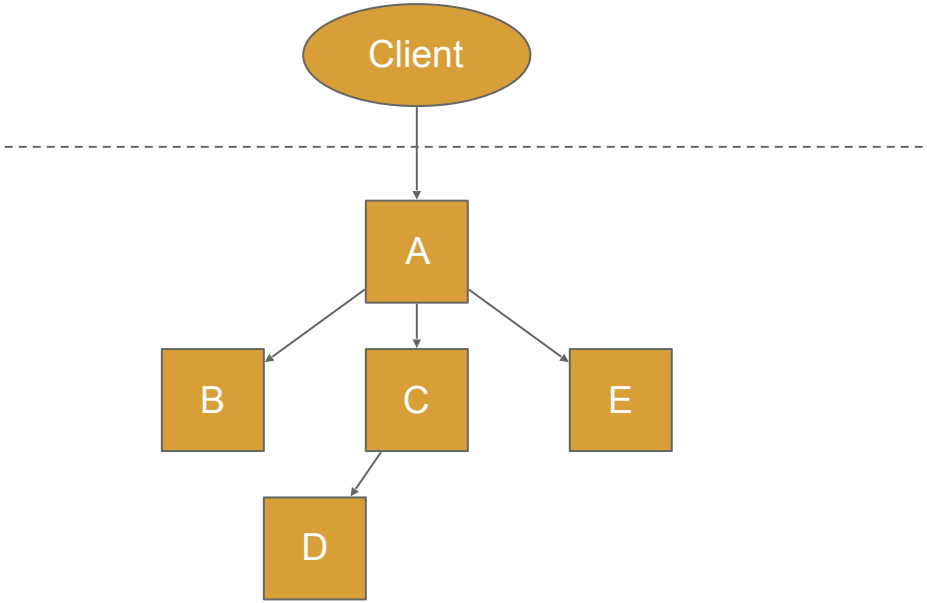
Instrumenting Mayhem Functional Chaos Engineering

Szymon Mentel
szymon.mentel@erlang-solutions.com
@szymonmetel
github.com/mentels

1.

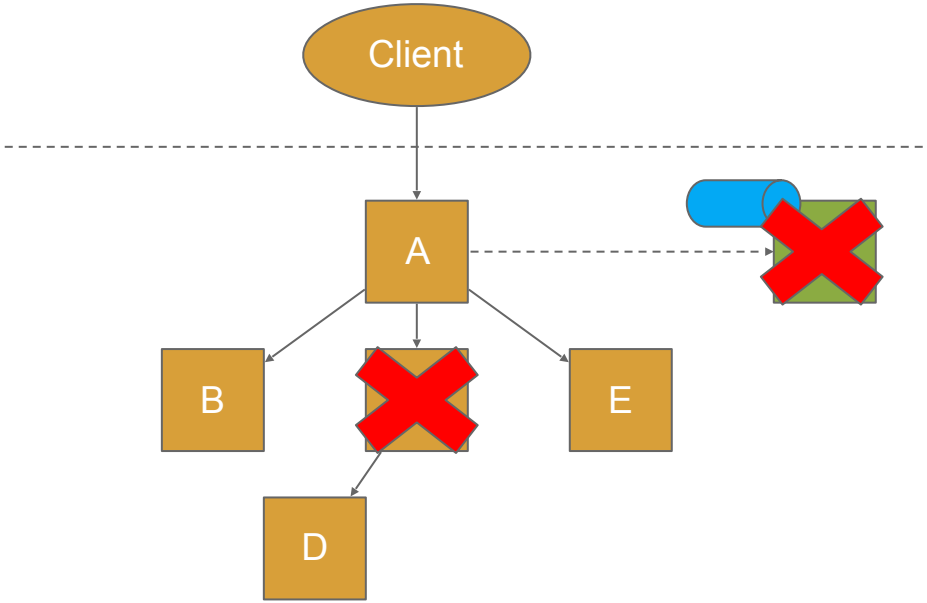
Chaos: WHAT WHY HOW

WHAT is Chaos: complex systems



WHAT is Chaos: complex systems

Interactions compounded with **real-world** events
may lead to unpredictable outcomes

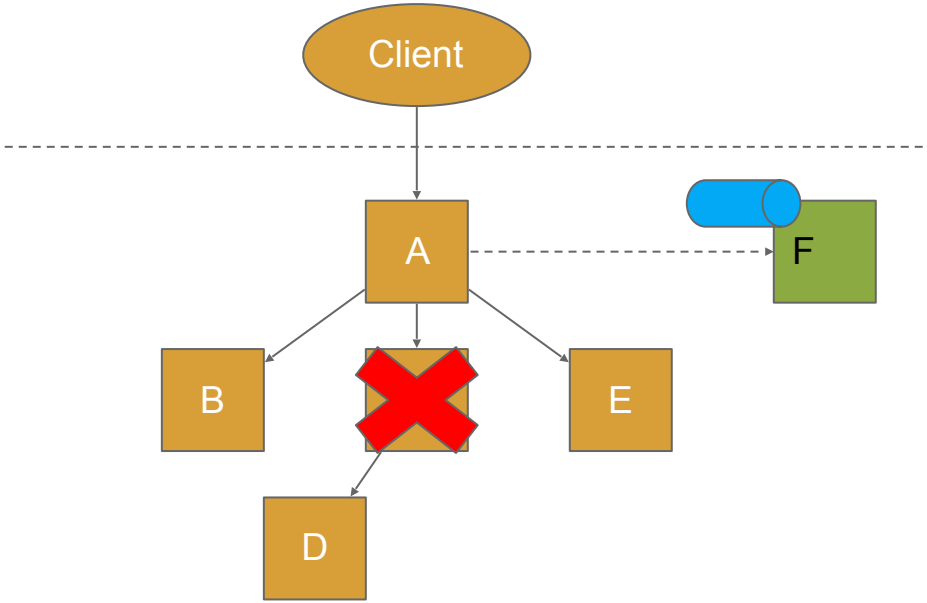


“

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.

