KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

From Bytes to Traces: A Deep Dive into OpenTelemetry eBPF Instrumentation Internals

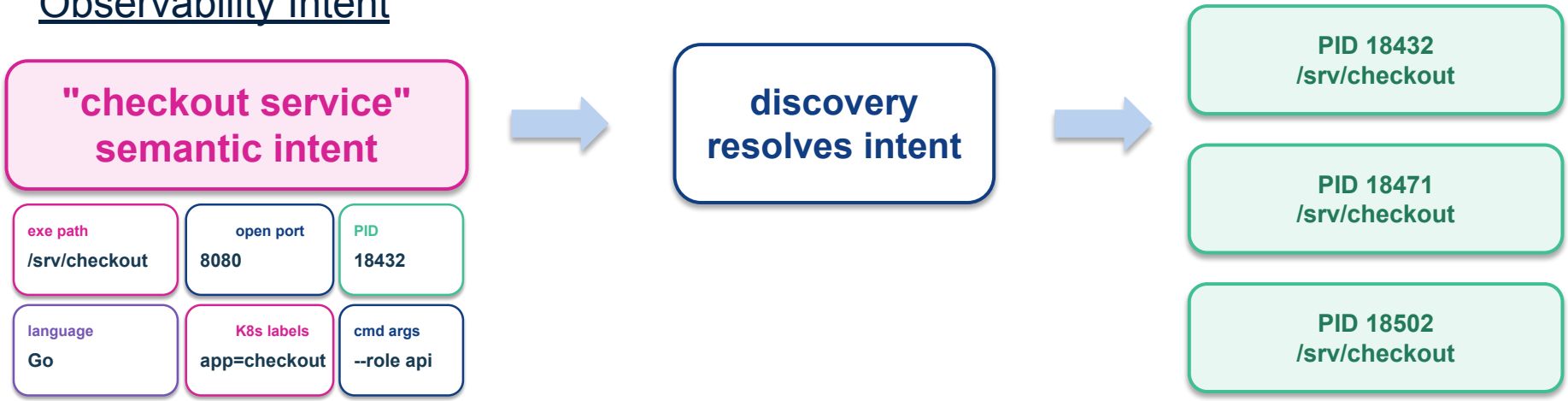
Mike Dame | Rafael Roquetto
OpenTelemetry eBPF Instrumentation maintainers



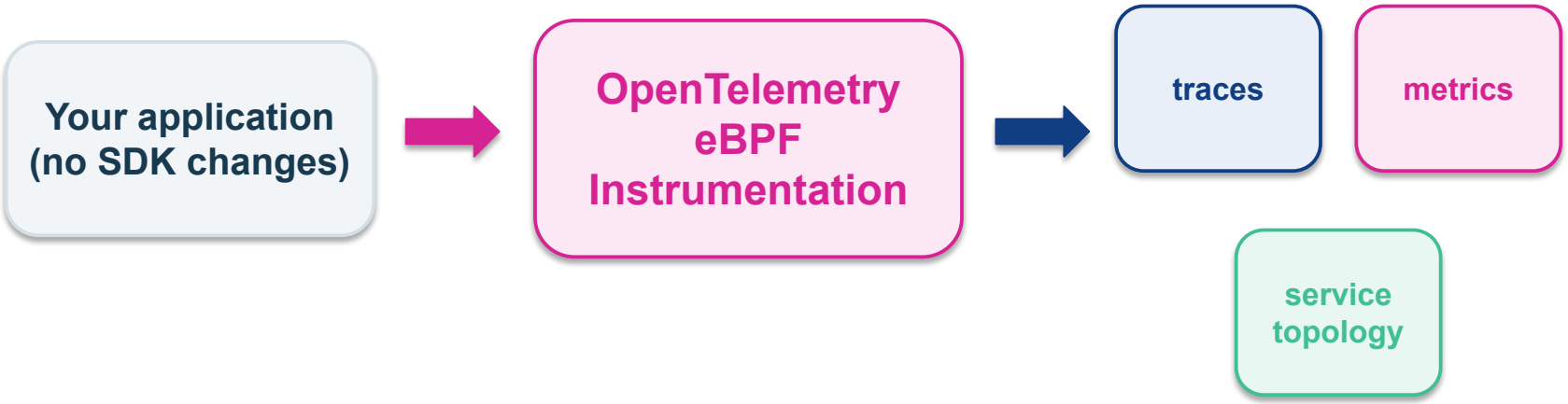
What is OBI?



Observability Intent



What is OBI?



Automatic instrumentation: observe application/runtime/kernel events, then model them into OpenTelemetry signals.

We configure a service - but OBI instruments running processes



A service is an abstraction. At runtime, the thing OBI can attach to and admit events from is a process lifecycle - PID + executable/runtime metadata.

Configuring OBI to discover a service

Selectors express intent using facts OBI can resolve on the host:

exe path
/srv/checkout

open port
8080

PID
18432

language
Go

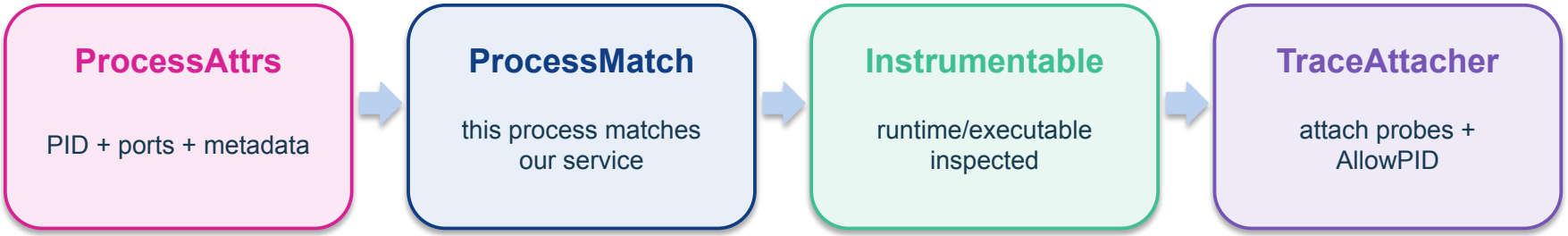
K8s labels
app=checkout

cmd args
--role api

Different selectors. Same destination: a set of matching processes that OBI can instrument.

pkg/appolly/discover/matcher.go

Discovery progressively turns facts into an Instrumentable



This is the control-plane half of the story: decide WHAT is allowed to generate telemetry.

