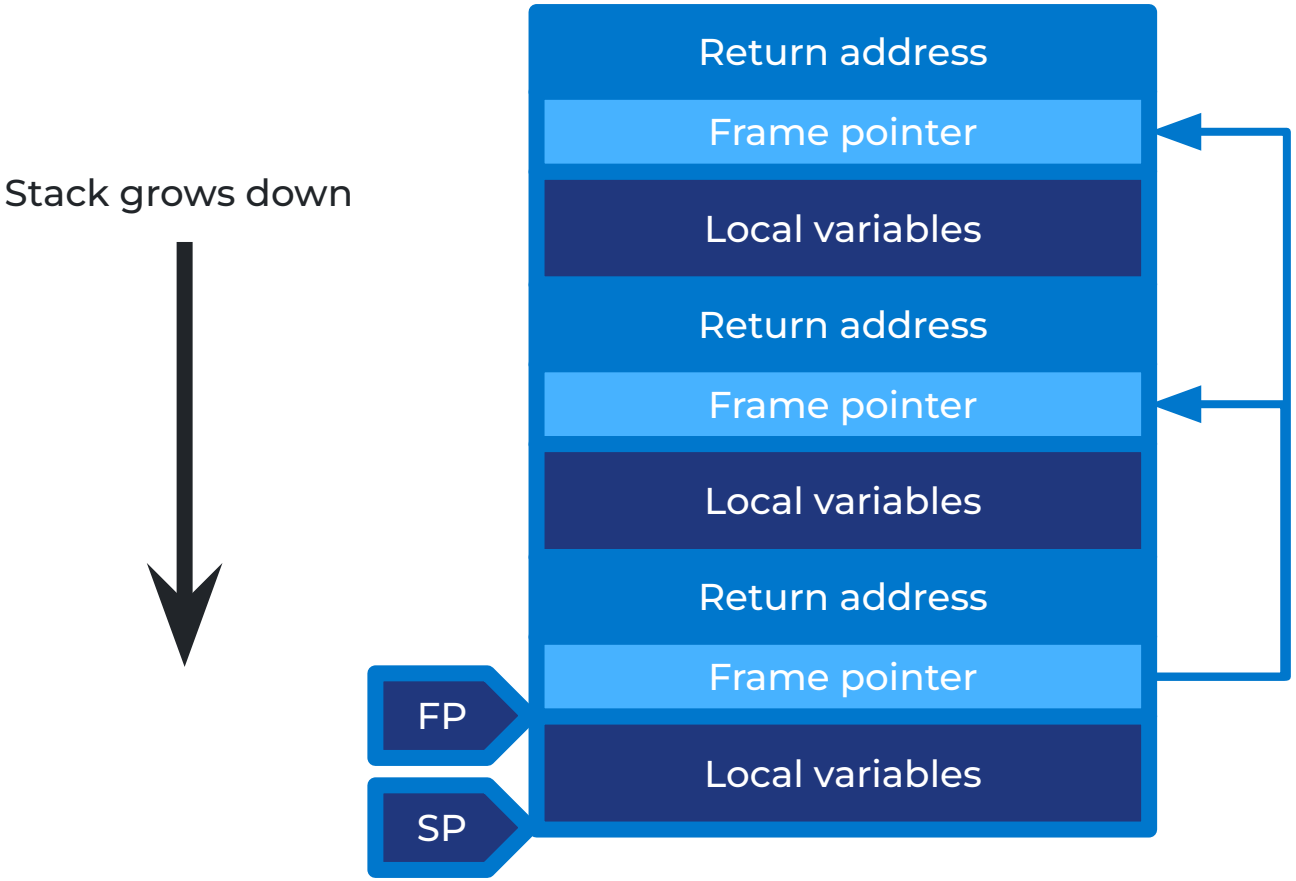
1. Attach custom code to the kernel timer interrupt to be run 20 times per second
2. Unwind native stack until HLL execution or thread entry point is detected
3. Stop unwinding (thread entry point) or unwind HLL stack
4. Send unwound stack to userspace for further processing:

   HLL symbolization, report via OTLP

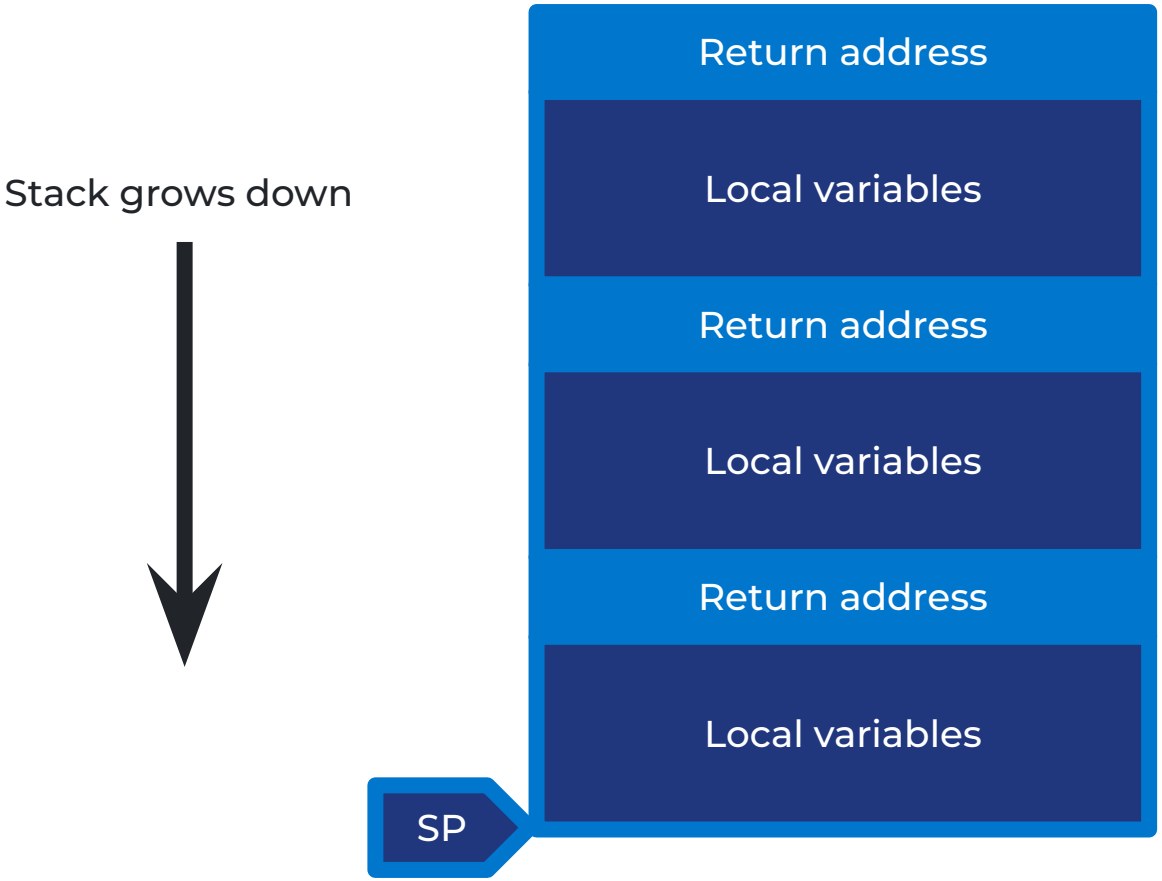
Backend



Unwinding with Frame Pointer



Frame Pointer Omission



Unwinding with FPO

.eh_frame

Contains data for every PC that helps find the next frame, in DWARF format.

DWARF ...

DWARF is terribly complicated and cumbersome to process / parse.

eBPF maps

Convert DWARF to compact form and load into eBPF maps.

Profit

Unwind FPO'ed binaries out of the box.

HLL stacks are unwound with interpreter-specific unwinding logic.

Unwinding - Native

Starting from an arbitrary execution point, recover the complete function call stack.
For native traces, given any program counter value:

```
repeat:
    moduleID, offset = getModuleOffset(pc)
    push_frame(moduleID, offset)
    pc = getNextPC(pc)
```

Output:

```
moduleID1+0x6c8
moduleID1+0x9c9
moduleID2+0xc8714
moduleID2+0x281e1
```

Unwinding begins here

Unwinding - HLL (Java, Python, ..)