~ http://principlesofchaos.org/

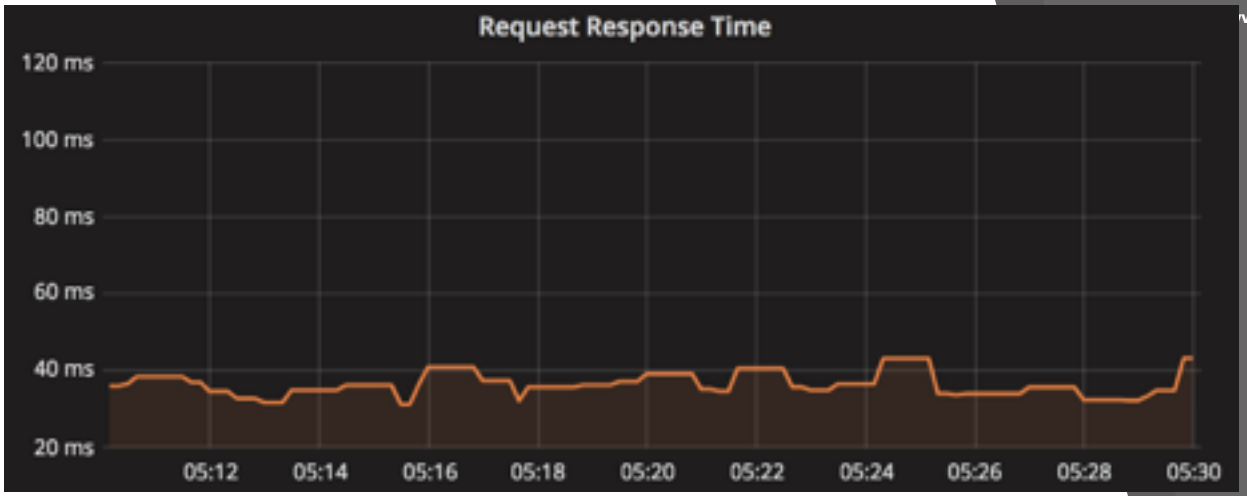
Chaos Engineering: experimenting

