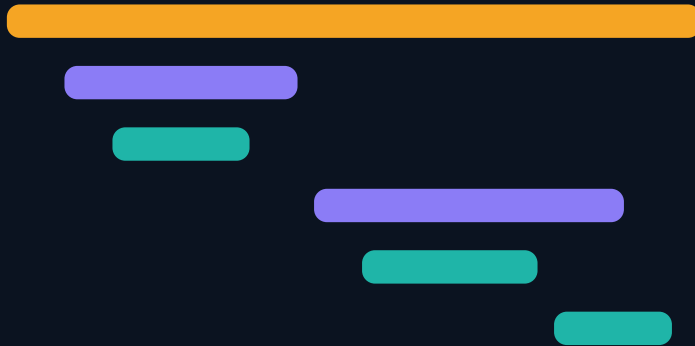


MOTACON 2026

Stop guessing, start seeing

The rise of Observability-Driven Testing

David Burns · @AutomatedTester



David Burns



Selenium

Core maintainer of the Selenium project



W3C

Browser Testing & Tools Working Group: WebDriver and WebDriver BiDi



DevRel

Developer Relations Engineer and host of the BrowserStack Talks podcast

Why I care

I've spent years helping browsers report what tests do. Now I want tests to learn from what users do.

“

**If your only signal of health is a
hard-coded assertion against mock data,
you aren't testing.**

You're hoping.

We test what we can imagine

Known-unknowns

What the lab covers

- Scripted journeys we thought of
- Static expected values
- Mocked services and seeded data
- Weeks of authoring and maintenance

Unknown-unknowns

Where incidents actually happen

- Journeys real users invent
- Emergent latency across services
- Retries, timeouts and partial failure
- Data shapes nobody seeded

Lab and production live in two worlds

LAB

Shift-left
Assertions
Mocks
Pass / fail

the gap

PRODUCTION

Traces
Metrics
Logs
Real journeys

A mountain of high-fidelity behavioural data sits in our telemetry pipelines, and our tests never read it.

Expected results are now patterns



AI agents

Non-deterministic output. The same prompt can take different tool paths, so 'equals' stops working.



Micro-frontends

One page is many teams, bundles and APIs. A single user click can fan out across a dozen services.



Distributed systems

Queues, retries, caches and third parties. Correctness depends on timing and ordering.



PART 1

Observability-Driven Testing

Production telemetry as the test oracle

What ODT is, and what it isn't

Use telemetry (traces, metrics, logs) as the oracle that decides whether the system behaved correctly: in CI, in pre-prod and in production.

IT IS

- Assert on traces, not just on the DOM
- Real-user journeys become test cases
- Invariants that apply in every environment
- Tests and incidents speak the same language

IT ISN'T

- Just 'testing in production'
- Replacing unit or contract tests
- Another dashboard to stare at
- Only for SREs

Spans, traces and the header that joins them



Traces

What happened, and in what order



Metrics

How much, and how often



Logs

Details, tied to a trace_id

W3C Trace Context: the traceparent header

```
00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
```



version



trace-id: the whole journey



parent span-id: who called me



flags: sampled

Every service that receives it continues the same trace.

Assert on the journey, not just the screen

Classic E2E asks...	Trace-based test also asks...
Did the row appear?	Did POST /api/tasks reach create_task, exactly once?
Is the button disabled?	Was every span status OK, with no hidden 500s behind a retry?
Did it load in 10 s?	Did the backend span stay under its latency budget?
Is the text right?	Is task.id on the span the id the UI shows?

The UI can look perfect while the system underneath is on fire. The trace can't hide it.

