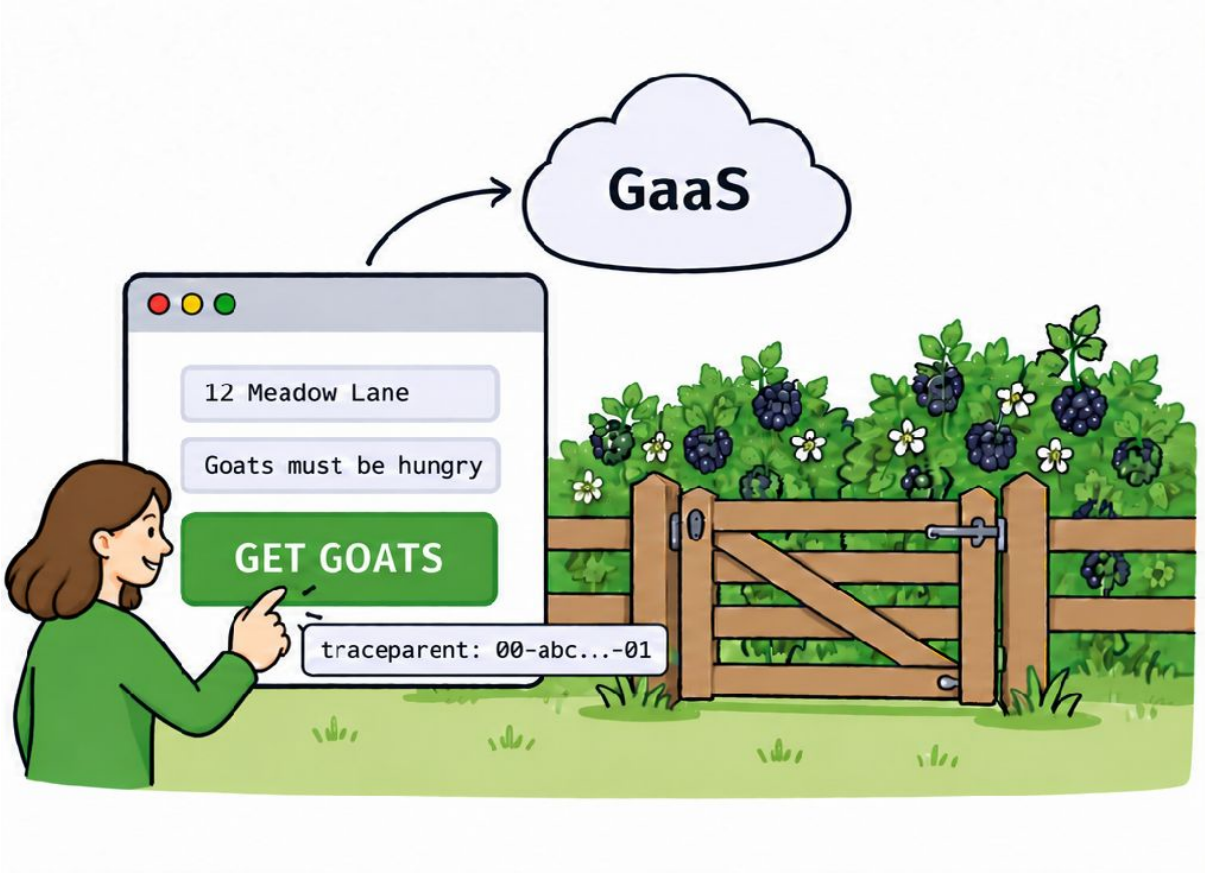
Observability Day
NORTH AMERICA

The Trace Stops Here?