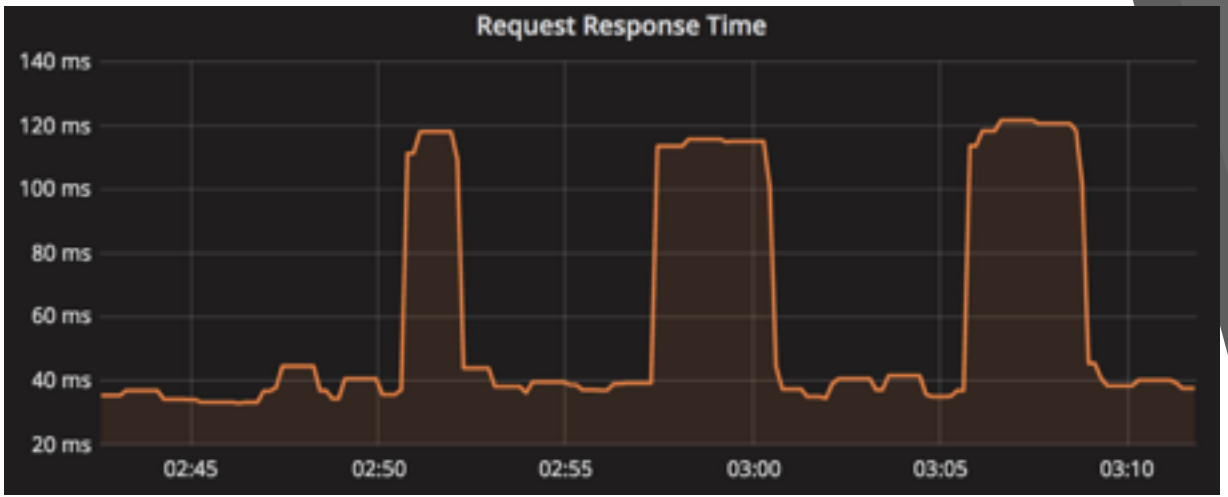


WHY

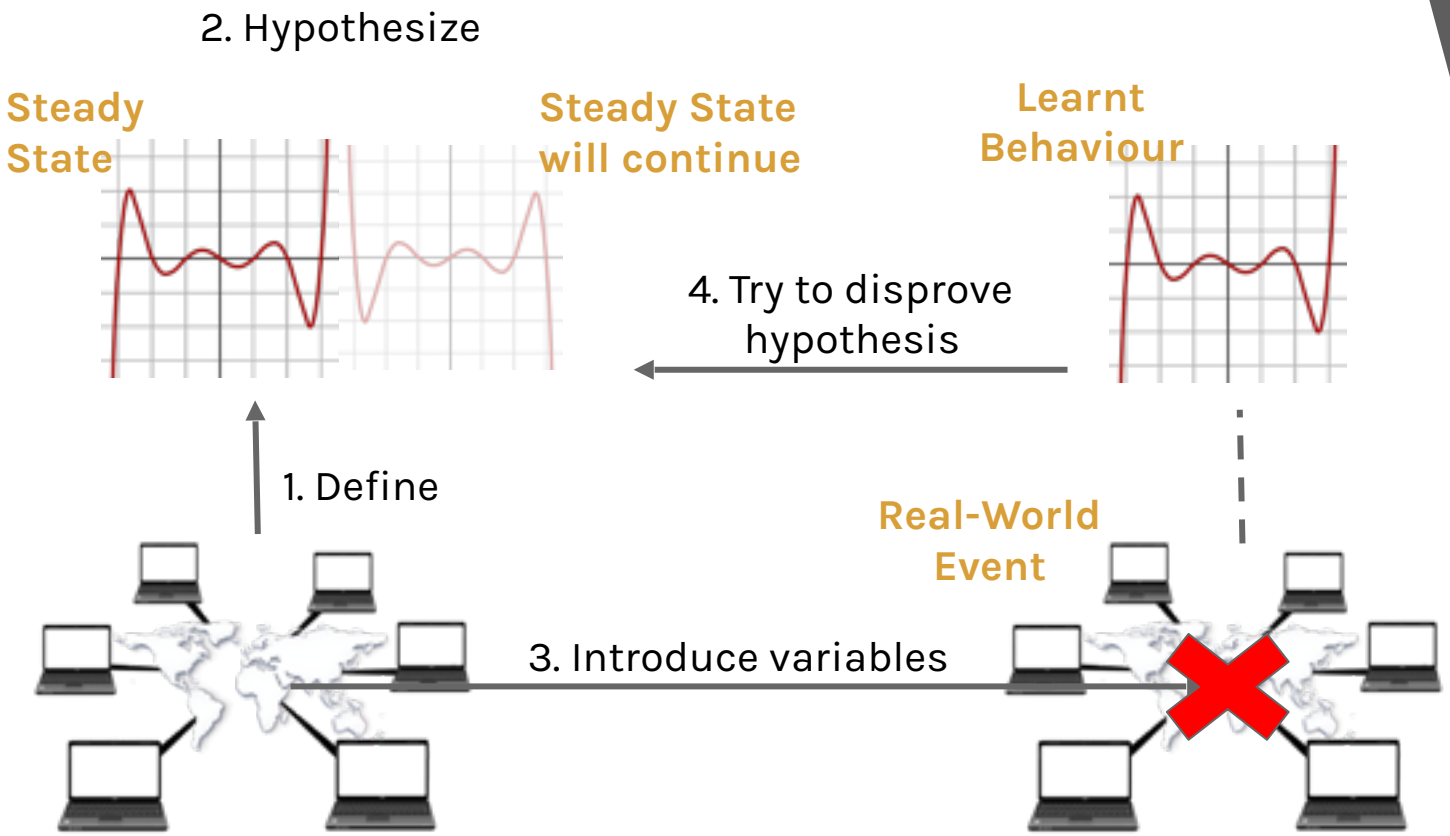
What is the rationale for Chaos Engineering?