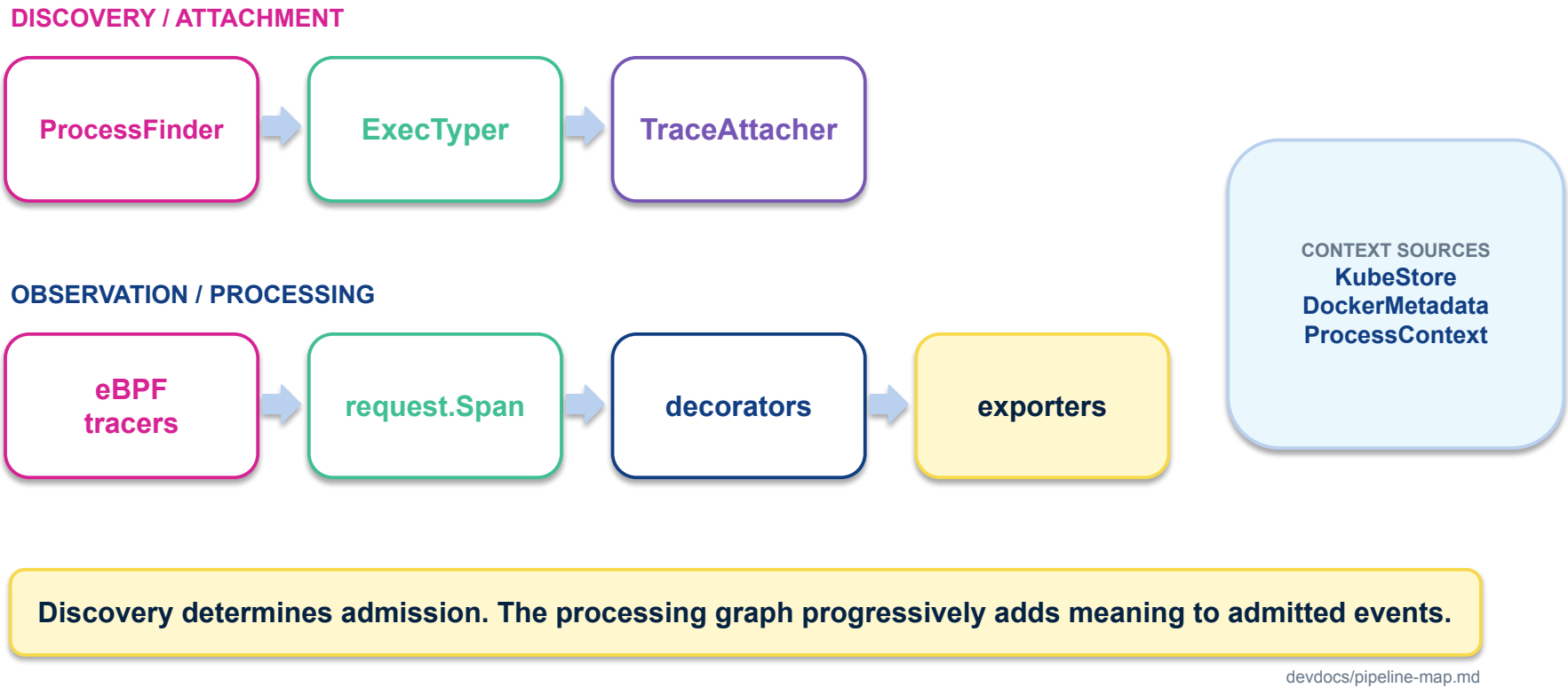
KubeCon CloudNativeCon
North America 2026

OBI bootstrap

Read configuration, build the graphs, start discovery and tracing.
Then events can flow.



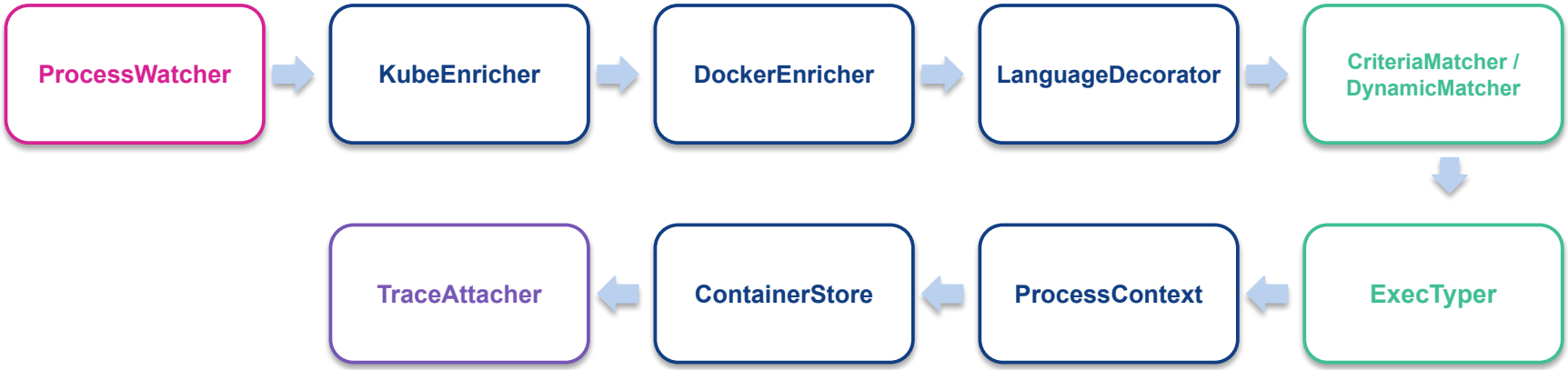
Bootstrap builds two connected graphs



Discovery determines admission. The processing graph progressively adds meaning to admitted events.