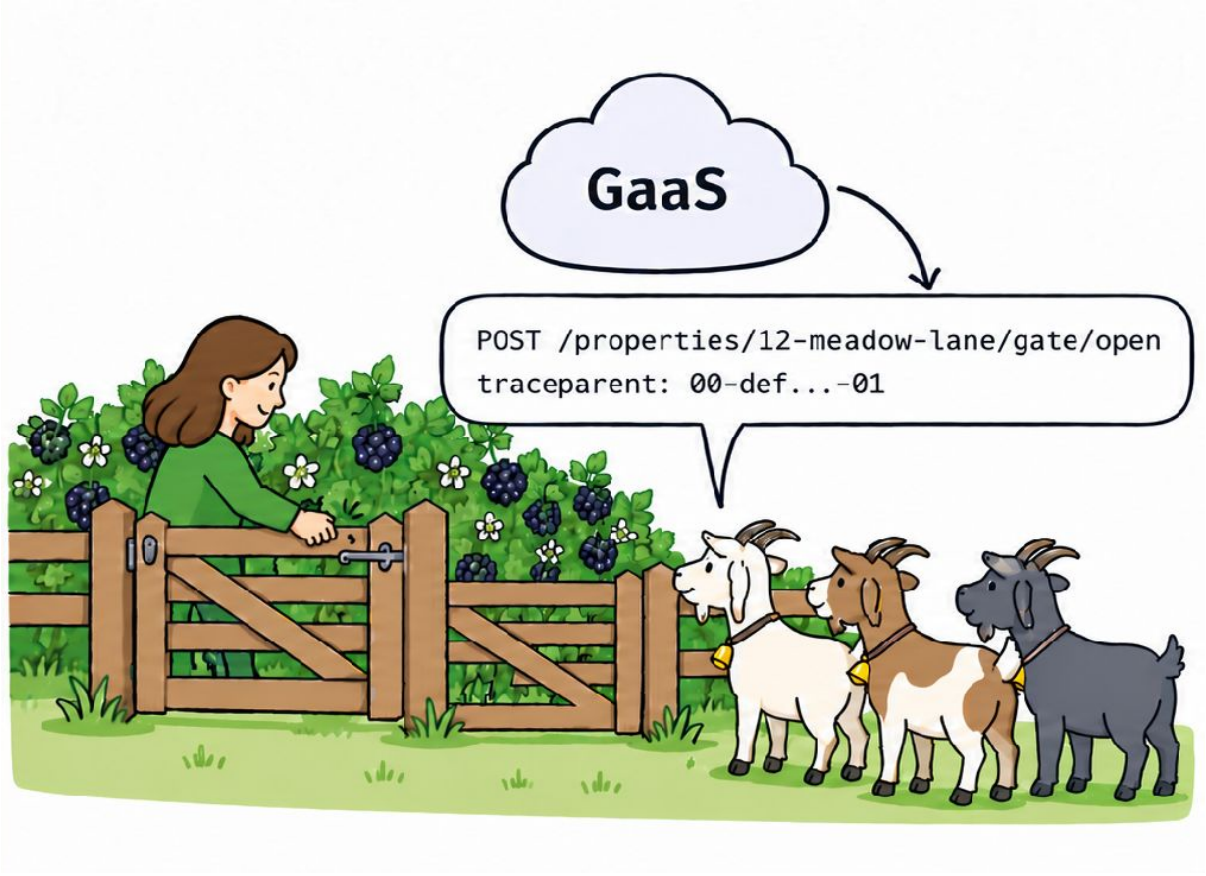
Rethinking Context Propagation for Managed Services

