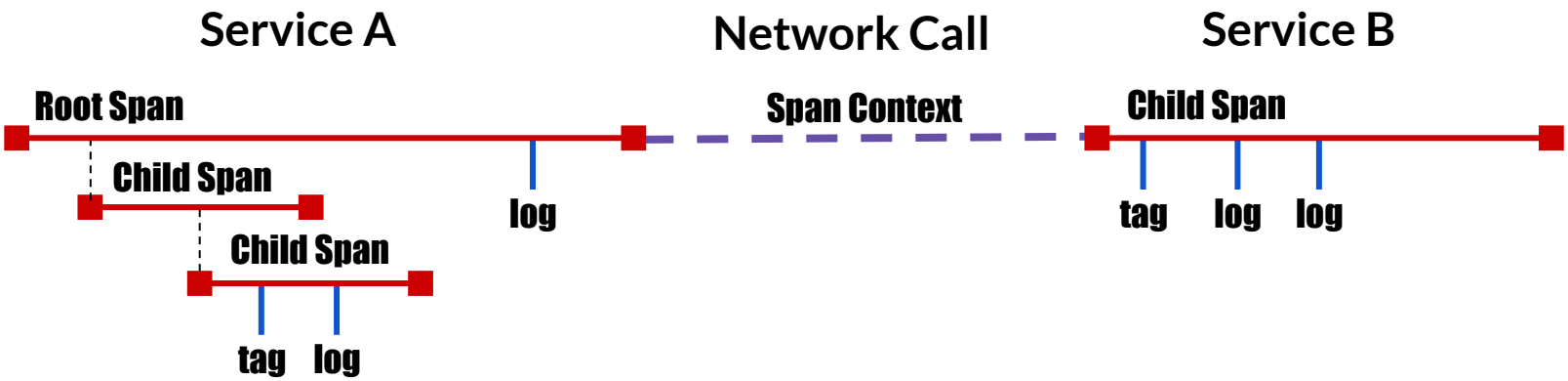
Trace Driven Testing

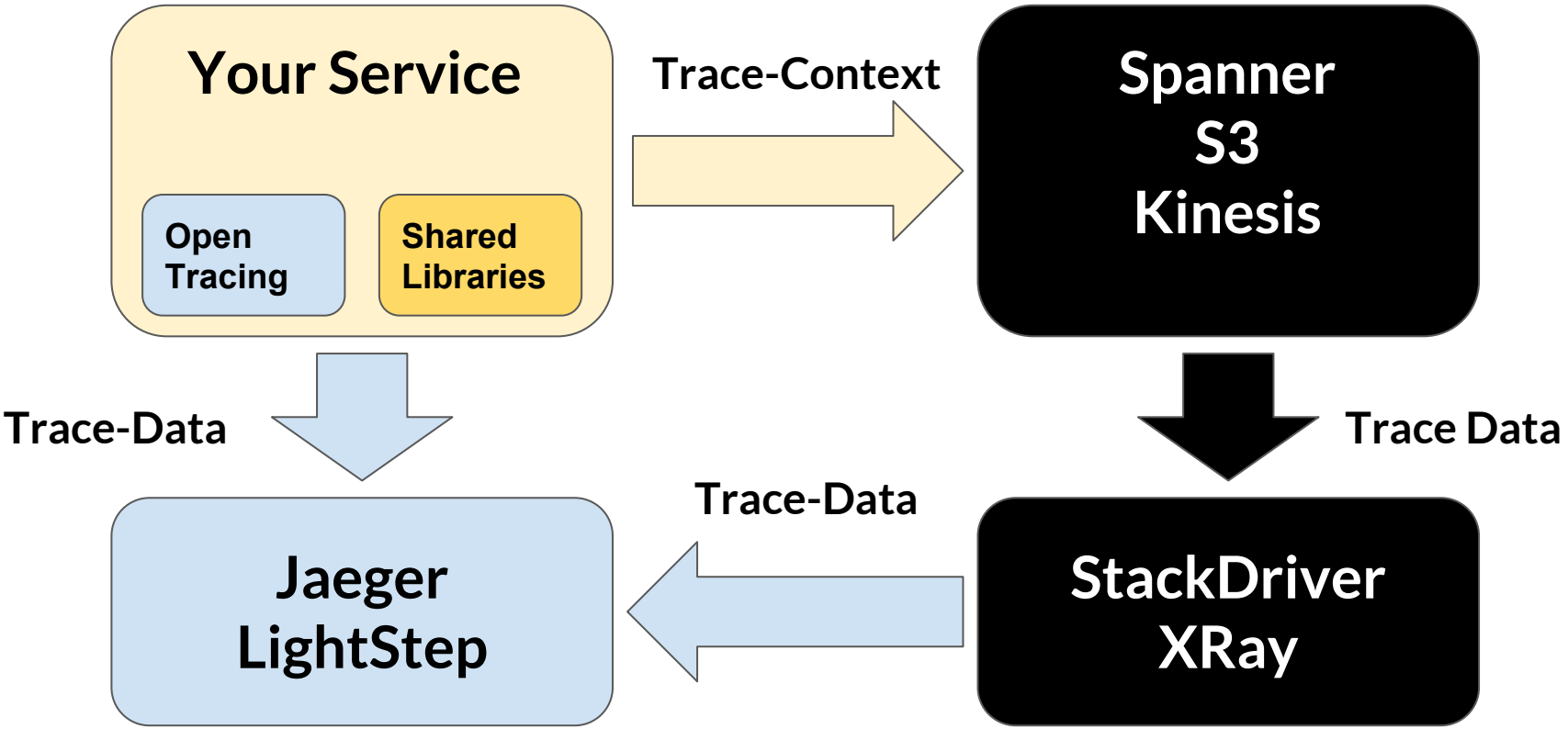
Ted Young, LightStep

Distributed Tracing: A Mental Model

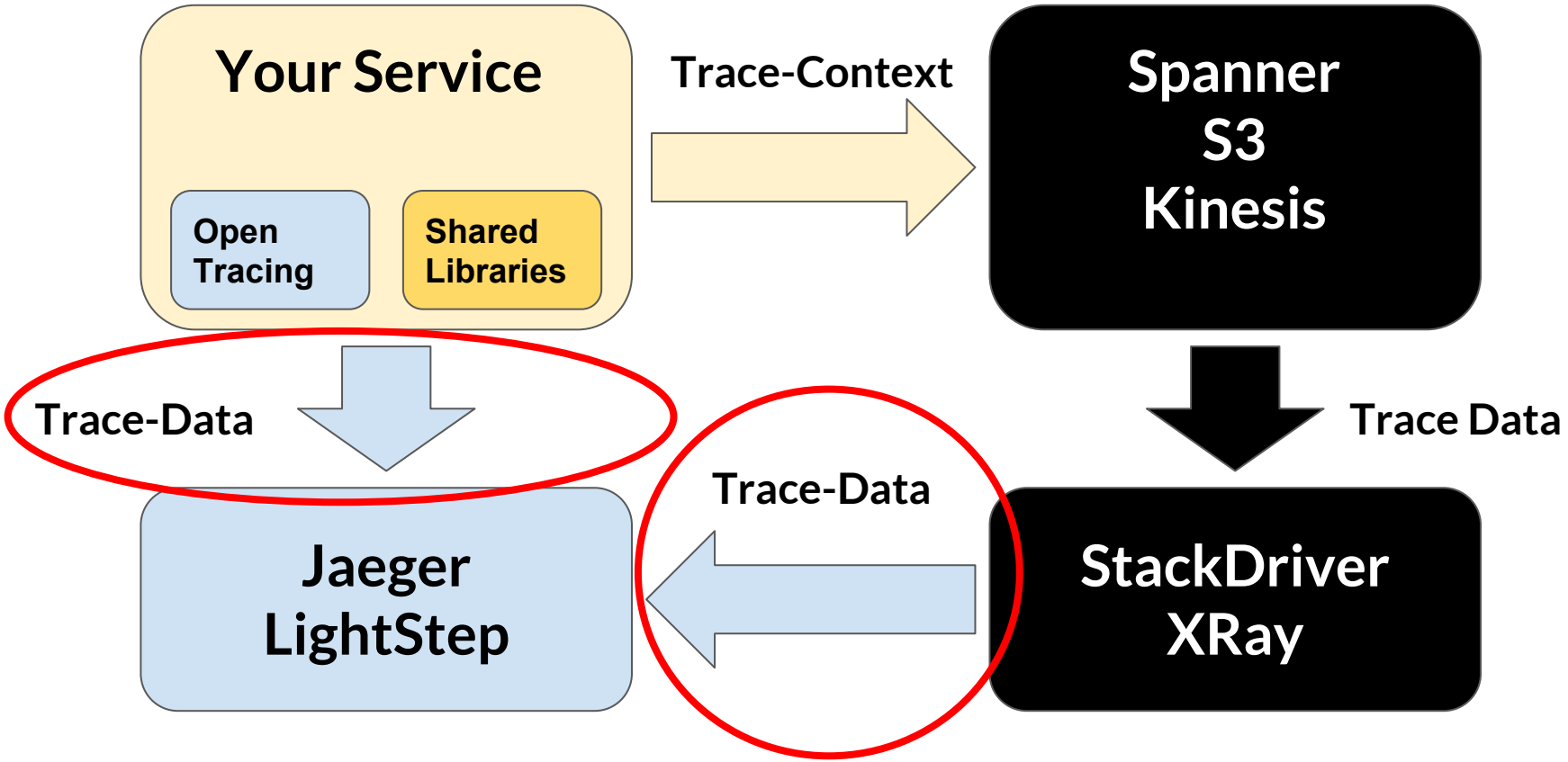


- **Trace:** A recording of a transaction as it moves through a distributed system. Traces are represented as a directed acyclic graph (DAG) of **Spans.**
- **Span:** A named, timed operation representing a piece of the workflow. Spans have a **Timestamp** and a **Duration**, and are annotated with **Tags** and **Logs**.
- **Span Context:** A set of **Trace Identifiers** injected into each network request, which the next service will extract and use in order to propagate the trace.