devdocs/pipeline-map.md

The current ProcessFinder pipeline

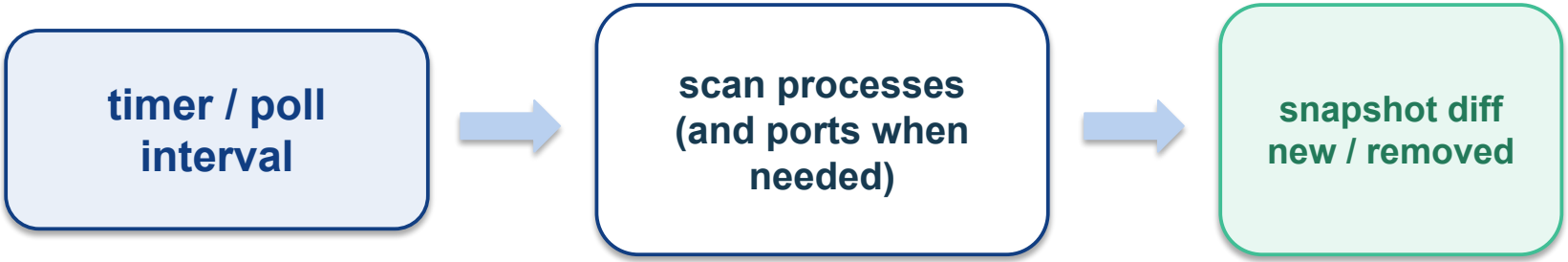


The important shape is linear: discover → enrich → match → inspect executable → attach.

Static CriteriaMatcher and DynamicMatcher converge on the same downstream path.

pkg/appolly/discover/finder.go

What actually causes a new process to be considered?



ProcessWatcher is fundamentally a polling loop. A newly observed process is emitted when a snapshot says “this PID was not in the previous state.”

Default interval in current code: 5s when discovery.poll_interval is zero.

pkg/appolly/discover/watcher_proc.go

What bind() contributes: a port-state invalidation signal

bpf/watcher/watcher.c

```
SEC("kprobe/sys_bind")
int obi_kprobe_sys_bind(...) {
    u16 port =
get_sockaddr_port_user(addr);
    ...
    trace->payload = port;
    bpf_ringbuf_submit(trace, 0);
}
```



NewPort 8080



set fetchPorts = true

The watcher does NOT learn the PID from this event. It refreshes process↔port ownership on a poll and then matches normally.

ProcessAttrs is the Candidate Record

watcher_proc.go

```
type ProcessAttrs struct {
    pid             app.PID
    openPorts       []uint32
    cmdArgs         string
    processAge      time.Duration
    metadata        map[string]string
    podLabels       map[string]string
    podAnnotations  map[string]string
    detectedType    svc.InstrumentableType
}
```

ProcessWatcher

Kube/ DockerEnricher

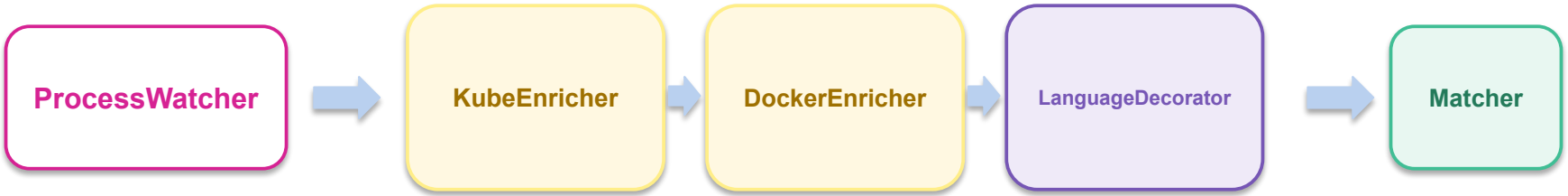
LanguageDecorator

Why enrich before selection?

Because selectors can refer to Kubernetes/container/runtime facts that are not visible in a raw /proc row.

There is a second enrichment stage later for request.Span.
Same word “decorator”, different object.

It still has not decided: “this is our service.”



watcher_proc.go + finder.go

ProcessAttrs is the watcher's candidate record

watcher_proc.go

```
type ProcessAttrs struct {
    pid             app.PID
    openPorts       []uint32
    cmdArgs         string
    processAge      time.Duration
    metadata        map[string]string
    podLabels       map[string]string
    podAnnotations  map[string]string
    detectedType    svc.InstrumentableType
}
```

ProcessWatcher

Kube/ DockerEnricher

LanguageDecorator

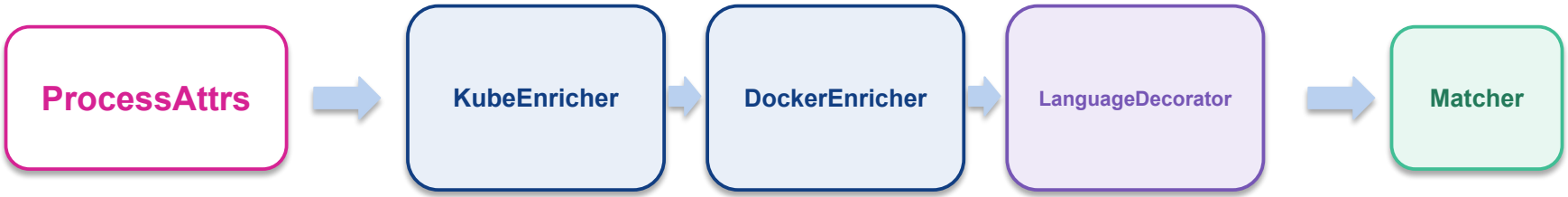
At this point OBI knows facts:

- which PID exists
- which ports it owns
- container/K8s metadata
- language hints
- command line / age

It still has not decided: "this is our service."