Liudmila Molkova
Staff Software Engineer at Google

Goats as a Service



Async goat delivery



Two dashboards, partial traces

Observability Day NORTH AMERICA

Customer

Two traces in the customer account

Trace `abc…` · Booking

Order goats

HTTP POST to GaaS

Trace `def…` · Gate callback

HTTP POST /gate/open

Open gate

GaaS

One trace in the provider account

Trace `def…` · Booking and dispatch

Frontend HTTP POST

Queue publish

queue delay

Dispatcher process

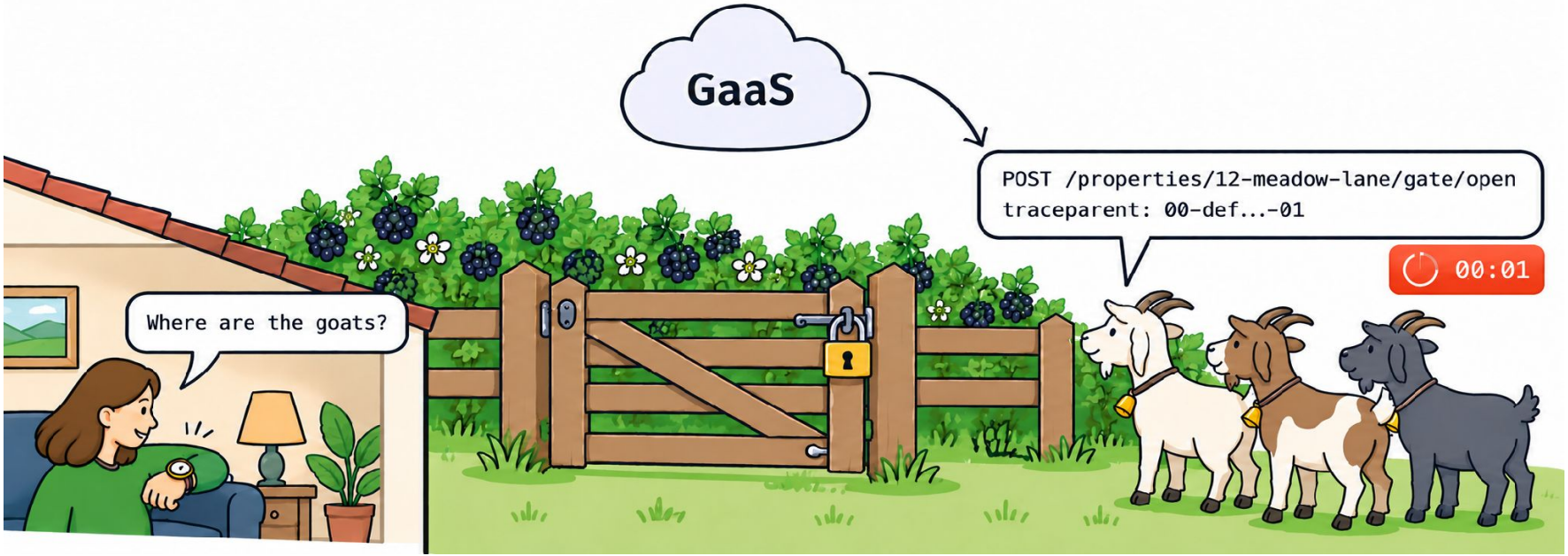
Drive goats

HTTP POST to gate

The gate SERVER span is stored in the customer account.

Where are the goats?