Distributed Tracing: An Architectural Model



Distributed Tracing: An Architectural Model



Distributed Tracing: Trace-Data

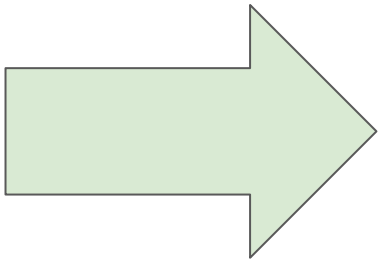
```
type Trace {
  repeated Span spans
}

type Span {
  required string traceID
  required string spanID
  required string operationName
  required int startTime
  required int endTime
  repeated Reference references
  repeated Tag tags
  repeated Log logs
}
```

```
type HttpClientTag : Tag {
  required string url
  required string httpmethod
  optional int statusCode
  repeated KeyValuePair requestheaders
  repeated KeyValuePair responseheaders
}

type DbClientTag : Tag {
  required string dbType
  required string dbInstance
  required string user
  required string statement
}
```

Distributed Tracing: New Test Flow

Unit Tests

- Before/Setup
- Test/Run
- After/Reset

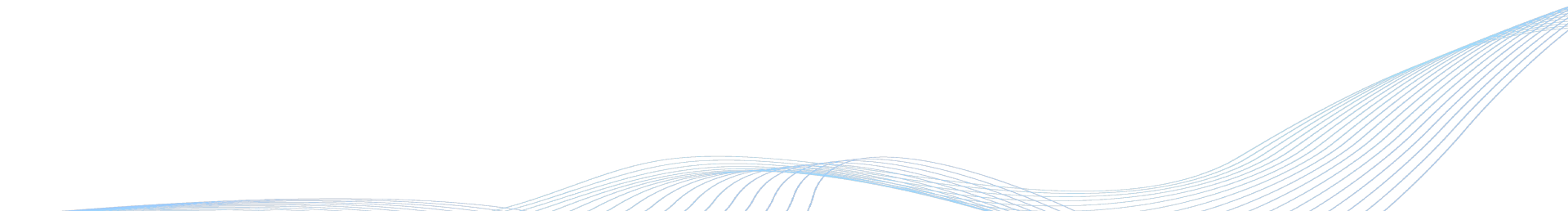


Trace Tests

- Define Model
- Gather Data
- Check Model

Example: Modeling a Bank Withdrawal

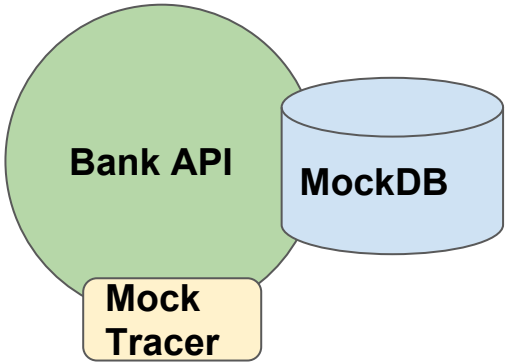
- We’re building a bank...
- We want to ensure that accounts cannot withdraw more money than they contain.
- Let’s define a test for this!