HOW:
Steady State of a System



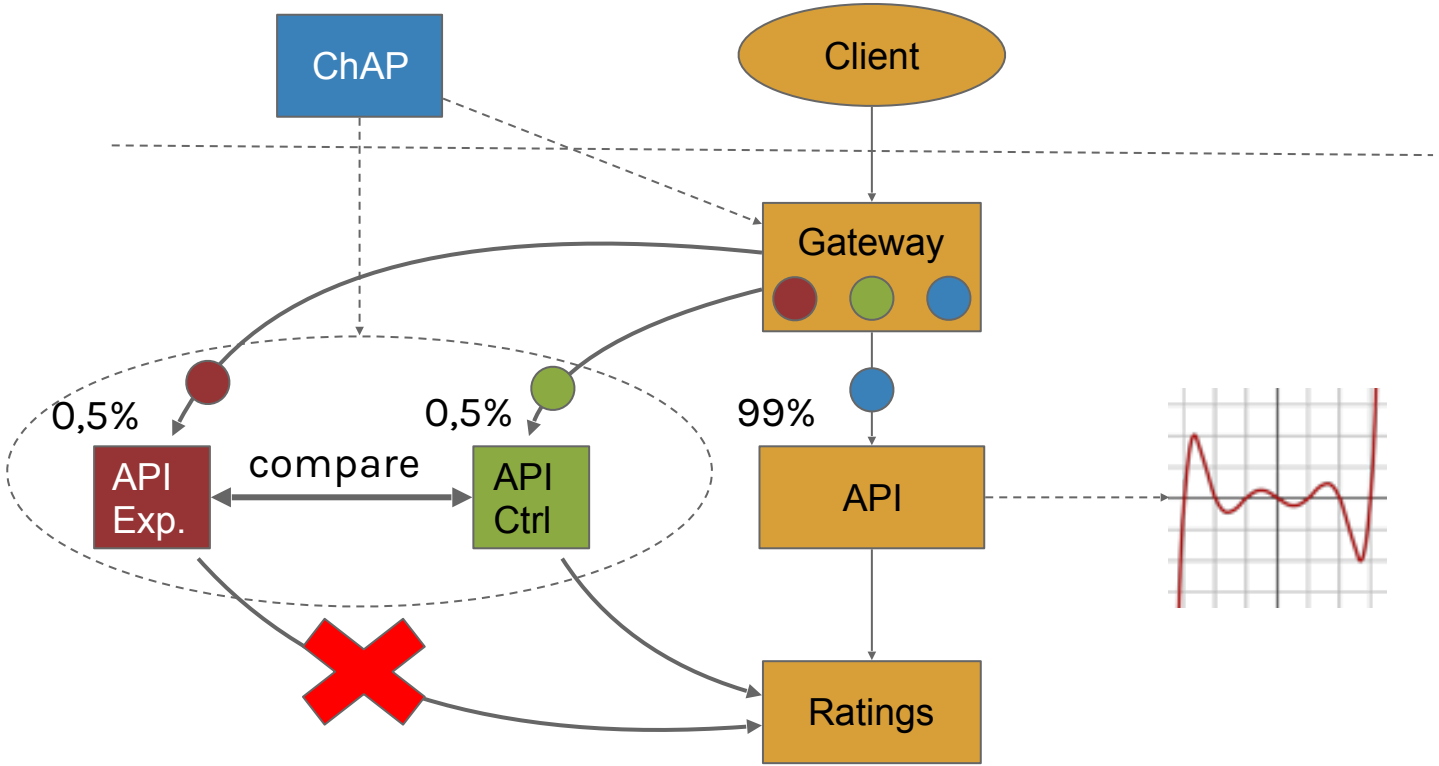


HOW: 4 steps


2. Hypothesize
Steady State
Steady State will continue
Learnt Behaviour
4. Try to disprove hypothesis
1. Define
Real-World Event
3. Introduce variables

HOW: Chaos Engineering at Netflix

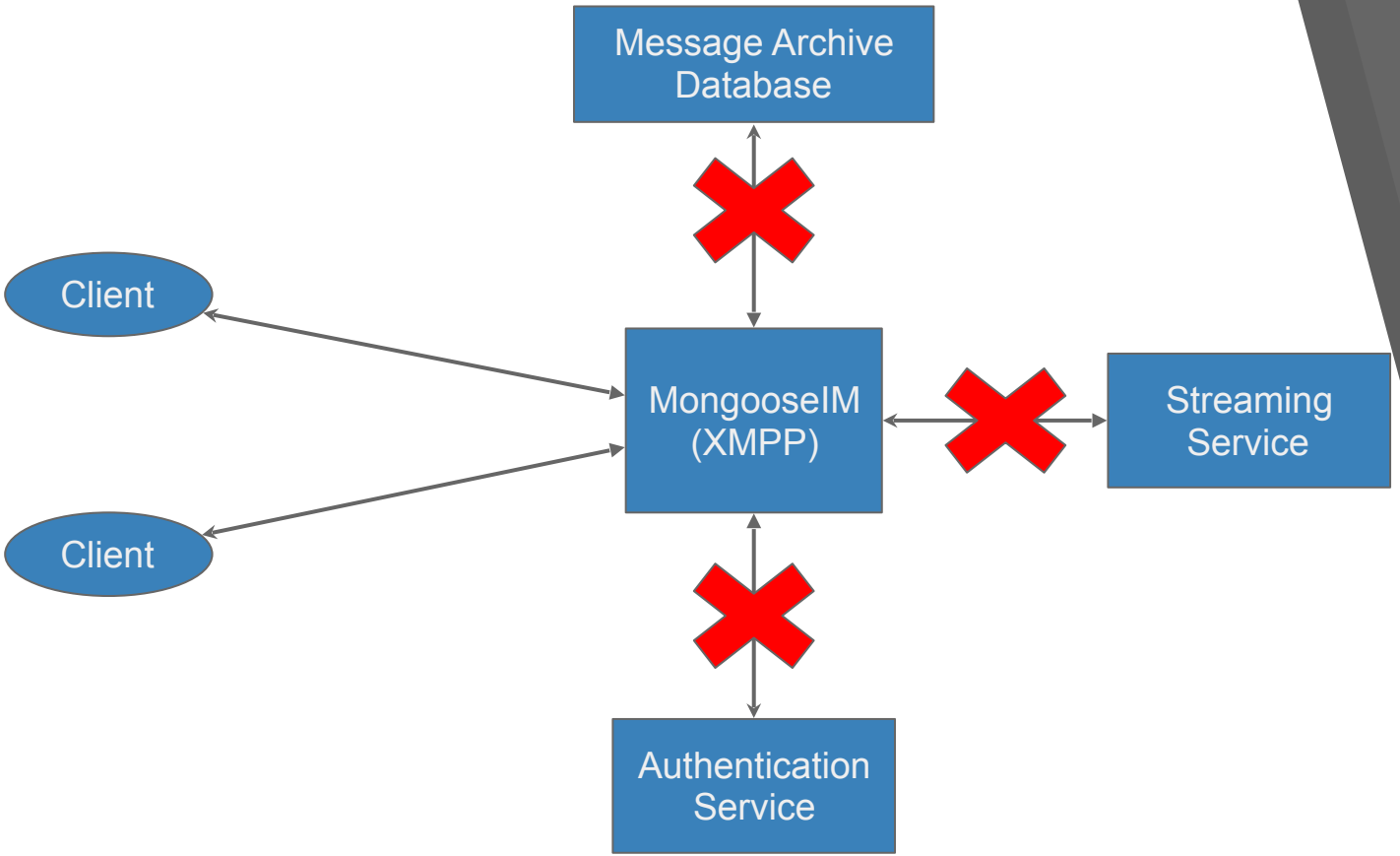
How **API** handles failure of **Ratings** service?