Observability Day
NORTH AMERICA



The callback never arrived

Observability Day
NORTH AMERICA

Customer

One trace · booking accepted

Trace `abc…` · Booking · 2 s

Order goats

HTTP POST to GaaS

No incoming gate request

No callback trace to investigate

GaaS

One trace · delivery failed

Trace `def…` · 3 attempts · 55 s

Frontend HTTP POST

Dispatcher process

Drive goats

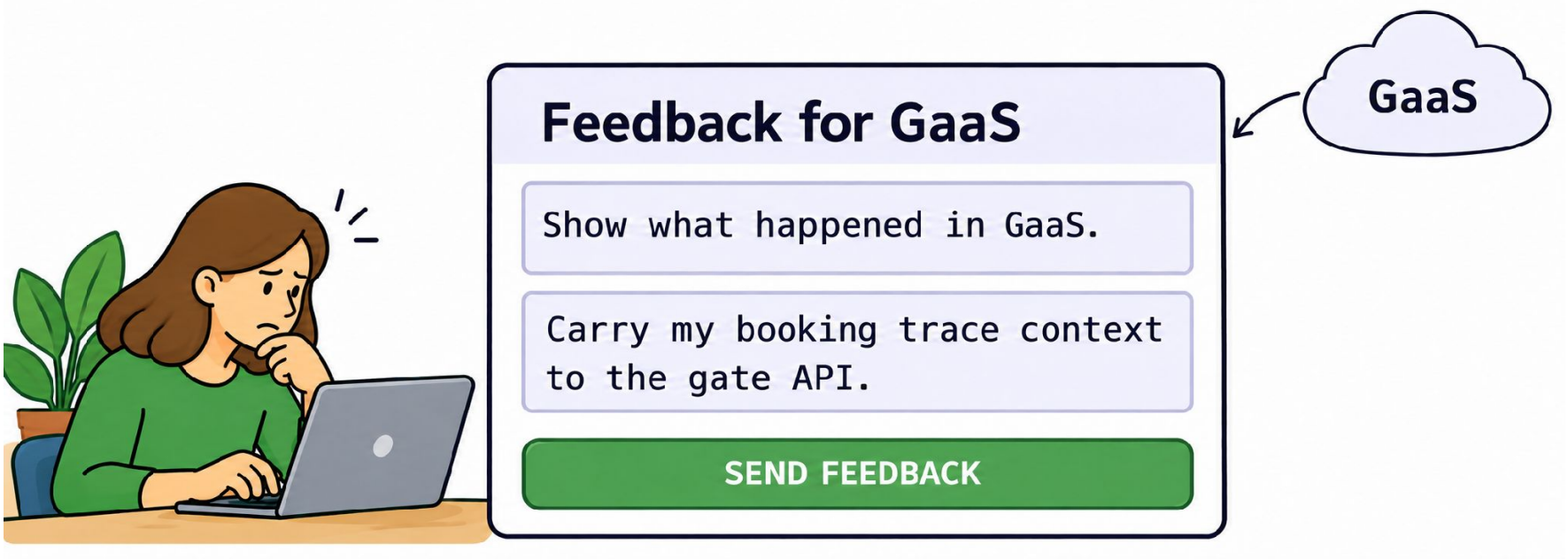
Gate POST · attempt 1 timeout

Gate POST · attempt 2 timeout

Gate POST · attempt 3

ERROR · gate unreachable

Show me what happened



Meanwhile in GaaS

Design doc: Native observability for customers

- **HTTP server span** for each incoming request
  - Export to the customer's **OTLP endpoint**
  - Parent: incoming **W3C Trace Context**
- **HTTP client span** for each gate request attempt
- `gaas.goats_dispatched` event with **estimated arrival time**

Now I can see what happened



Customer trace

Trace abc...

Order goats

POST booking · CLIENT

GaaS booking · SERVER

Goats dispatched · ETA 5 min

Gate attempt 1 · CLIENT ×

Gate attempt 2 · CLIENT ✓

Open gate · SERVER

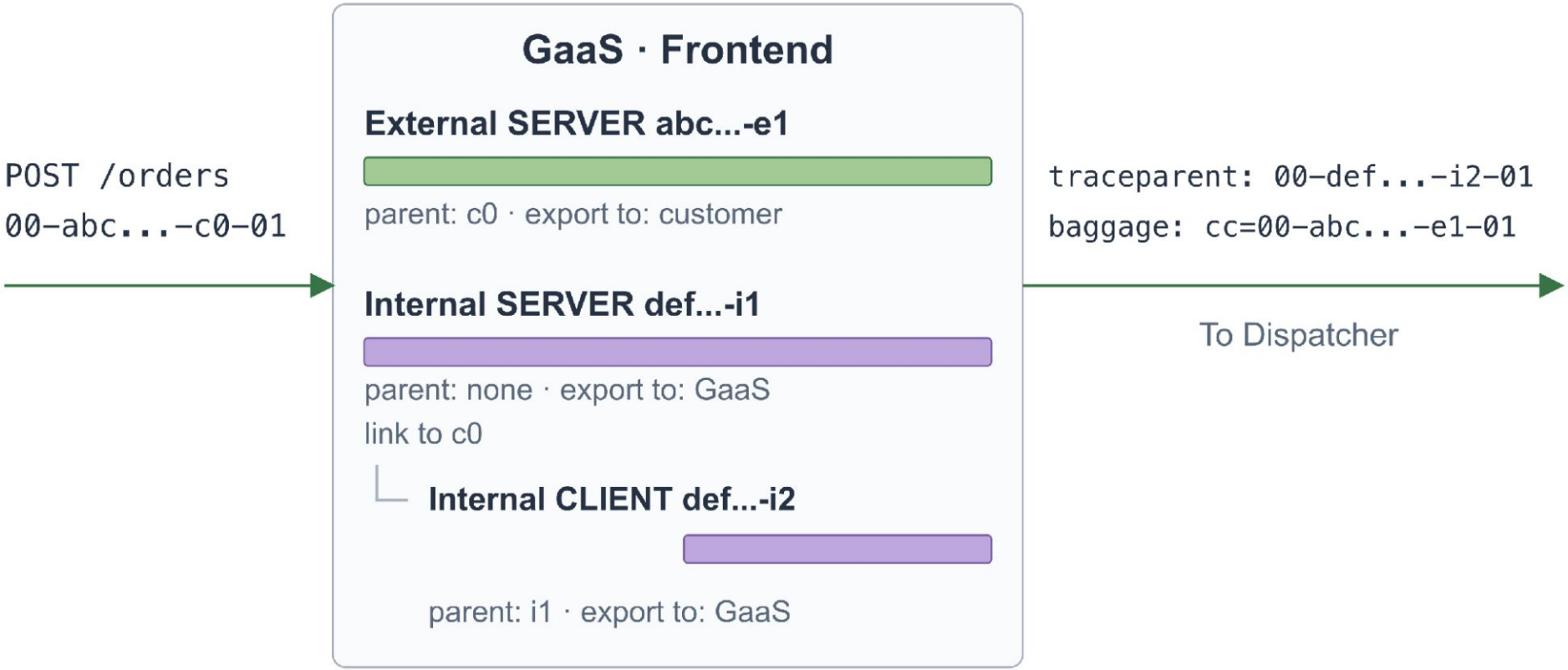
Meanwhile in GaaS

Postmortem: Customer context broke our internal tracing

- Observability **costs grew 5×** after rollout. We followed the customer's sampling flag.
- Some customers sampled out every request, leaving us **no traces** to investigate their issues.
- Some customers **hardcoded a trace ID**, merging all their requests into one trace.