Example: Modeling a Bank Withdrawal

```
model = NewModel()
model("Accounts cannot withdraw more than their balance")
      .When(
        LessThan(
            Span.Name("fetch-balance").Tag("amount"),
            Span.Name("withdrawal").Tag("amount")))
      .Expect( Span.Name("rollback") )
      .NotExpect( Span.Name("commit") )
      .Expect( Span.Name("/:account/withdrawl/")
                    .HttpStatusCode(500))

Check(model, testData)
```

Example: Unit Test

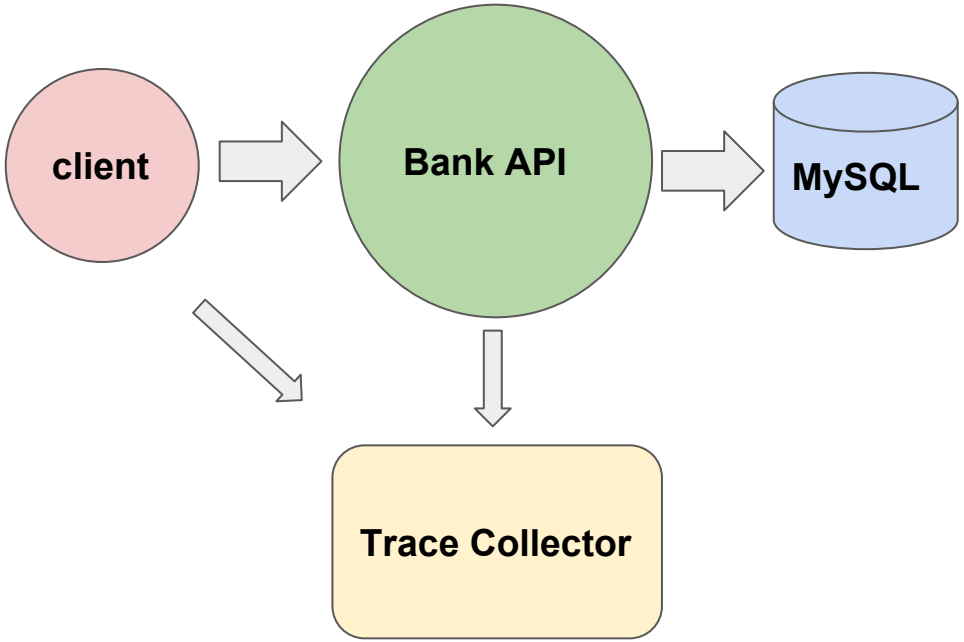



```
tracer = NewMockTracer()
mockDB = NewMockDatabase()
bankServer =
  NewBankAPI(tracer, MockDB)

account = 123

mockDB.getBalanceReturns(300)

bankServer.withdraw(account,500)

testData = tracer.GetData()
```

Example: Modeling a Bank Withdrawal

```
model = NewModel()
model("Accounts cannot withdraw more than their balance")
      .When(
        LessThan(
            Span.Name("fetch-balance").Tag("amount"),
            Span.Name("withdrawal").Tag("amount")))
      .Expect( Span.Name("rollback") )
      .NotExpect( Span.Name("commit") )
      .Expect( Span.Name("/:account/withdrawl/")
                    .HttpStatusCode(500))

Check(model, testData)
```

Example: Integration Test

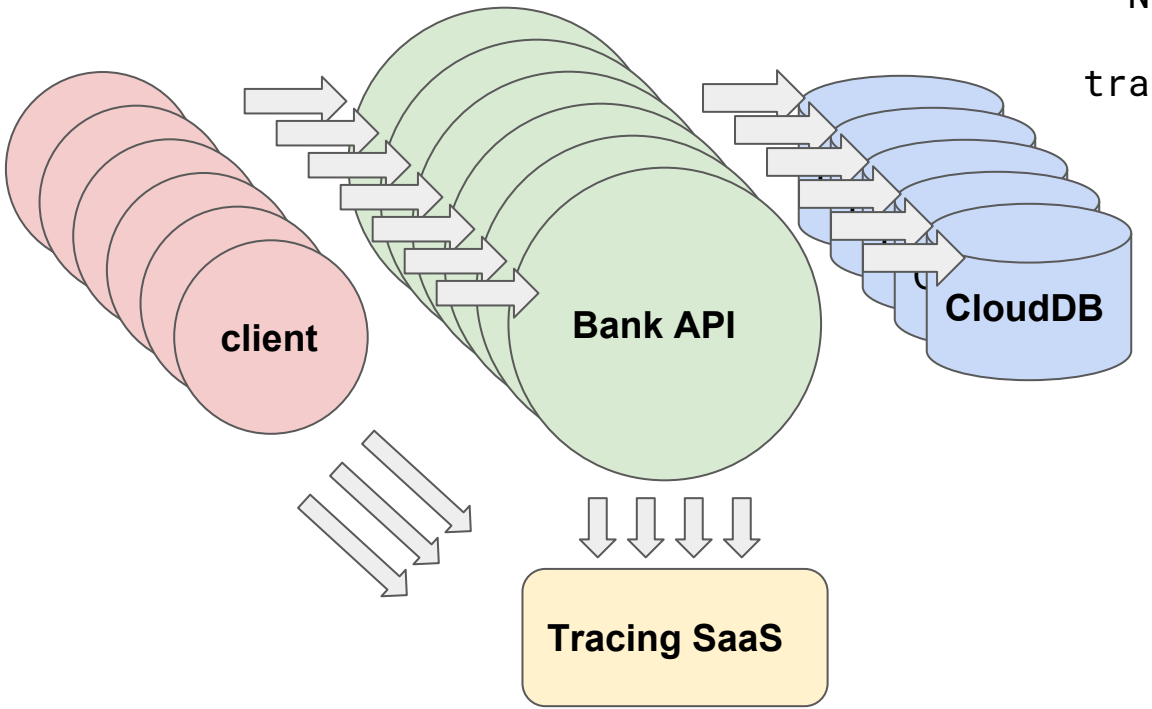



```
exec(`start trace_collector`)
exec(`start bank`)
exec(`start mysql`)
exec(`setup_test_db`)

account = 123
client =
  NewClient("localhost", account)

balance = client.balance()
client.withdraw(balance*2)

testData =
  fetchTraceData("localhost")

exec(`stop bank`)
exec(`stop mysql`)
exec(`stop trace_collector`)
```