pkg/appolly/discover/watcher_proc.go

Before matching, process events can be enriched

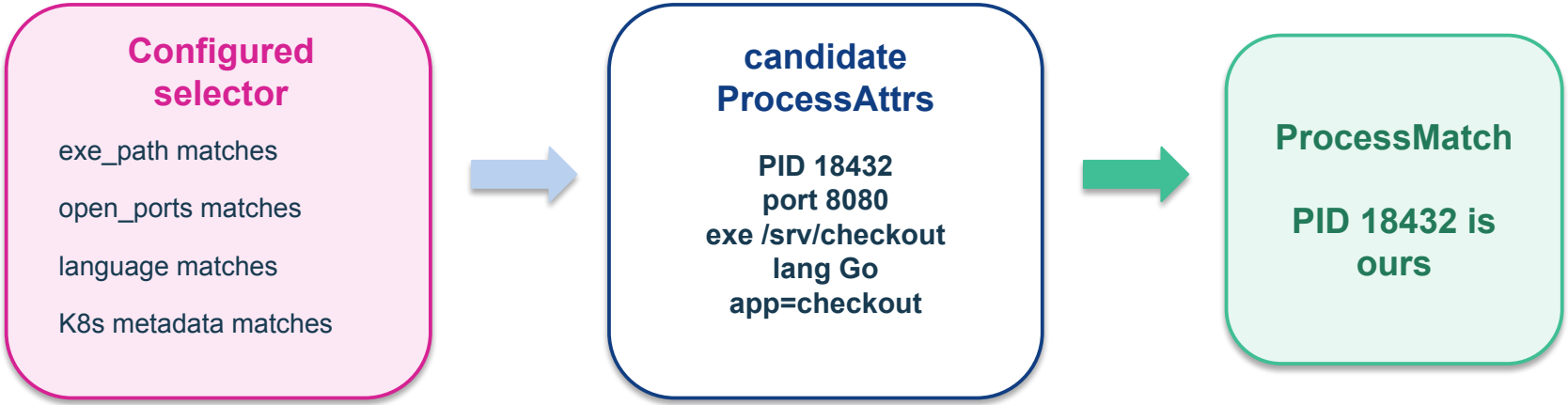


Why enrich before selection? Because selectors can refer to Kubernetes/container/runtime facts that are not visible in a raw /proc row.

There is a second enrichment stage later for request.Span. Same word “decorator”, different object.

pkg/appolly/discover/finder.go

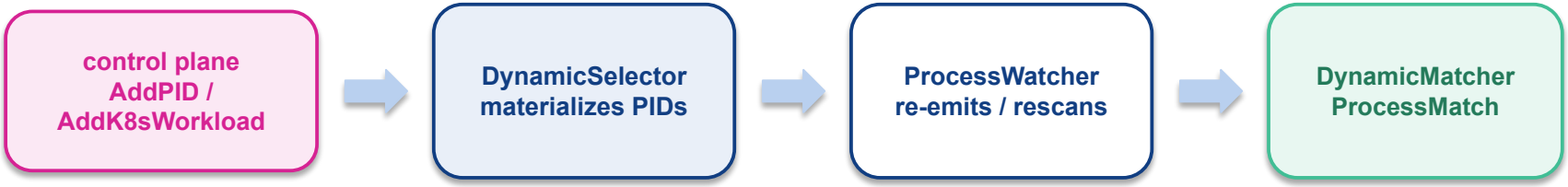
CriteriaMatcher: service intent becomes a concrete process set



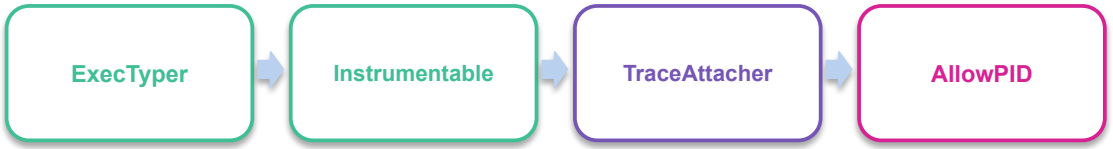
Static selection ends here: the semantic “service” is now represented by one or more matching PIDs.

pkg/appolly/discover/matcher.go

Dynamic selection changes WHO is selected - not the downstream machinery

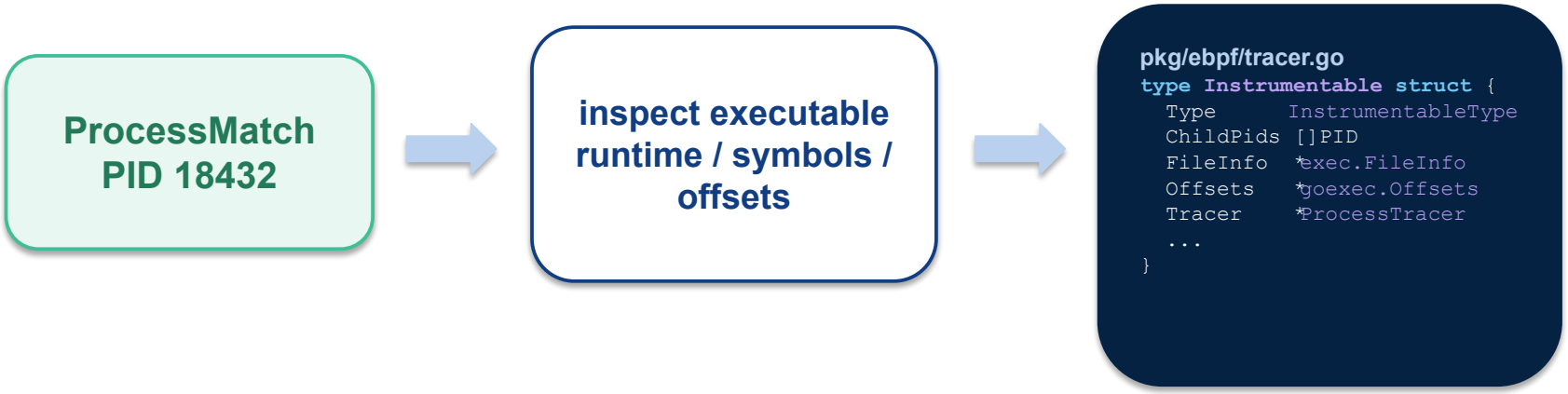


Then it rejoins the exact same path:



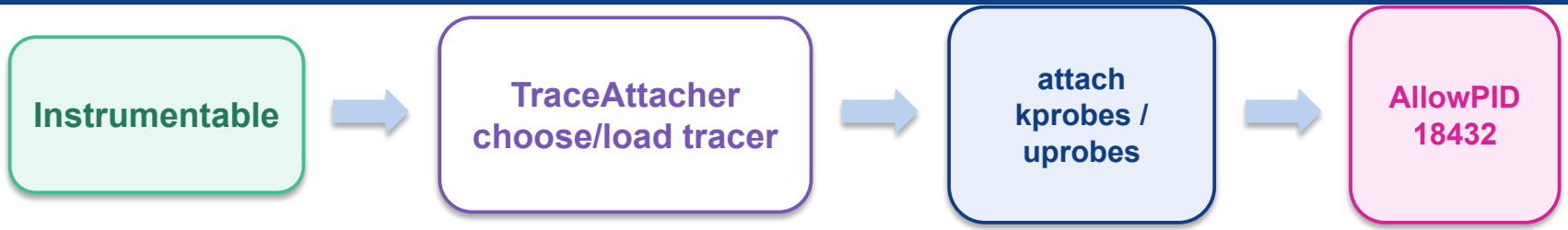
dynamic_selector.go + matcher_dynamic.go

ExecTyper turns a matched process into an Instrumentable



This is where “a selected process” becomes “something for which OBI knows how to instrument the executable/runtime.”

TraceAttacher makes the selection live



attacher.go

```
tracer.AllowPID(ie.FileInfo.Pid(),
                ie.FileInfo.Ns(),
                ie.FileInfo)
```

AllowPID is the bridge: only telemetry attributed to admitted process lifecycles is allowed through.

pkg/appolly/discover/attacher.go

Which tracer? OBI has several instrumentation strategies

kernel/network + TLS/runtime hooks

runtime-specific Go probes

tpinjector / log / GPU etc.

For the rest of the main story, choose one path:

Generic HTTP/1 server request

One protocol is enough to expose the mechanics: bytes → protocol state → ring buffer → request.Span.

pkg/appolly/discover/finder.go

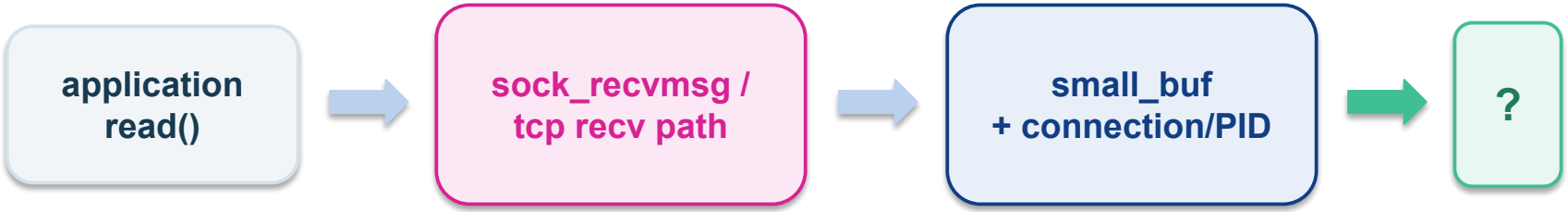


From bytes to traces

GENERICTRACER

Main actors:
1) http_info_t - kernel-side HTTP transaction state
2) request.Span - userspace semantic record

1. A recv path exposes a bounded view of application bytes



observed bytes

```
47 45 54 20 2f 70 61 79 20 48 54 54 50 2f 31 2e 31 ...
 G  E  T     /  p  a  y     H  T  T  P  /  1  .  1
```

2. The protocol dispatcher chooses a parser/state machine

bpf/generictracer/protocol_handler.c

```
if (args->protocols.http &&
    is_http(args->small_buf, MIN_HTTP_SIZE,
            &args->packet_type)) {
    preempt_guarded_tail_call(
        ctx, &jump_table,
k_tail_protocol_http);
}
```

Other branches

- HTTP/2 + gRPC
- MySQL / PostgreSQL / MSSQL
- Kafka / SunRPC / Aerospike
- generic TCP

Protocol detection does not create an OTel span. It chooses which kernel-side state machine now owns the bytes.

3. is_http() is intentionally small and cheap

bpf/generictracer/protocol_http.h

```
static __always_inline u8 is_http(
    const unsigned char *p, u32 len, u8 *packet_type) {
    if (len < MIN_HTTP_SIZE) return 0;

    if (p[0]=='H' && p[1]=='T' && p[2]=='T' && p[3]=='P' &&
        p[4]=='/' && p[5]=='1' && p[6]=='.') {
        *packet_type = PACKET_TYPE_RESPONSE;
        return 1;
    } else if (is_http_request_buf(p)) {
        *packet_type = PACKET_TYPE_REQUEST;
        return 1;
    }
    return 0;
}
```

What this buys us

- request vs response
- bounded verifier-friendly classification
- a handoff into HTTP-specific state