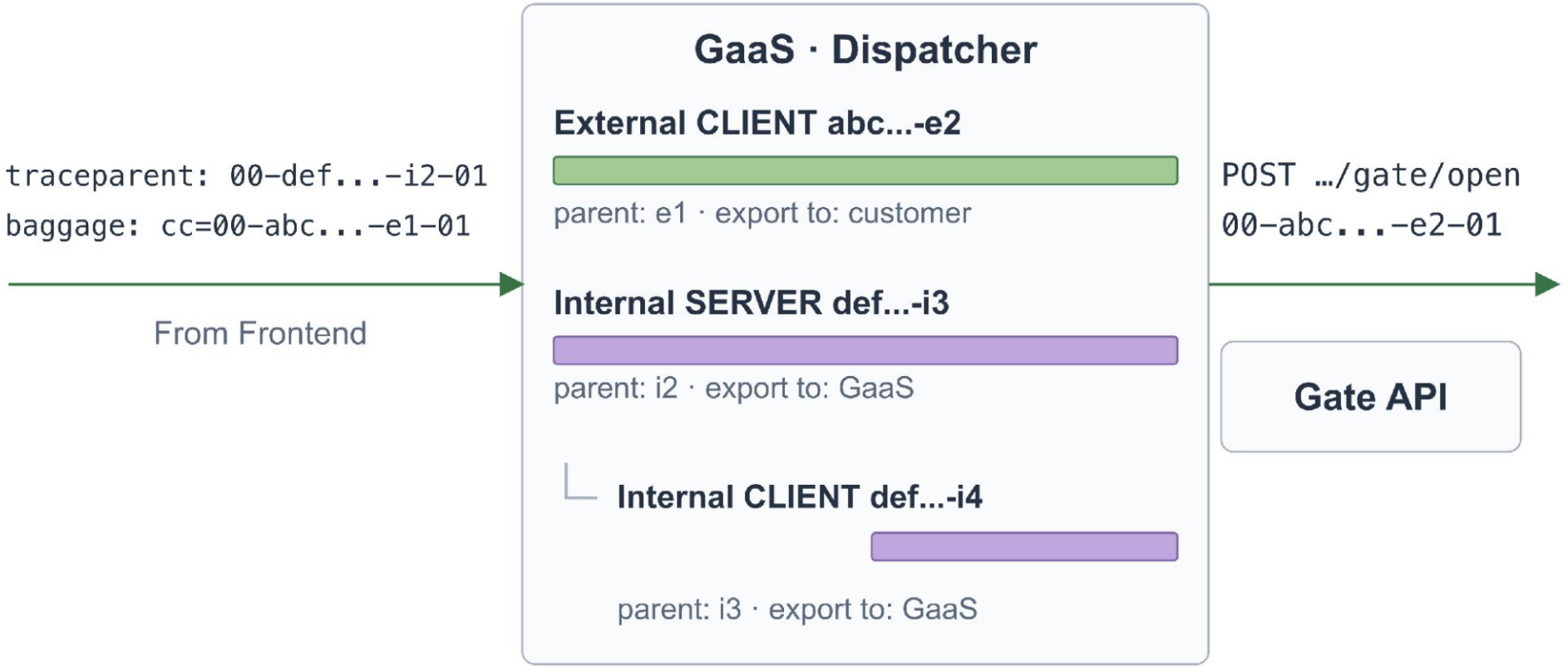
Meanwhile in GaaS

Design doc: Propagate customer context independently of internal one



Meanwhile in GaaS

Design doc: Propagate customer context independently of internal one



Two independent traces

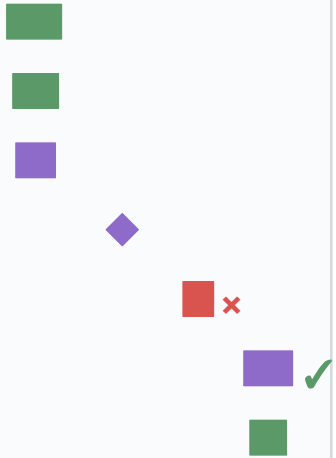
Observability Day
NORTH AMERICA

Customer

Customer sampling policy

Trace `abc…`

Order goats
POST booking · CLIENT
GaaS booking · SERVER
Dispatched · ETA 5 min
Gate attempt 1 · CLIENT
Gate attempt 2 · CLIENT
Open gate · SERVER