Example: Modeling a Bank Withdrawal

```
model = NewModel()
model("Accounts cannot withdraw more than their balance")
      .When(
        LessThan(
            Span.Name("fetch-balance").Tag("amount"),
            Span.Name("withdrawal").Tag("amount")))
      .Expect( Span.Name("rollback") )
      .NotExpect( Span.Name("commit") )
      .Expect( Span.Name("/:account/withdrawl/")
                    .HttpStatusCode(500))

Check(model, testData)
```

Example: Production Test




```
client =
  NewTraceClient(“api.tracing.com”)

traceData = client.streamData()
```

Example: Modeling a Bank Withdrawal

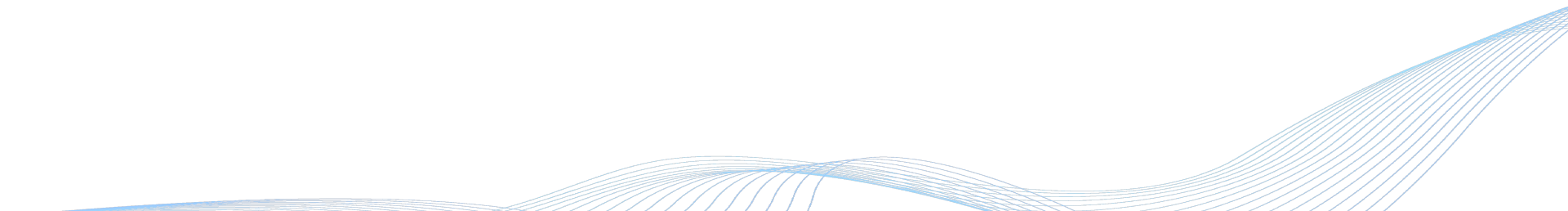
```
model = NewModel()
model("Accounts cannot withdraw more than their balance")
      .When(
        LessThan(
            Span.Name("fetch-balance").Tag("amount"),
            Span.Name("withdrawal").Tag("amount")))
      .Expect( Span.Name("rollback") )
      .NotExpect( Span.Name("commit") )
      .Expect( Span.Name("/:account/withdrawl/")
                    .HttpStatusCode(500))

Check(model, testData)
```