- Every language needs to keep track of a call stack
- Reverse engineer how each HLL runtime does it
- Prepare eBPF maps with struct offsets and other information to enable unwinding
- eBPF code can follow the same data structures that runtime uses
- New major versions of HLL runtimes typically change DS internals
- **Supporting new versions can be a matter of only updating a few offsets**
- **Major** changes to the internals may require deeper investigations, **more time**

Why go through this effort? (many HLLs support **jitdump/perfmap**)

- Formats that allow runtimes to report unwinding instructions for JIT regions
- Supported by Linux **perf**

Unwinding - HLL - jitdump/perfmap

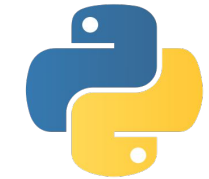
- Not enabled by default, require restarting the target process
- Function names only, no file names or line numbers
- No support for inline functions
- Costly overhead for continuous profiling

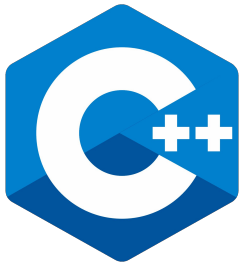
Export perf maps and jit dumps

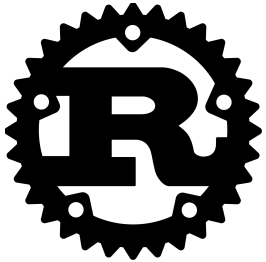
- Enables or disables selective enablement of perf maps or jit dumps. These files allow third party tools, such as the Linux `perf` tool, to identify call sites for dynamically generated code and precompiled ReadyToRun (R2R) modules.
- If you omit this setting, writing perf map and jit dump files are both disabled. This is equivalent to setting the value to `0`.
- When perf maps are disabled, not all managed callsites will be properly resolved.
- Depending on the Linux kernel version, both formats are supported by the `perf` tool.
- Enabling perf maps or jit dumps causes a 10-20% overhead. To minimize performance impact, it's recommended to selectively enable either perf maps or jit dumps, but not both.

.NET core documentation: https://learn.microsoft.com/en-us/dotnet/core/runtime-config/debugging-profiling

Supported Languages







Java OpenJDK

.NET

NodeV8 and DotNet are not supported on ARM64

ARM64

x86-64

Perl

Supported Languages

- Minimum supported kernel version is either **4.19** for x86_64 or **5.5** for ARM64 machines.
- Supporting a new version of a runtime can be as simple as bumping version check or tweaking offsets

```
$ wc -l *_tracer.ebpf.c | sort -rn
  2814 total
   929 hotspot_tracer.ebpf.c
   428 perl_tracer.ebpf.c
   330 v8_tracer.ebpf.c
   320 python_tracer.ebpf.c
   291 dotnet_tracer.ebpf.c
   275 ruby_tracer.ebpf.c
   241 php_tracer.ebpf.c
```

- JDK 7 - 23
- Python 3.6 - 3.13
- NodeV8 8.1.0+ (x86_64 only)
- PHP 7.3 - 8.3
- Perl 5.28 - 5.40.x
- Ruby 2.5 - 3.2
- DotNet (Linux/x86_64 only) 6 - 8

Unwinding native executables (C, C++, Go, ..) depends on **.eh_frame** or **.gopclntab**

Testing - Coredump Suite

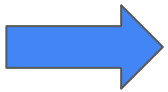
- Need **regression** tests for unwinders (eBPF C)
- A **coredump** is a snapshot of a process in execution (includes mapped memory, thread states)
- Implemented eBPF helpers in user-space
- Compile and run unwinder eBPF code in user-space, reading memory from **coredump** files
- Test cases are **JSON** files
- Can run a test case in **GDB** for debugging

```
{
  "coredump-ref": "462c77cc…",
  "threads": [
    {
      "lwp": 14,
      "frames": [
        "libpthread-2.28.so+0x11471",
        "perl+0x173cd4",
        "perl+0x1766f9",
        "perl+0x175496",
        "perl+0x17601c",
        "perl+0x14b7cd",
        "Perl+0xe602b",
        "HelloWorld::print+0 in hi.pl:12",
        "<top-level>+0 in hi.pl:19",
        "perl+0xe2c25",
        "perl+0x5e116",
```

Symbolization

Replace addresses (offsets) with symbol names (also source filenames, line numbers when available):

```
moduleID1+0x6c8
moduleID1+0x9c9
moduleID2+0xc8714
moduleID2+0x281e1
```



```
vmlinux: __x64_sys_nanosleep
vmlinux: do_syscall_64
pf-host-agent: runtime.usleep
pf-host-agent: runtime.runqgrab
```

- Native stack traces may be symbolized on-target (if symbols are available) or post-collection (e.g. at the backend)
- Non-native language stack traces (JVM, Python, DotNet, V8, ..) are **always** symbolized on-target by the agent

Symbolization - Native (C, C++, Go, ..)