4. HTTP detection materializes http_info_t - the first core actor

bpf/common/http_info.h

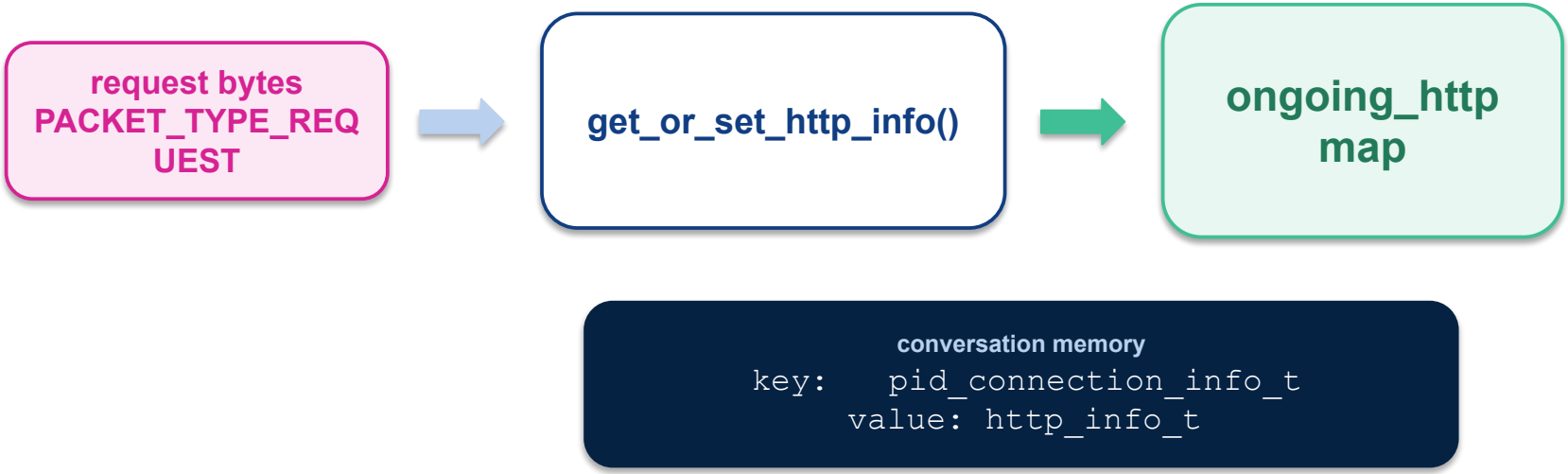
```
typedef struct http_info {
    u8 flags, type, ssl, delayed;
    connection_info_t conn_info;
    u64 start_monotime_ns;
    u64 end_monotime_ns;
    u64 req_monotime_ns;
    tp_info_t tp;
    pid_info pid;
    u32 len, resp_len;
    u16 status;
    unsigned char buf[256];
    ...
} http_info_t;
```

What it represents

identity	PID + connection
timing	start / request / end
HTTP	status + lengths + buffer
trace context	trace / span / parent IDs
lifecycle	submitted / delayed / response state

bpf/common/http_info.h

5. The request context persists in BPF map state



The struct is not “one packet.” It accumulates a transaction across multiple observations on the same process+connection.

bpf/generictracer/protocol_http.h

6. The request side initializes semantic state

protocol_http.h

```
process_http_request(info, ...) {
    info->pid = ...;
    info->type = HTTP_SERVER / HTTP_CLIENT;
    fixup_connection_info(...);
    info->start_monotime_ns = bpf_ktime_get_ns();
    info->req_monotime_ns = ...;
    info->status = 0;
    info->len = len;
    ...
}
```

http_info_t now knows

- who: PID
- which connection
- server or client
- when it started
- request byte count

bpf/generictracer/protocol_http.h

Trace identity is established here too

protocol_http.h (simplified)

```
urand_bytes(tp_p->tp.span_id,  SPAN_ID_SIZE_BYTES);

found_tp =  find_trace_for_server_request(...,
                                          &tp_p->tp,
...);

if (!found_tp) {
    new_trace_id(&tp_p->tp);
    memset(tp_p->tp.parent_id, 0, ...);
}

// when enabled, header parsing can adopt an incoming
traceparent
```

tp_info_t establishes

- fresh SpanID
- look for existing context
- reuse / adopt TraceID
- ParentSpanID from carrier
- or new TraceID + no parent

bpf/generictracer/protocol_http.h

7. The response completes the same transaction context



simplified

```
info->end_monotime_ns = bpf_ktime_get_ns();
info->status = parsed_status;
info->resp_len += sent_len;
```

Now the kernel has enough state to describe one HTTP request/response transaction.

bpf/generictracer/protocol_http.h

8. eBPF flushes http_info_t through the ring buffer

protocol_http.h

```
submit_http_event(info, pid_conn) {
    if (!http_info_emittable(info) || info->submitted)
        return;

    http_info_t *trace =
        bpf_ringbuf_reserve(&events, sizeof(*trace), 0);
    __builtin_memcpy(trace, info, sizeof(*trace));
    trace->flags = k_event_type_k_http_request ;
    bpf_ringbuf_submit(trace, get_flags());
}
```

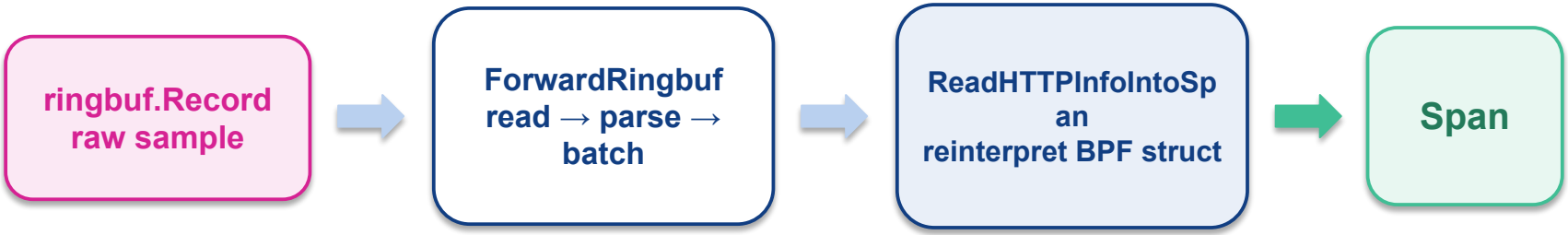


BPF ring buffer

http_info_t

bpf/generictracer/protocol_http.h

9. Userspace reads the record and chooses the matching parser



pkg/ebpf/common/http_transform.go

```
event, err := ReinterpretCast[BPFHTTPInfo](record.RawSample)
...
return HTTPInfoEventToSpan(parseCtx, event)
```

ringbuf.go + http_transform.go

10. http_info_t becomes request.Span

http_transform.go (trimmed)

```
span := request.Span{
    Type:          reqType,
    Method:        req.Method,
    Path:          removeQuery(req.URL.String()),
    Peer:          peer,
    Host:          host,
    HostPort:      int(event.ConnInfo.D_port),
    Start:         int64(event.StartMonotimeNs),
    End:           int64(event.EndMonotimeNs),
    Status:        resp.StatusCode,
    TraceID:       event.Tp.TraceId,
    SpanID:        event.Tp.SpanId,
    ParentSpanID:  event.Tp.ParentId,
    Pid:           ...,
}
```