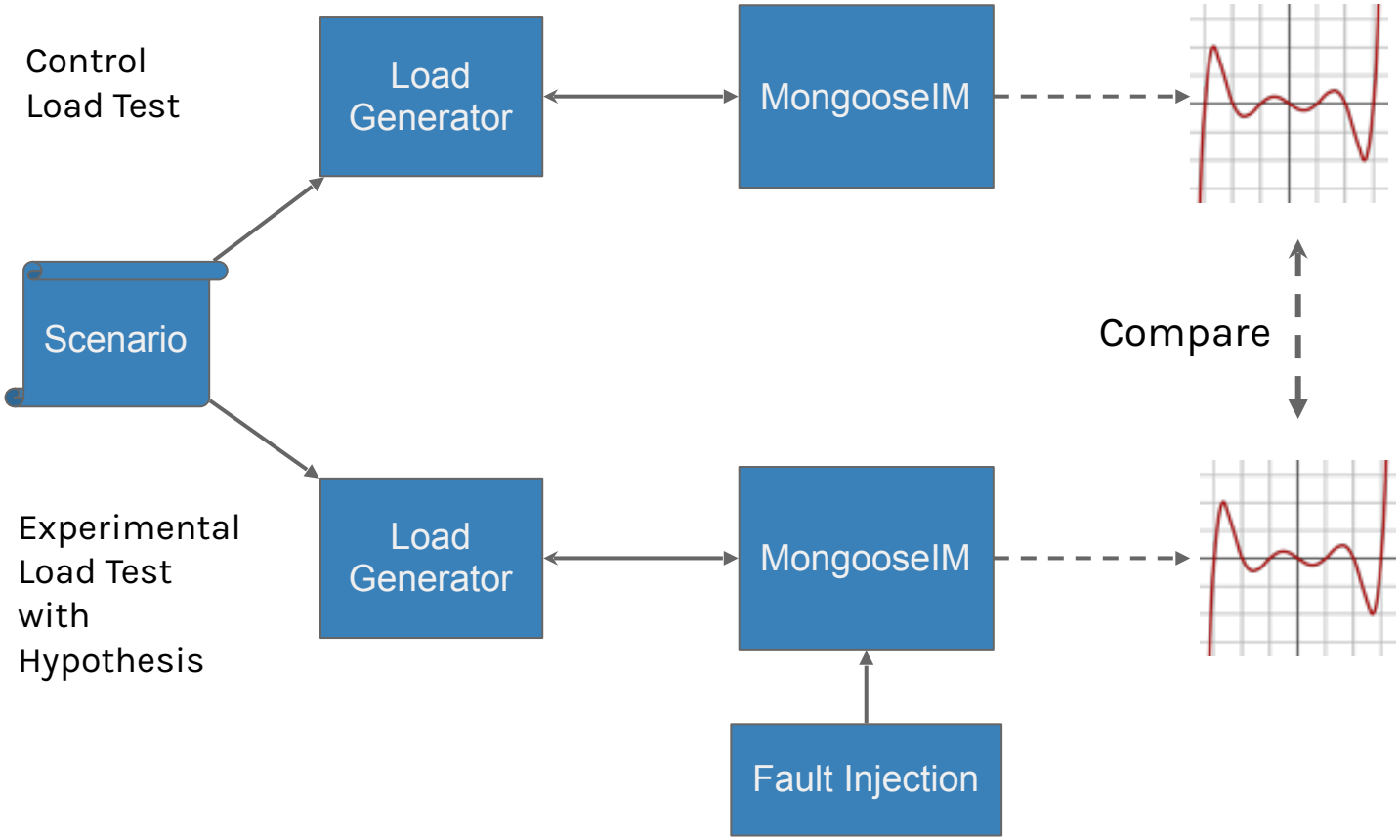
2.