Formal Specification

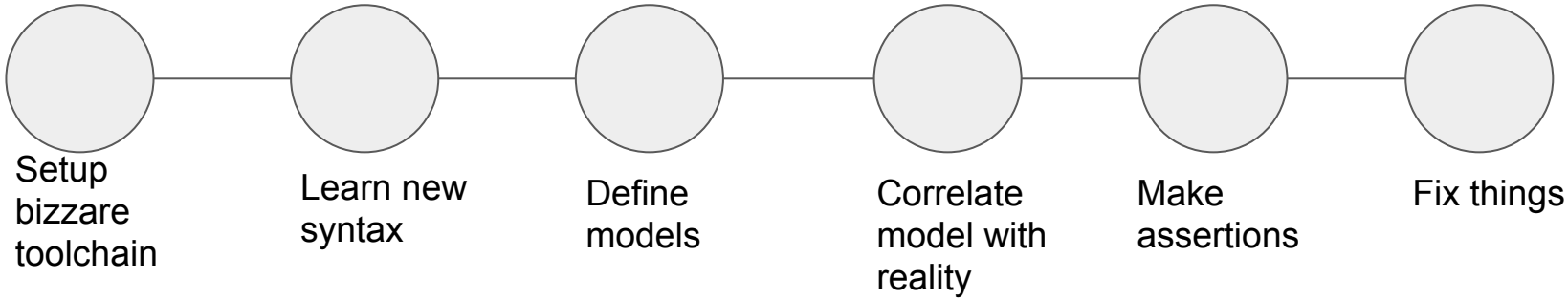
```
\* Server i times out and starts a new election.
Timeout(i) == /\ state[i] \in {Follower, Candidate}
              /\ state' = [state EXCEPT ![i] = Candidate]
              /\ currentTerm' = [currentTerm EXCEPT ![i] = currentTerm[i] + 1]
              \* Most implementations would probably just set the local vote
              \* atomically, but messaging localhost for it is weaker.
              /\ votedFor' = [votedFor EXCEPT ![i] = Nil]
              /\ votesResponded' = [votesResponded EXCEPT ![i] = {}]
              /\ votesGranted'   = [votesGranted EXCEPT ![i] = {}]
              /\ voterLog'       = [voterLog EXCEPT ![i] = [j \in {} |-> <<>>]]
              /\ UNCHANGED <<messages, leaderVars, logVars>>
```

Formal Specification

- Modeling is difficult
- Reality is totally separate from the verified model
- Toolchain is ad-hoc, painful, unfamiliar
- The starting point feels miles away from the finish line



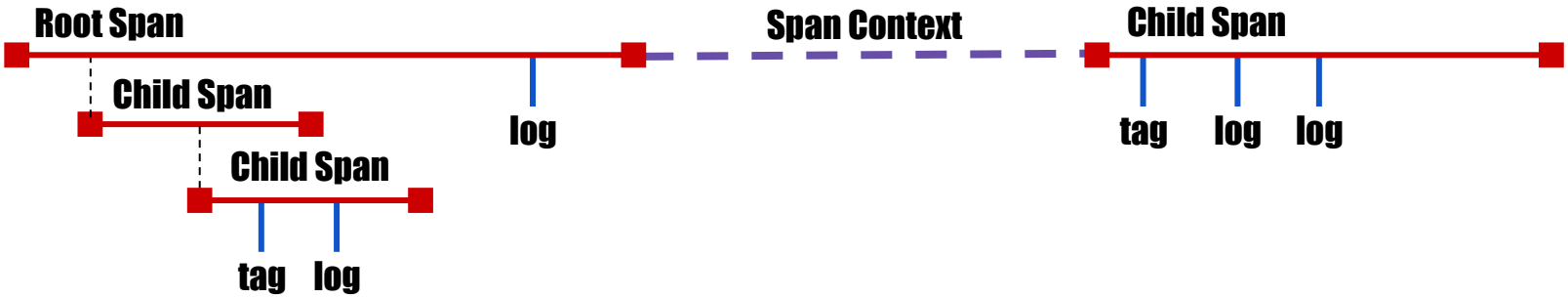
The starting point feels miles away from the finish line



Formal Specification

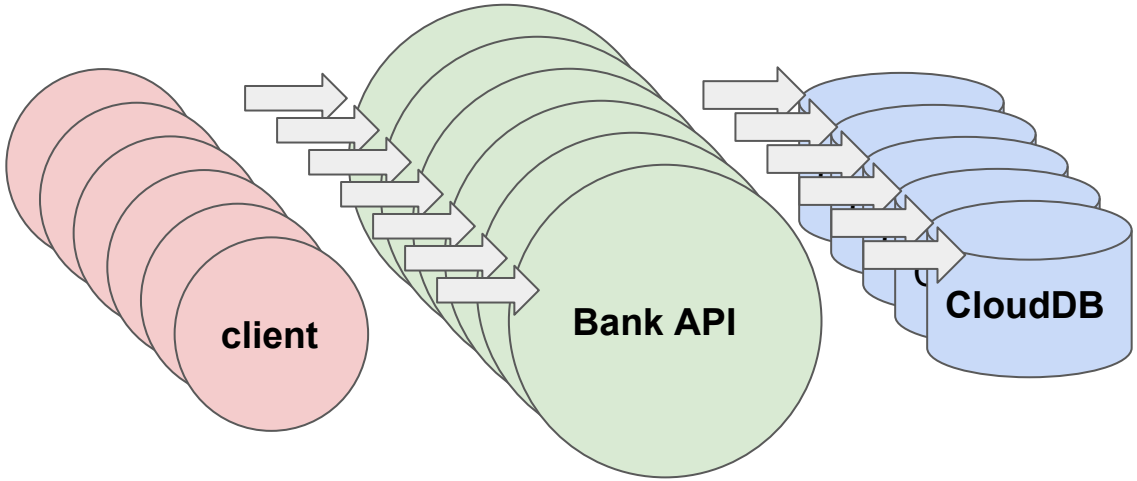
- Modeling is difficult
  - **Distributed tracing is a simple, standard model**
- Reality is totally separated from the verified model
  - **You are running against your actual code**
- Tooling and environment is ad-hoc, unfamiliar, painful...
  - **Tooling is… you guessed it...**