Symbols for native executables are not typically present in production. Allow for symbolization to take place after profiling data has been collected.

Module IDs: Unique identifiers for executable / shared library files

```
moduleID1+0x6c8
moduleID1+0x9c9
moduleID2+0xc8714
moduleID2+0x281e1
```

- GNU Build ID: Not always present
- Go Build ID: Specific to Golang
- **Custom Build ID**: defined in specification

Algorithm for **process.executable.build_id.htlhash**

```
Input   ← Concat(File[:4096], File[-4096:], BigEndianUInt64(Len(File)))
Digest  ← SHA256(Input)
BuildID ← Digest[:16]
```

Symbolization - HLL (Java, Python, ..)

Symbols for high level language runtimes are present in target process memory space. The agent can perform symbolization on-target.

Instead of **module IDs+offsets**, HLL unwinders produce **frame IDs.**

`moduleID1+0x6c8`	**`frameID1`**
`moduleID1+0x9c9`	**`frameID1`**
`moduleID2+0xc8714`	**`frameID2`**
`moduleID2+0x281e1`	**`frameID2`**

Frame IDs are comprised of information (typically a set of pointers) that allows the agent userspace process to symbolize the frame by lookups in the target process memory space.

Can be **cached**, lifetime is not fixed, infrequent **races** (e.g. short-lived processes).

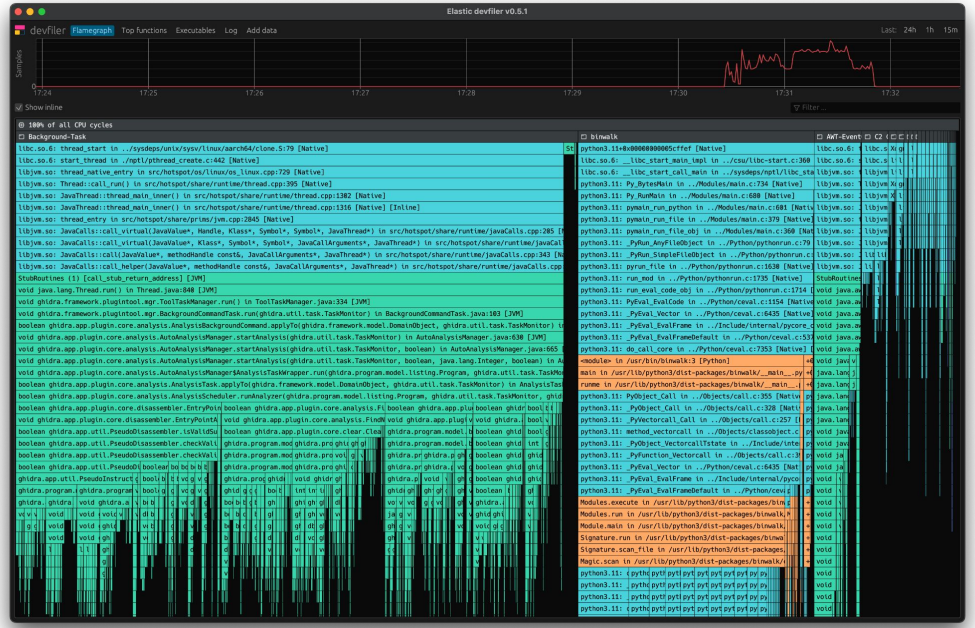
Protocol

- OTLP Profiling signal
- Defined as part of Profiling SIG (OTEPS 212, 239)
- Introduced as a new signal type in v1.3.0 (April 2024)
- Used Google pprof profile.proto as a starting point
- Breaking changes, no wire compatibility with pprof
- Striving for convertibility

- Stateless, unidirectional, gRPC
- Support multiple producers and use-cases
- Continuous profiling, instrumentation, SDKs
- Emphasis on performance
- Off-CPU, On-CPU, high-frequency sampling, aggregated samples and discrete events
- Interoperability with other OTel signals
- Still experimental!