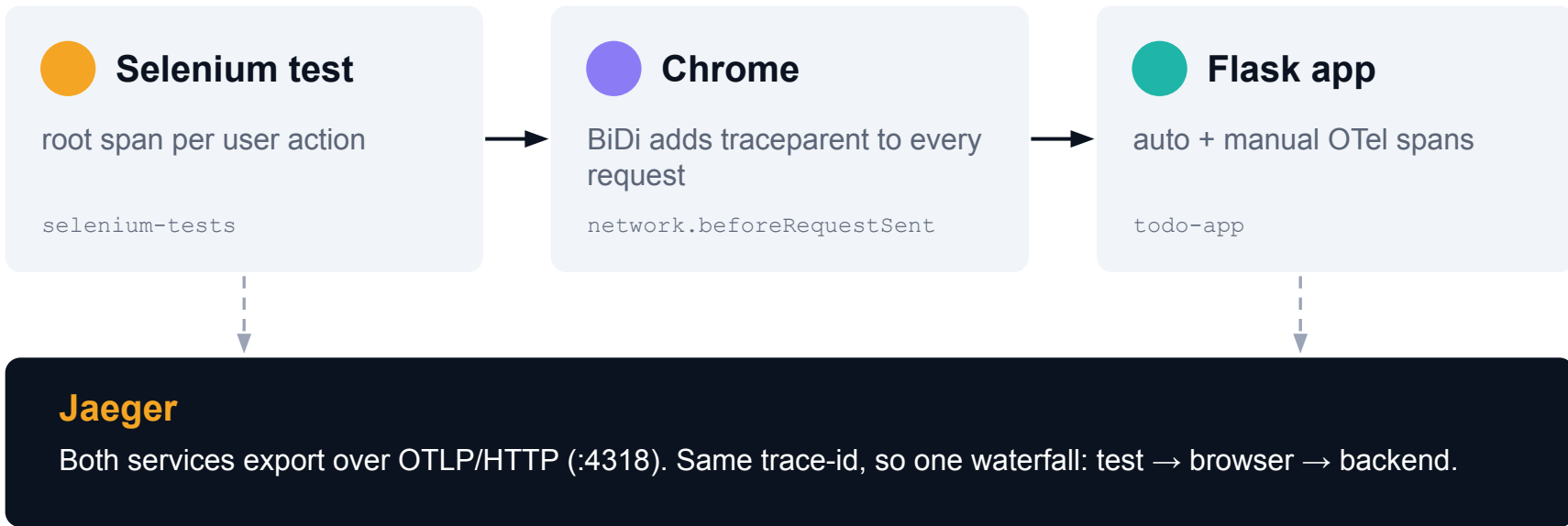


PART 2

One click, one trace

Selenium → BiDi → Chrome → Flask → Jaeger

Making the test the root of the trace



Metrics and trace-correlated logs go to the app console

Every user action is a span

```
@contextmanager
def traced_action(tracer, injector, name):
    with tracer.start_as_current_span(name) as span:
        injector.set_context()      # capture this span's traceparent
        try:
            yield span
        finally:
            injector.set_context({}) # stop injecting

with tracer.start_as_current_span("test: full task lifecycle"):
    with traced_action(tracer, injector, "add task"):
        driver.find_element(*_css("task-title")).send_keys(title)
        driver.find_element(*_css("add-task")).click()
        wait.until(lambda d: title in _task_titles(d))
```

Readable in the test, readable in Jaeger

- Span names read like test steps
- Context is captured on the test thread, then handed to the browser
- Injection stops between steps, so there's no cross-talk

BiDi puts the traceparent on the wire

```
self._driver.network.add_request_handler(
    "before_request", self._handle_request)

def _handle_request(self, request):
    carrier = dict(self._carrier)    # set by the test thread
    drop = ("traceparent", "tracestate")
    headers = [h for h in request.headers or []
               if h.get("name", "").lower() not in drop]
    for name, value in carrier.items():
        headers.append({"name": name,
                        "value": {"type": "string", "value": value}})
    with _SEND_LOCK:
        request.continue_request(headers=headers)
```

Why BiDi?

- Classic WebDriver can't set request headers
- BiDi's network module intercepts before the request is sent
- Always continue the request, even on error, or the browser hangs