Formal Specification: Modeling is difficult



Distributed tracing is a simple, standard model.

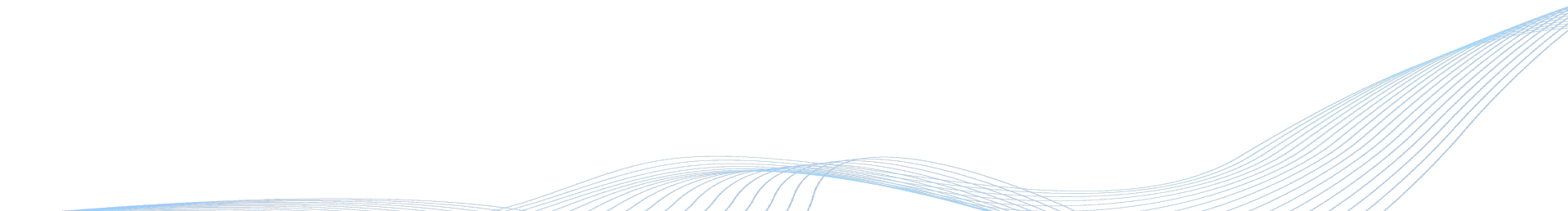
Formal Specification: Reality is separate from model



Reality is extremely similar to the model.

Formal Specification: Toolchain is ad-hoc, unfamiliar

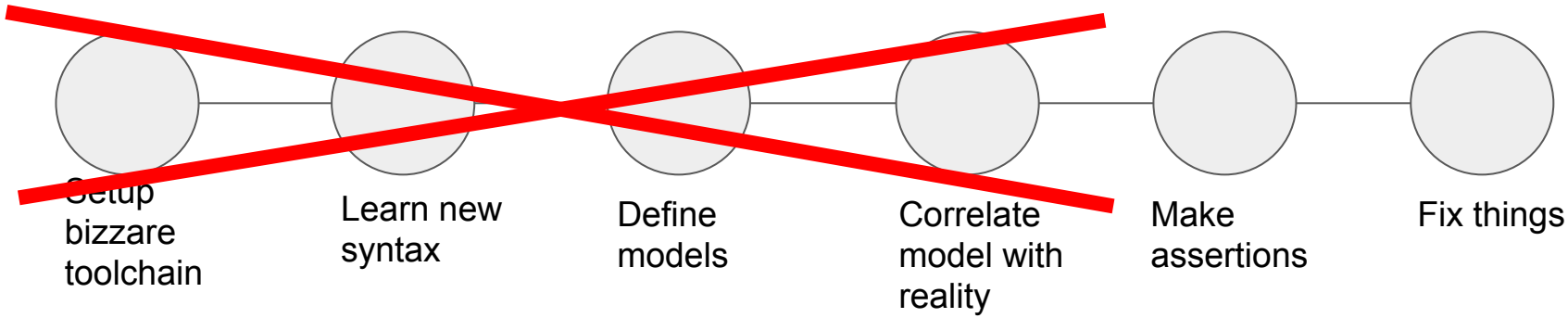
You guessed it...