Applying Chaos Engineering Principles

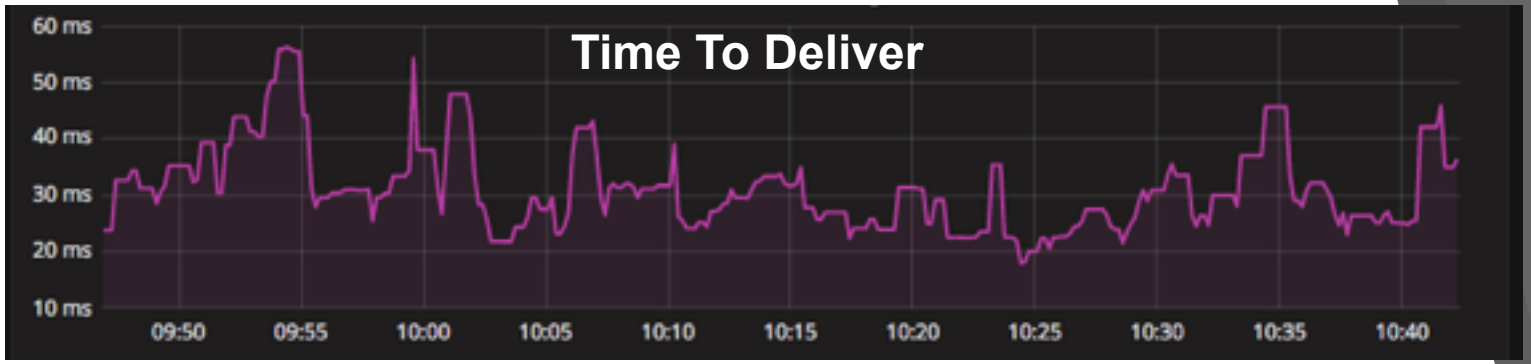
Applying ChE: Injection Points

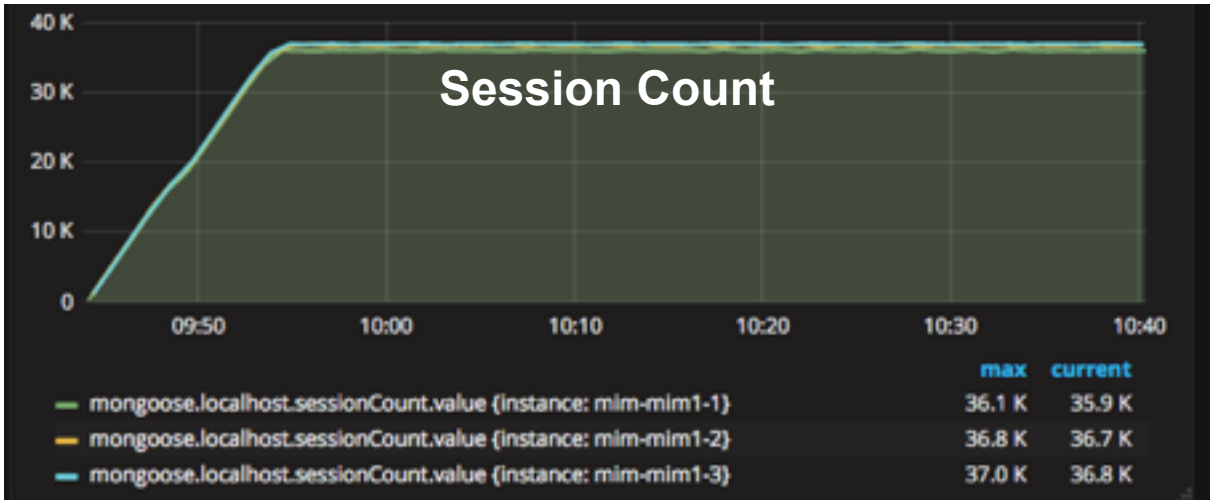


Applying ChE: Comparing