LIVE DEMO

Switch to terminal

1

```
./scripts/run_jaeger.sh
```

Jaeger UI on :16686

2

```
./scripts/run_app.sh
```

to-do app on :5000

3

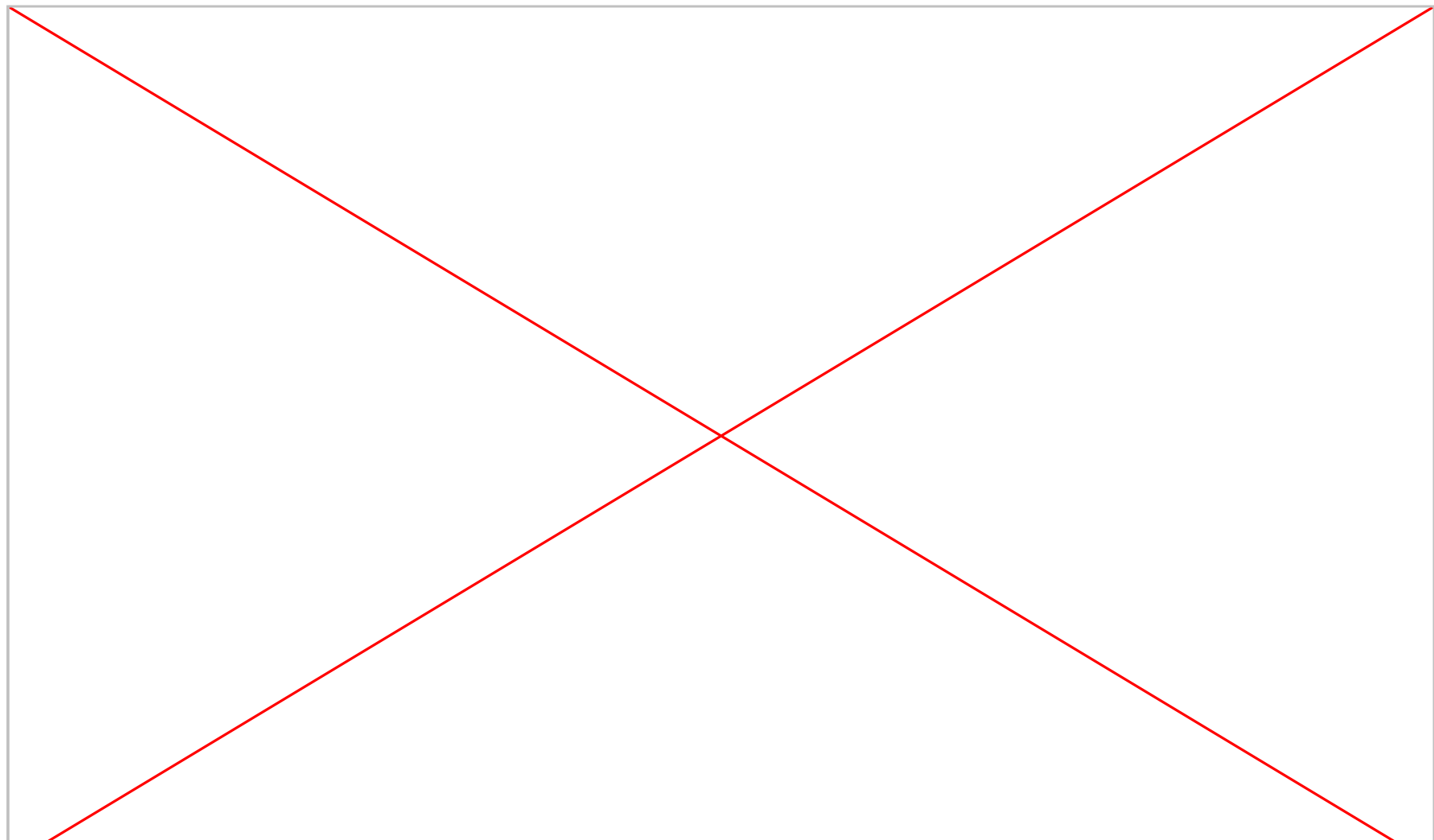
```
HEADLESS=0 .venv/bin/pytest tests/ -v
```

watch Chrome drive the app

4

```
Jaeger → service: selenium-tests
```

open the newest trace



One trace, two services



Two gotchas that will bite you



Navigate with BiDi, not driver.get()

With a network intercept active, classic navigation never gets its load signal and times out: “Timed out receiving message from renderer”.

```
driver.browsing_context.navigate(handle,  
url, wait="complete")
```



Serialise continue_request()