Formal Specification: Toolchain is ad-hoc, unfamiliar

```
model = NewModel()
model("Accounts cannot withdraw more than their balance")
      .When(
        LessThan(
            Span.Name("fetch-balance").Tag("amount"),
            Span.Name("withdrawal").Tag("amount")))
      .Expect( Span.Name("rollback") )
      .NotExpect( Span.Name("commit") )
      .Expect( Span.Name("/:account/withdrawl/")
                    .HttpStatusCode(500))

Check(model, testData)
```

The starting point is much closer now



Notable Issues

Only works if trace points are accurate

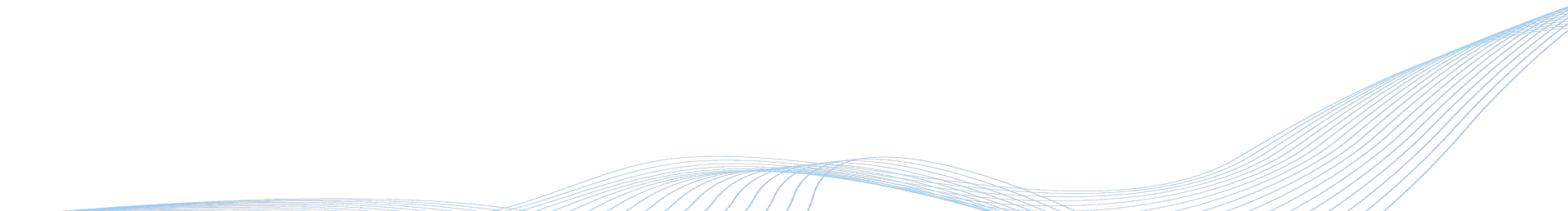
- How do we ensure trace points are 1:1 with actual code execution?
- Hmmmm…. shouldn't we be checking that anyways?

No "outside observer"

- Can only use data from code we run
- Can't double check results by executing additional, non-production code
- Well, you can in test environments. But it's artificial and perhaps misses the point...

Why does testing matter for observability?

- Our development practices are totally divorced from our monitoring practices.
- But our monitoring practices depend on our development practices.
- Automation might help, but it's no panacea.
- If our observability code is not helpful in development, there's no feedback loop.
- Quality always suffers.

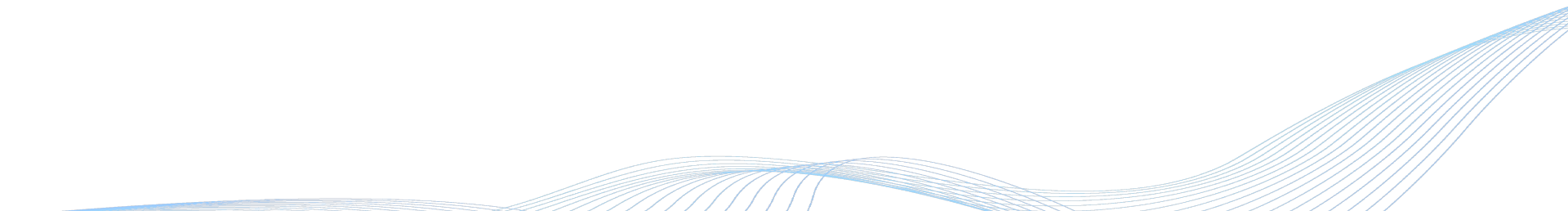


Why does this matter for development?

- All test environments are artificial.
- It's literally crazy that we don't test (well) in production.
- Smoke tests are crude tests.
- SLOs are crude tests.
- If running your tests against production wasn't so hard... you would do it!
- If you could easily write more production tests as part of triaging a problem… you would do it!

Data Driven Development, Data Driven Monitoring

- Trace Data is just… data.
- Trace Driven Development is really just Data Driven Development
- There is no reason you cannot write the exact same kinds of tests against aggregate data: resource usage, latency outliers, error rates.
- We have so many tools for analyzing structured data… let's use them all!



What's Next: We need Trace-Data

```
type Trace {
  repeated Span spans
}

type Span {
  required string traceID
  required string spanID
  required string operationName
  required int startTime
  required int endTime
  repeated Reference references
  repeated Tag tags
  repeated Log logs
}
```

```
type HttpClientTag : Tag {
  required string url
  required string httpmethod
  optional int statusCode
  repeated KeyValuePair requestheaders
  repeated KeyValuePair responseheaders
}

type DbClientTag : Tag {
  required string dbType
  required string dbInstance
  required string user
  required string statement
}
```

What's Next: We need Trace-Data

W3C Trace-Context Working Group

- https://github.com/w3c/trace-context
- Currently defining a wire protocol for in-band context propagation.
- AKA, standard HTTP headers for distributed tracing.
- Will help enable multiple tracing systems to participate in the same trace.
- Which means you can potentially pull trace data from 3rd party services you connect to, and add it to your application's trace data.
- Wait… did someone say trace data?

THANKS!!!!!

Next Steps

- @tedsuo on twitter for updates.
- Help make Trace-Data real!
- Play with trace-based testing interfaces.
- Experiment with temporal logic, model checkers, and other fun.
- Let me know what you're up to and I'll retweet it.