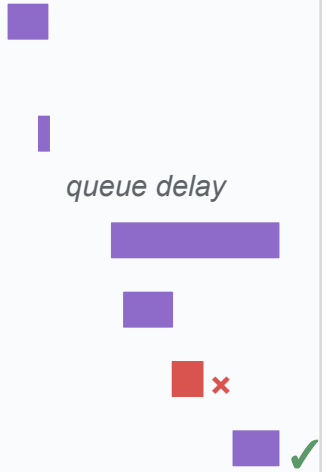


GaaS

Independent provider sampling policy

Trace `def…`

Frontend · SERVER
link → abc…
Queue publish
queue delay
Dispatcher process
Drive goats
Gate attempt 1 · CLIENT
Gate attempt 2 · CLIENT



What GaaS had to build

- **Custom boundary instrumentation:** create internal and external spans with the right parents and links.
- **Custom propagator:** carry customer context and select outgoing context.
- **Multi-tenant export** to customer OTLP endpoints**:** *out of scope.*

Incoming call: two spans

Observability Day
NORTH AMERICA

```
incoming = propagator.extract(request.headers)
external = customer_tracer.start_span(
    "POST /orders", kind=SERVER, parent=incoming,
)

with internal_tracer.start_active_span(
    "POST /orders", kind=SERVER,
    parent=None,
    links=[Link(incoming)] if incoming.is_valid else [],
):
    # Baggage carries the external SERVER context
    baggage["cc"] = serialize(external.context)
    handle(request)
```

Where should context travel?

- **Baggage: application-defined context**, independent of tracing. Propagates by default
- **Custom propagation:** store it with the order, carry it in a custom header. *More implementation work, more control.*
- **Tracestate: tracing-system metadata**, not app context. Travels with `traceparent`; limited to 256 chars.

Outgoing call: two spans


```
customer = get_customer_context(Context.current())
external = customer_tracer.start_span(
    "POST /gate/open", kind=CLIENT, parent=customer,
)

with internal_tracer.start_active_span(
    "POST /gate/open", kind=CLIENT,
) as internal:
    # The gate receives the external CLIENT context
    inject(external.context, request.headers)
    send(request)
```

Can auto-instrumentation help?

What if we could **configure auto-instrumentation** to use extracted context as a **parent**, a **link**, or **drop** it? See OTEP 5055.

```
trace_continuation:
  ingress:
    default_action: restart_with_link
    rules:
    - action: continue
      attribute_matchers:
      - http.route: /admin
```

Can auto-instrumentation help?

Observability Day
NORTH AMERICA

```
incoming = propagator.extract(request.headers)
action = config.get_ingress_action(request)

with internal_tracer.start_active_span(
    "POST /orders", kind=SERVER,
    parent=incoming if action == "continue" else Context(),
    links=[Link(incoming)]
        if action == "restart_with_link" and incoming.is_valid
        else [],
):
    handle(request)
```

HYPOTHETICAL EXAMPLE · NOT A REAL API

Which context leaves GaaS?

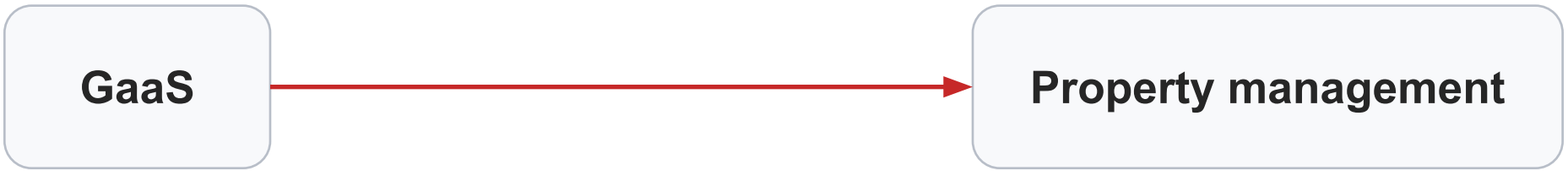
For the gate API, keep the internal context active; inject the **customer-facing CLIENT context**.

```
trace_continuation:
  egress:
    default_action: suppress
    rules:
    - action: inject
      attribute_matchers:
      - server.address: internal*
```