The semantic jump

raw C layout	→ Go values
bytes	→ method / path
conn tuple	→ peer / host
timestamps	→ span duration
tp_info	→ trace IDs

pkg/ebpf/common/http_transform.go

request.Span is the second core actor - OBI's userspace currency

pkg/appolly/app/request/span.go

```
type Span struct {
    Type, SubType      ...
    Method, Path       string
    Peer, Host         string
    PeerPort, HostPort int
    Status             int
    RequestStart       int64
    Start, End         int64

    Service            svc.Attrs
    TraceID             trace.TraceID
    SpanID              trace.SpanID
    ParentSpanID        trace.SpanID
    Pid                 PidInfo
    ...
}
```

Notice the layers already inside one object

- protocol semantics
- network endpoints
- timing + status
- trace identity
- service metadata slot
- process identity

pkg/appolly/app/request/span.go

11. PID closes the loop back to discovery metadata

Earlier: discovery

PID 18432 → FileInfo service attrs / runtime / metadata

→

PIDsFilter.Filter namespace + user PID → generation

→

Now: request.Span

span.Service = FileInfo.ServiceAttrs( )

pids.go

```
inputSpans[i].Service =
gen.fileInfo.UnsafeServiceAttrs()
pf.normalizeTraceContext(&inputSpans[i])
```

This is where “bytes from PID 18432” becomes “the checkout service.”

pkg/ebpf/common/pids.go

12. Now request.Span enters the userspace processing pipeline

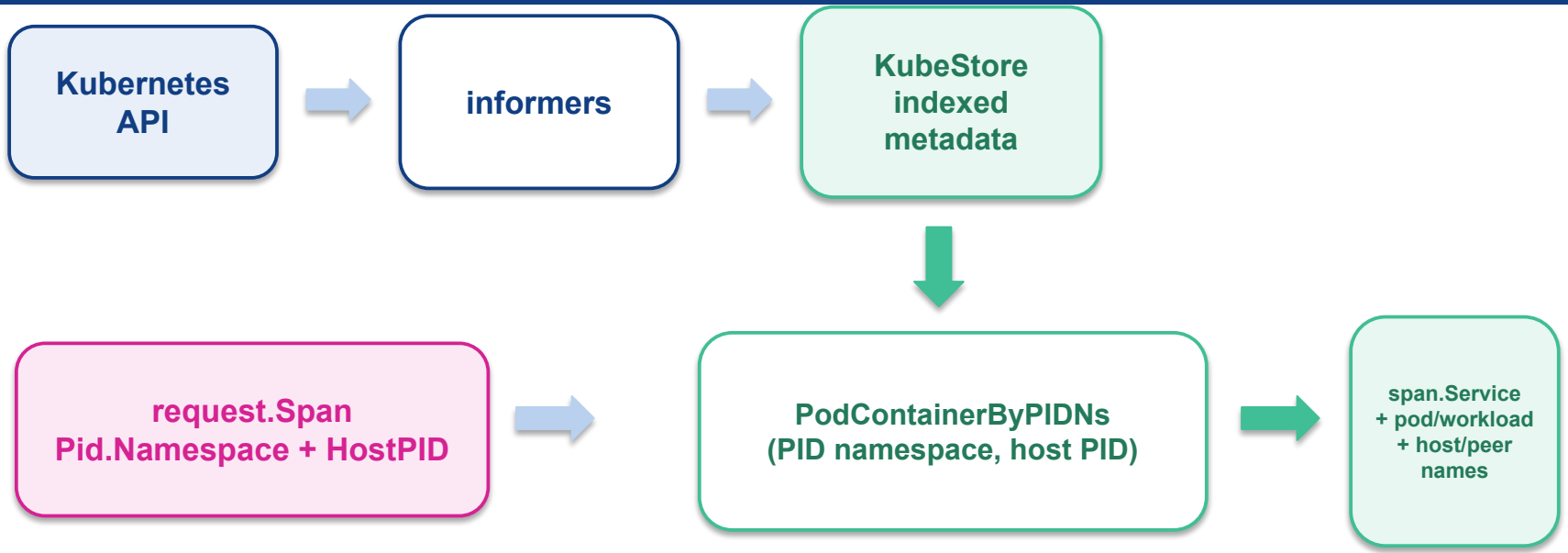


Same object, progressively more context:

- host instance ID
- route / span naming
- Kubernetes resource metadata
- container metadata
- peer/host names
- attribute gating

pkg/appolly/instrumenter.go

13. Kubernetes decoration is a userspace join



No Kubernetes semantics are required in the HTTP BPF state machine. They are joined later, where richer caches and APIs are available.

pkg/transform/k8s.go

Other decorators fill different semantic gaps

ReadDecorator	host name + service instance ID
Routes	route / low-cardinality span name
DockerDecorator	container metadata outside K8s
NameResolution	peer / host names

The kernel captures facts cheaply; userspace turns those facts into the semantic conventions we want to export.

pkg/appolly/instrumenter.go

14. Export is the final translation from OBI's Span to OpenTelemetry

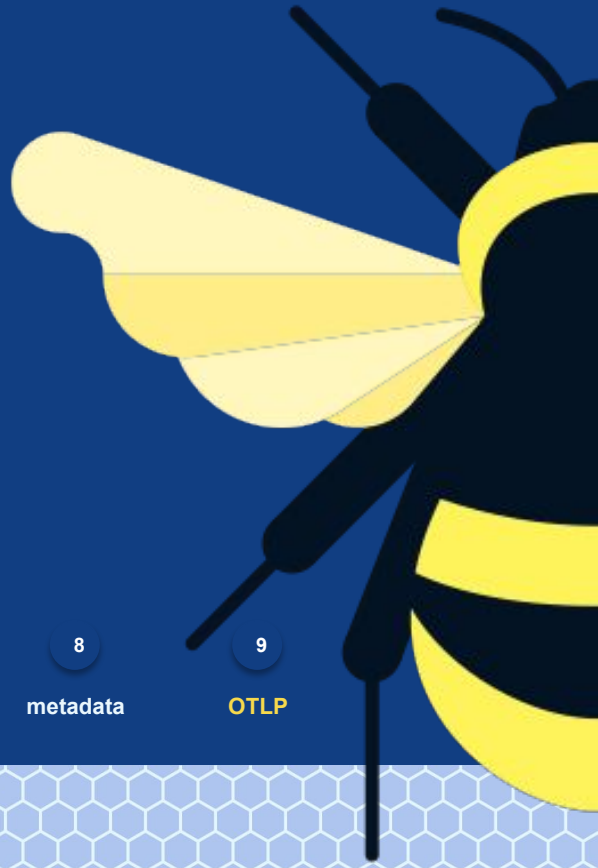


The same normalized span stream also feeds:

- RED / app metrics
- span metrics
- service graph metrics
- Prometheus

That is why traces are the useful linear story: metrics/topology can be explained as downstream aggregations of the same semantic records.

pkg/appolly/instrumenter.go + tracesgen



That completes “Bytes → Traces”

service selector → process → Instrumentable → probes → bytes → http_info_t → ring buffer → request.Span → enrichment → OTLP

1	2	3	4	5	6	7	8	9
service	PID	probes	bytes	http_info_t	ringbuf	request.Span	metadata	OTLP

The complete mental model

1	**Select**	service intent → processes/PIDs
2	**Instrument**	process → Instrumentable → attached tracers
3	**Materialize**	bytes → protocol state + trace IDs → http_info_t
4	**Translate**	ring buffer → request.Span
5	**Enrich**	PID/service/K8s/container/route metadata
6	**Export**	OTLP traces + span-derived metrics/topology

Thank you!

KubeCon CloudNativeCon North America 2026

CONTRIBUTE

Help build OBI

Report issues, propose improvements,
or contribute code to the project.

github.com/open-telemetry/opentelemetry-ebpf-instrumentation



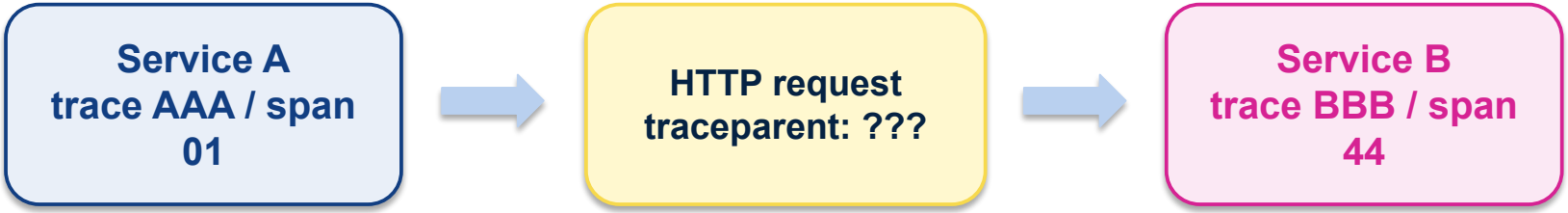
Scan to contribute



Bonus chapter: trace context propagation

Those IDs now exist locally. How does the caller's trace context cross the network so the next service joins the same distributed trace?

A local span is not yet a distributed trace



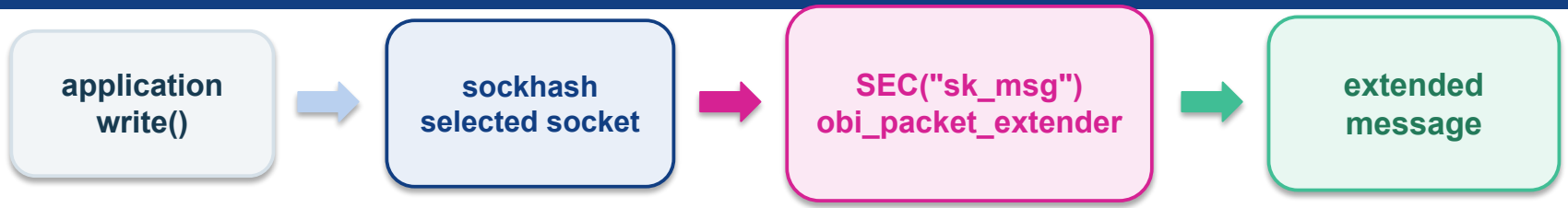
Without carrying the caller's trace ID + parent span ID, both sides can observe the request perfectly and still build unrelated traces.

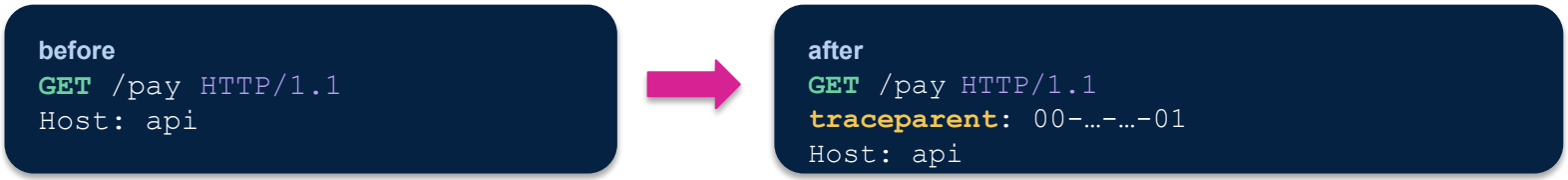
Why OBI cannot depend on bpf_probe_write_user() for propagation

1. **Kernel lockdown** — LOCKDOWN_BPF_WRITE_USER can reject it on many deployed kernels
2. **Wrong abstraction** — mutating an existing userspace buffer is not the same as inserting bytes into a socket stream
3. **Space problem** — a traceparent header needs extra room; the original buffer may not have it

So the generic HTTP propagation path moves rewriting to a hook that owns a socket message: sk_msg.

tpinjector can extend supported egress at sk_msg





At this layer OBI can make room in the socket message and insert the header before TCP transmits it.

The receiver adopts the incoming parent and continues the same trace



simplified receive path

```
decode_hex(info->tp.trace_id, t_id, ...);
decode_hex(info->tp.parent_id, s_id, ...);
existing->tp = info->tp;
```

The next request.Span therefore carries the caller's TraceID and a ParentSpanID that links the services.

bpf/generictracer/protocol_handler.c