Backends

- Profiling agent can talk to any service that supports OTLP Profiling signal: Elastic, Polar Signals, Pyroscope, Datadog (coming soon)
- Elastic devfiler: a standalone self-contained backend as a desktop application
- Goal is to have profiling agent run as an OTel collector receiver
- Further integration with OTel collector



Roadmap

M0

LuaJIT

Support for OpenResty 1.15+ AMD64/ARM64

M1

Off-CPU Profiling

WIP, sampling-based.

M2

Native Symbols

WIP, discussed in Profiling SIG:

- Extract symbols in Agent
- Symbol upload protocol

M3

OTLP Profiling

Focus on stabilization:

- Pprof cleanups
- Semconv
- Specification
- Events

M4

OTel Collector

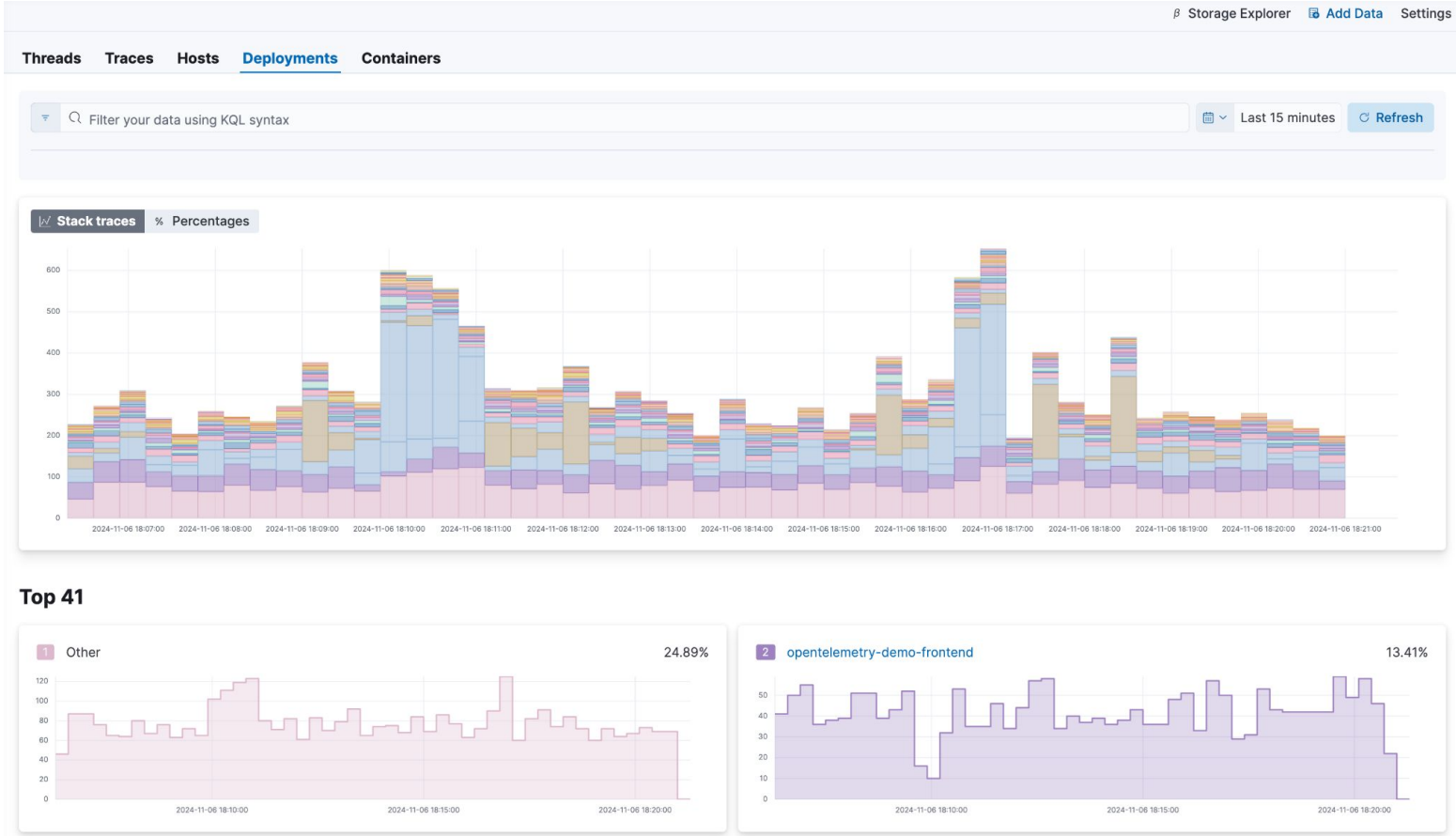
Integration work:

- Agent receiver
- Processors
- Profiling distribution

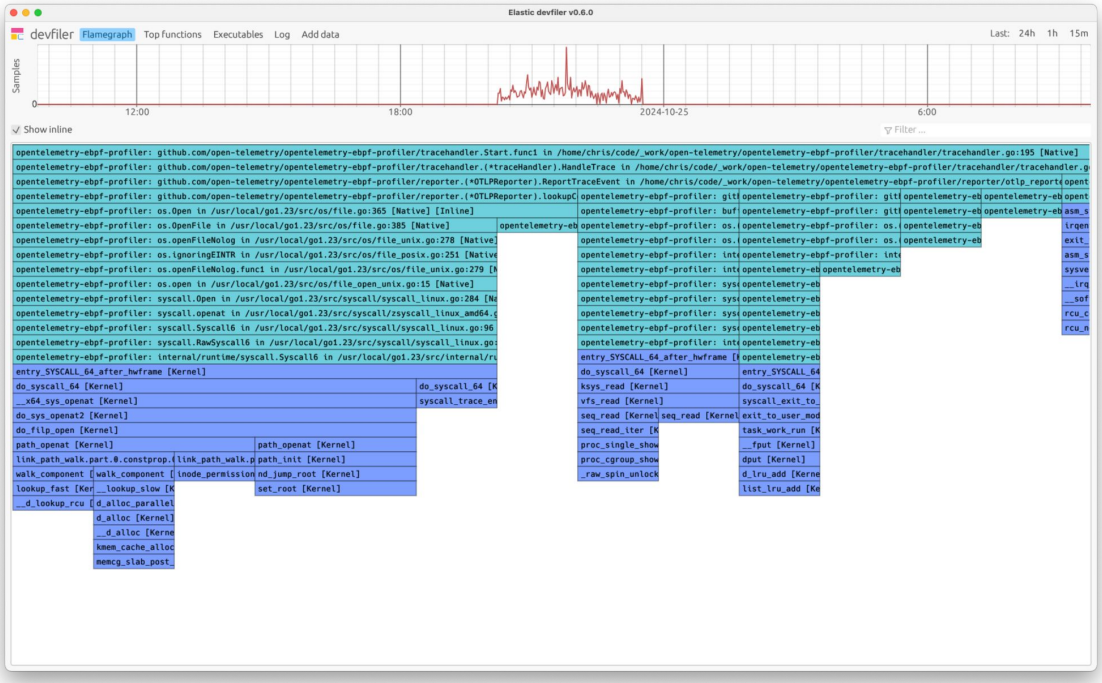
Future

- Memory profiling
- Erlang, other runtimes?
- High-frequency sampling
- Trigger unwinding from any kernel event
- Your idea here!

Demo


Storage Explorer
Add Data
Settings
Threads
Traces
Hosts
Deployments
Containers
Filter your data using KQL syntax
Last 15 minutes
Refresh
Stack traces
Percentages
600
500
400
300
200
100
0
2024-11-06 18:07:00
2024-11-06 18:08:00
2024-11-06 18:09:00
2024-11-06 18:10:00
2024-11-06 18:11:00
2024-11-06 18:12:00
2024-11-06 18:13:00
2024-11-06 18:14:00
2024-11-06 18:15:00
2024-11-06 18:16:00
2024-11-06 18:17:00
2024-11-06 18:18:00
2024-11-06 18:19:00
2024-11-06 18:20:00
2024-11-06 18:21:00
Top 41
1
Other
24.89%
120
100
80
60
40
20
0
2024-11-06 18:10:00
2024-11-06 18:15:00
2024-11-06 18:20:00
2
opentelemetry-demo-frontend
13.41%
50
40
30
20
10
0
2024-11-06 18:10:00
2024-11-06 18:15:00
2024-11-06 18:20:00

Live Demo - Devfiler



1. [Devfiler](#) (Use auth token: **c74dfc4db2212015**)
2. [opentelemetry-ebpf-profiler](#) (Commit: **38669a76**)

Questions

Thank you!


[opentelemetry-ebpf-profiler]


[opentelemetry-proto]

optimyze