We leak internals to customers

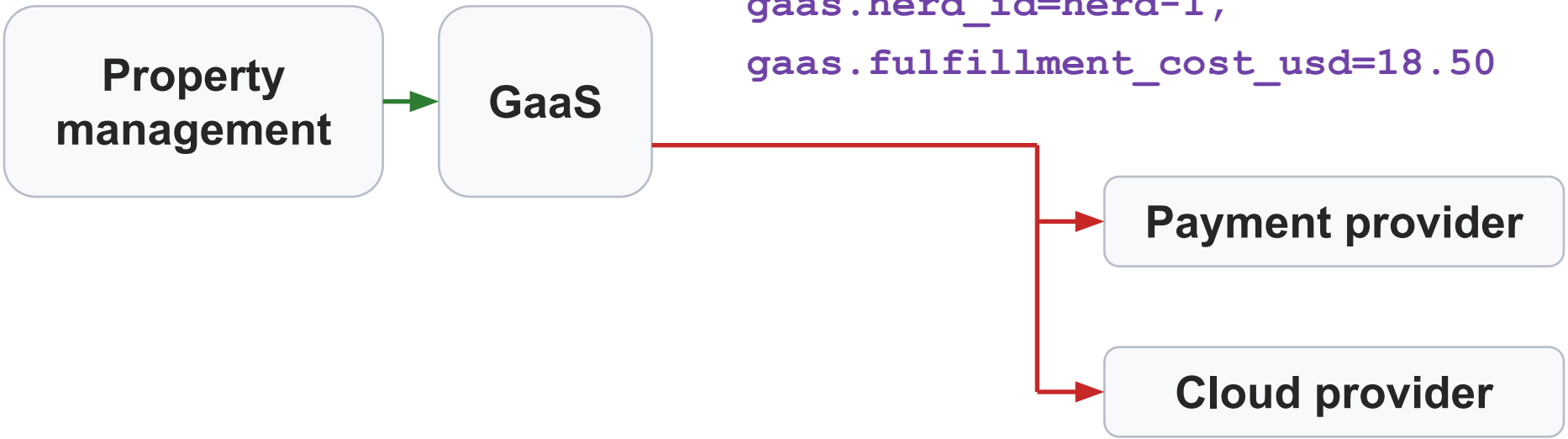
Observability Day
NORTH AMERICA

```
POST …/gate/open
baggage:
   gaas.herd_id=herd-1,
   gaas.fulfillment_cost_usd=18.50
```



We also leak customer data


```
baggage:
   customer_id=owner@example.test,
   order_id=WO-4821,
   cc=00-abc...-e1-01,
   gaas.herd_id=herd-1,
   gaas.fulfillment_cost_usd=18.50
```



Keep baggage inside GaaS

- **Incoming external calls:** allowlist, drop everything else.
- **Outgoing internal calls:** forward baggage only to internal endpoints.
- **Other destinations:** allowlist by destination.

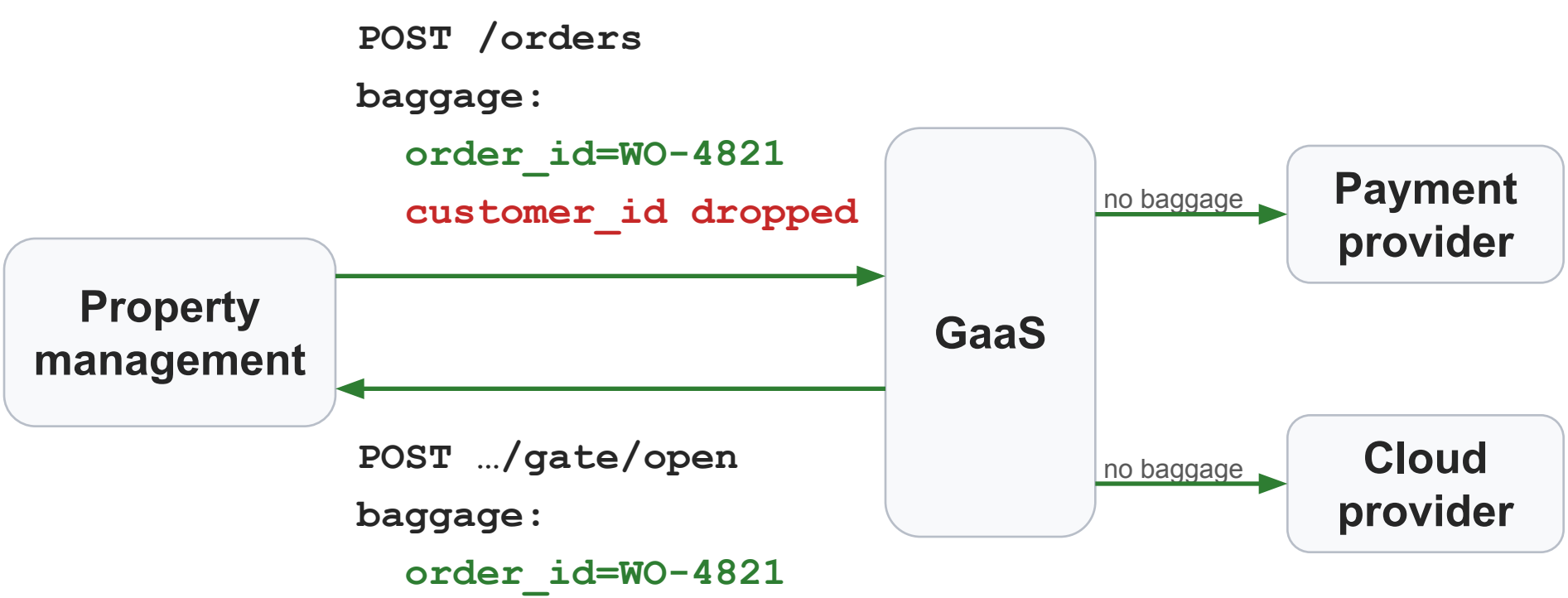
```
baggage_policy:
  ingress:
    default_strategy: drop
    rules:
    - route: /orders
      allow: [order_id]
    - route: /admin
      allow: ["*"]
  egress:
    default_strategy: drop
    internal_origins:
    - ${DISPATCHER_URL}
    rules:
    - url_pattern: "*/gate"
      allow: [order_id]
```