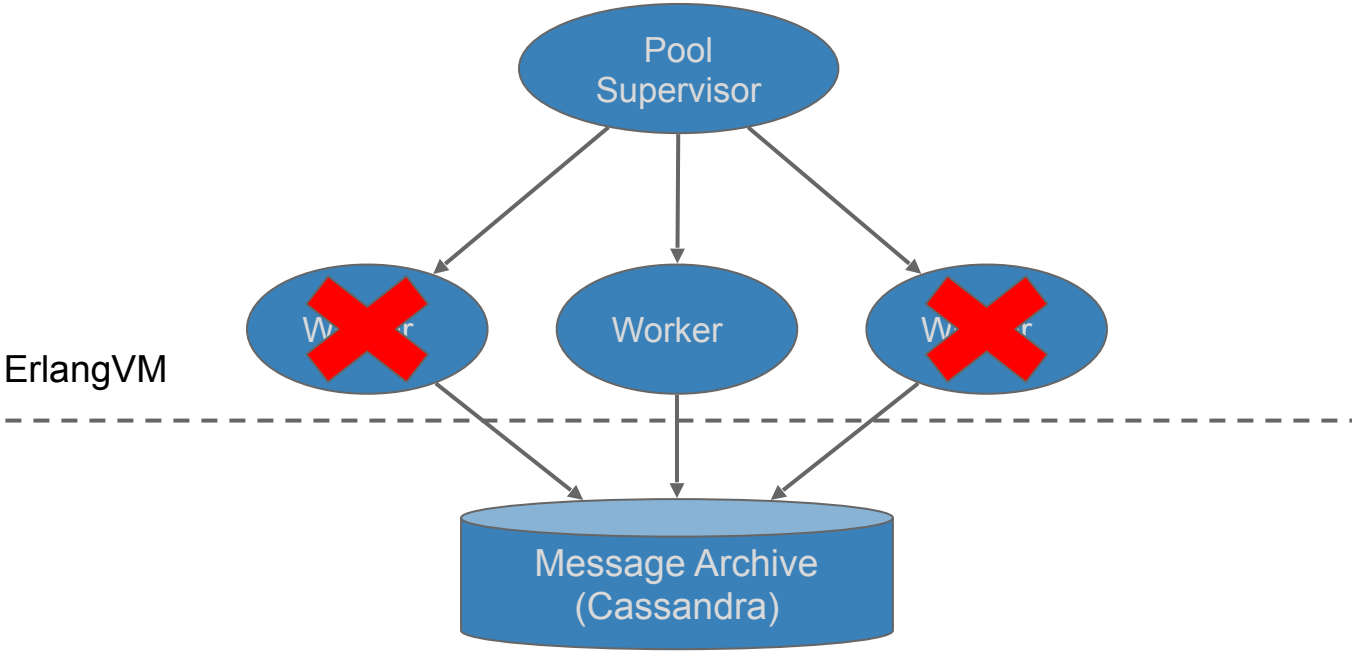
Applying ChE: Steady State





Hypothesis 1: Database Failure

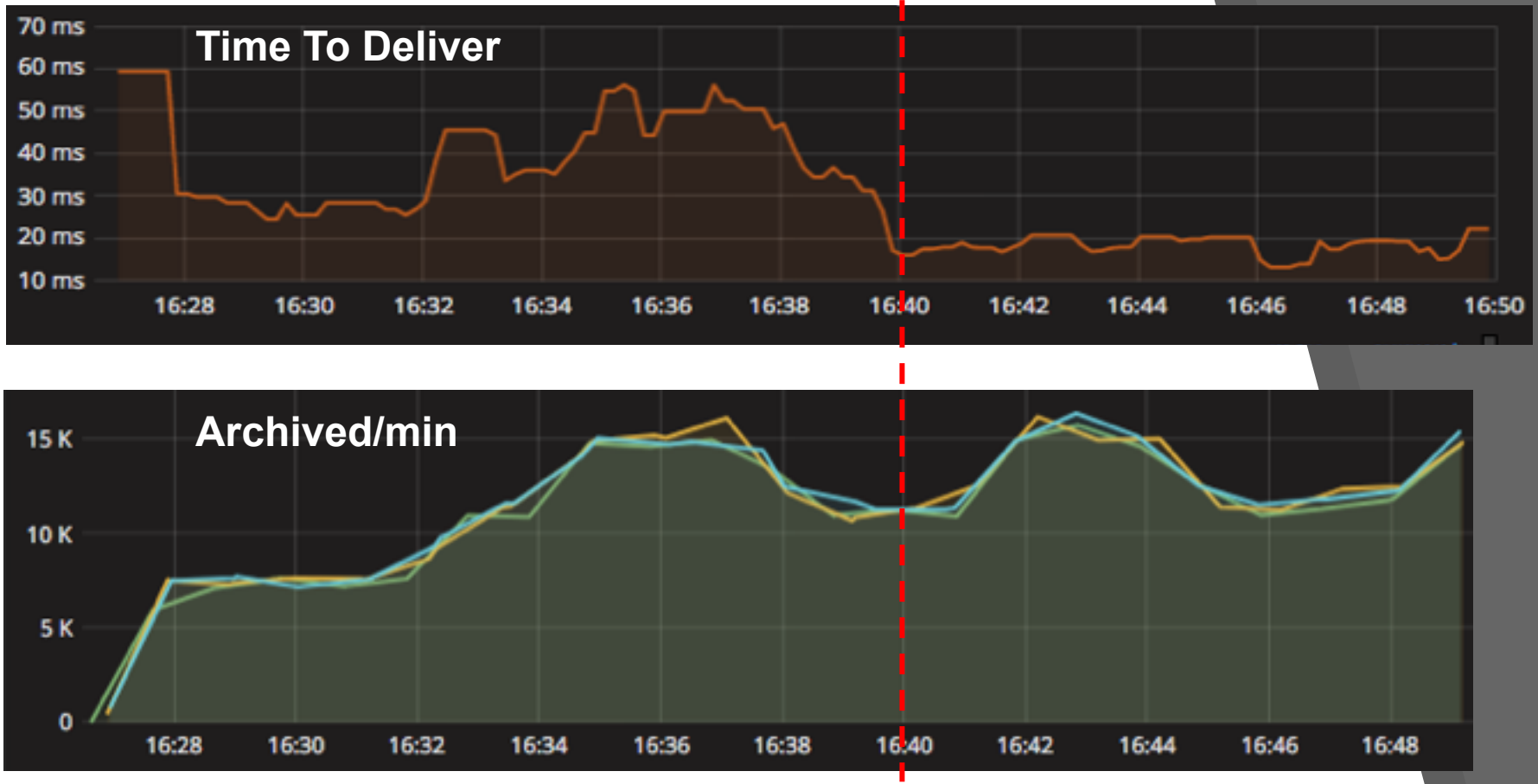
Failure to write to the database won't disrupt the service