The Python BiDi websocket isn't thread-safe. Concurrent calls interleave frames and drop the connection, leaving requests hung.

```
with _SEND_LOCK:  
    request.continue_request(headers=headers)
```

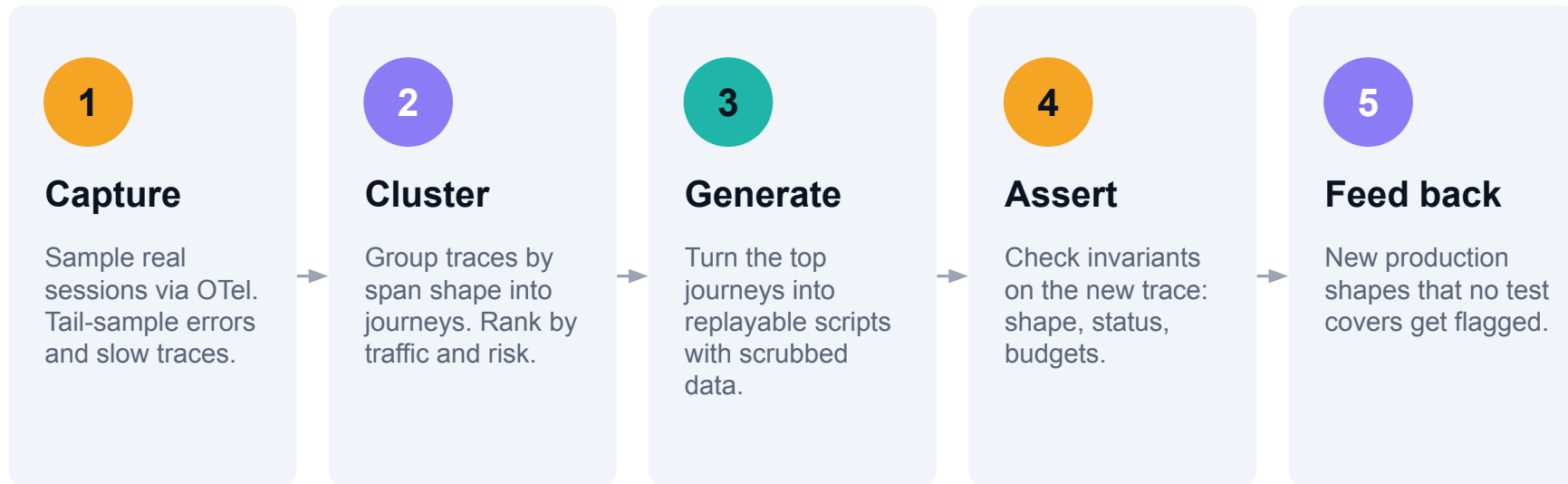


PART 3

From production traces to tests

Real-user journeys as your regression suite

Turning traces into integration tests



The same trace format flows both ways: production → test cases, and tests → traces you can diff against production.

From expected values to invariants

```
# tests/test_task_flow.py::test_add_task_trace
spans = jaeger.get_trace(trace_id, expect=["create_task"])

add = spans.child_of("add task")
assert add.has("POST /api/tasks")
assert add.has("create_task", attrs={"task.id": ui_task_id})
assert all(s.status != "ERROR" for s in spans)
assert spans.op("create_task").duration_ms < 50

# AI agent (illustrative): shape, not string
assert spans.order("retrieve", "plan", "tool:search")
assert spans.count("llm.call") <= 3
```

Shape

the right services, in the right order

Health

no error spans, including behind retries

Budget

latency and call-count limits

Identity

UI data matches span attributes

Start small, and watch the sharp edges

DO THIS

- 1 Instrument one service with OTel auto-instrumentation
- 2 Propagate traceparent from your E2E tests (BiDi, or a proxy)
- 3 Add one trace assertion: no error spans
- 4 Mine production traces for your top 5 journeys

WATCH OUT FOR

- PII in span attributes: scrub before replay
- Sampling: a missing span is not always a bug
- Exporter flush: `force_flush()` before you query
- Cardinality and cost of high-volume telemetry



Stop guessing. Start seeing.

- Telemetry is the oracle
- The test is the root of the trace
- Production journeys are your regression suite

Thank you

David Burns
github.com/AutomatedTester