What about tracestate?

- Use it for **tracing metadata**, not application context.
- Keep it with the trace context it describes. **Link or drop** consistently.
- Support **removing your own entries** when calling external endpoints.

Baggage stays within bounds

Observability Day
NORTH AMERICA



We herd goats, not webhooks

Design doc: Use EventBleat from our cloud provider

Pros

- Outsource retries, **delivery guarantees**, delivery status, and endpoint management.
- **80% reduction** in worker load, saving millions.

Cons

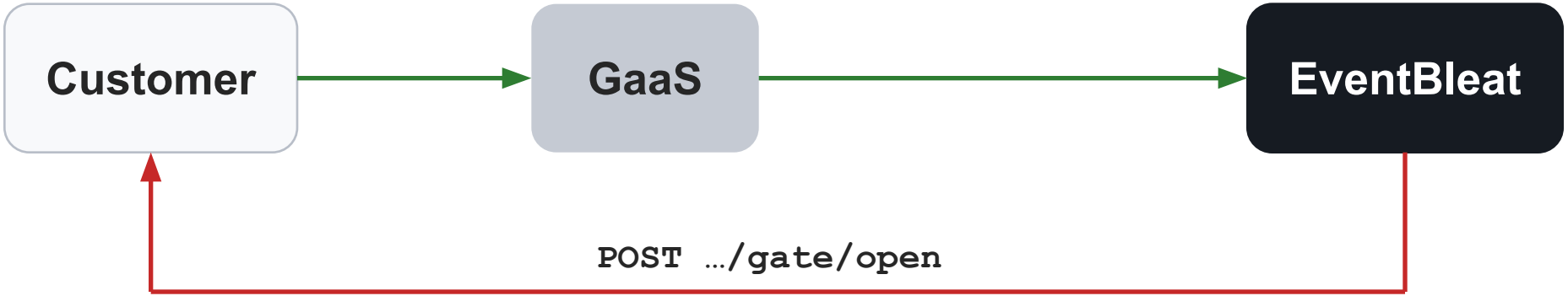
- EventBleat needs to propagate **both our context and our customer's context**.
- It also needs to provide **observability to us**.

EventBleat is a black box

Observability Day
NORTH AMERICA

```
POST /orders
traceparent: 00-abc...-c0-01
```

```
Publish notification
traceparent: 00-def...-i4-01
baggage: cc=00-abc...-e1-01
```



Goats all the way down



Recap

- W3C context propagation is **ubiquitous** now
- Propagating context is **easy**
- Now we need to make it easy to **control**

What should we do in OTel?

Make **trust boundaries configurable**:

- Where should incoming calls **start a new trace**?
- Which **external baggage** should we accept?
- Which trace context and baggage should we propagate on **outgoing calls**?

Make it **easy**, with **good defaults**, and part of **auto-instrumentation**.

Thank you!

Observability Day
NORTH AMERICA



The Trace Stops Here? Rethinking Context Propagation for Managed Services