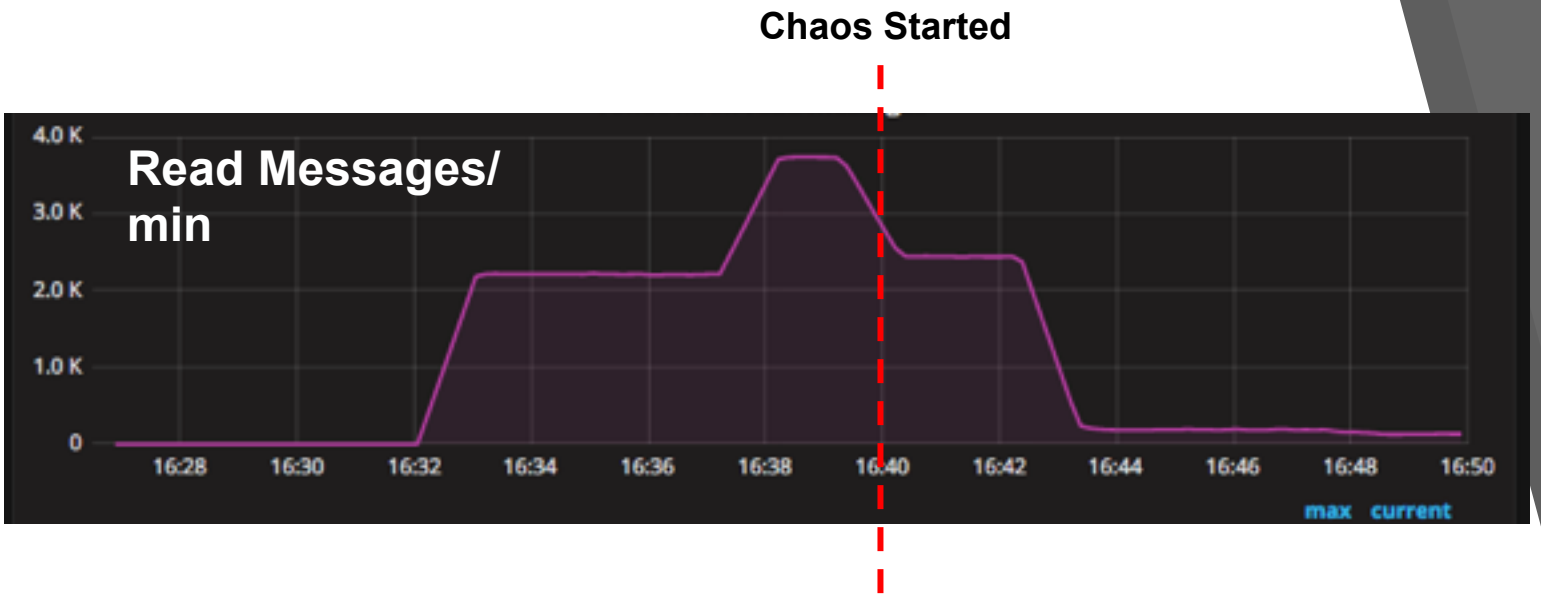
Hypothesis 1: Database Failure

```
# setup
kill_pool = fn ->
  :default
  |> MongooseCassandra.get_all_workers()
  |> Enum.random()
  |> Process.exit(:chaos)
end
break_cassandra = fn f ->
  kill_pool.()
  Process.sleep(15)
  f.(f)
end
# run
fault = spawn(fn -> break_cassandra.(break_cassandra) end)
# stop
Process.exit(fault, :stop)
```

Hypothesis 1: Database Failure

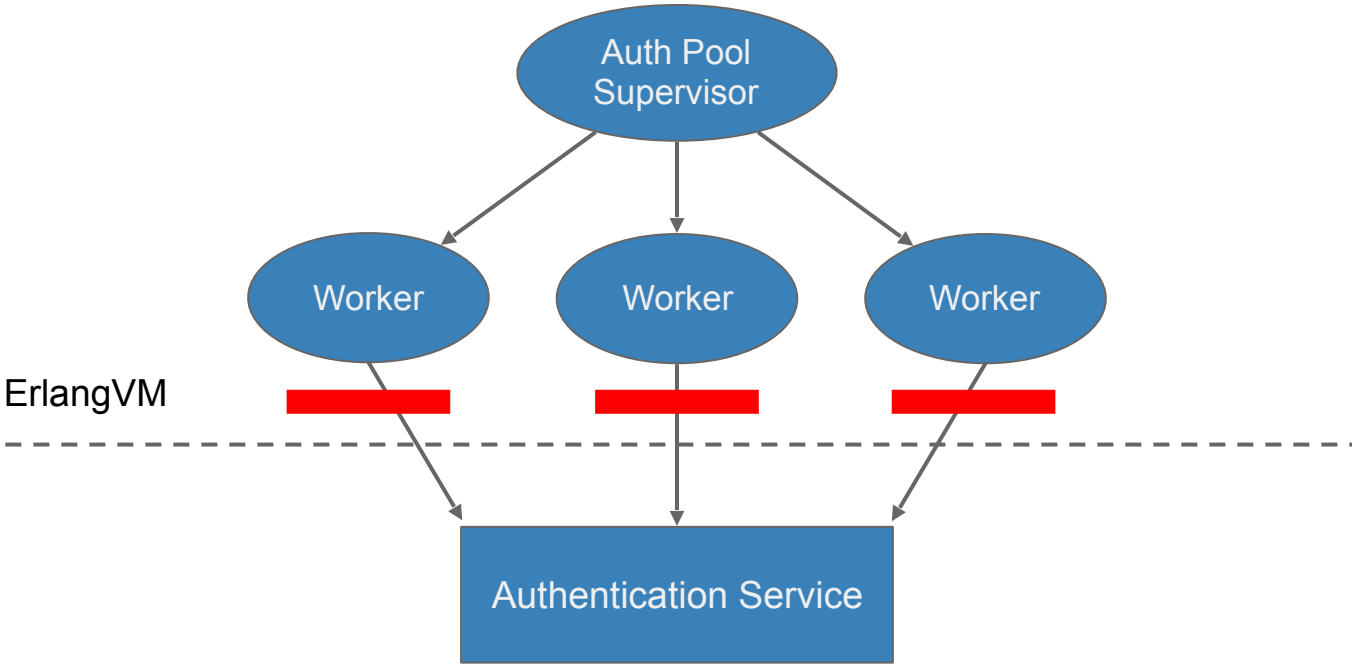


Hypothesis 1: Database Failure



Hypothesis 2: Slow Network

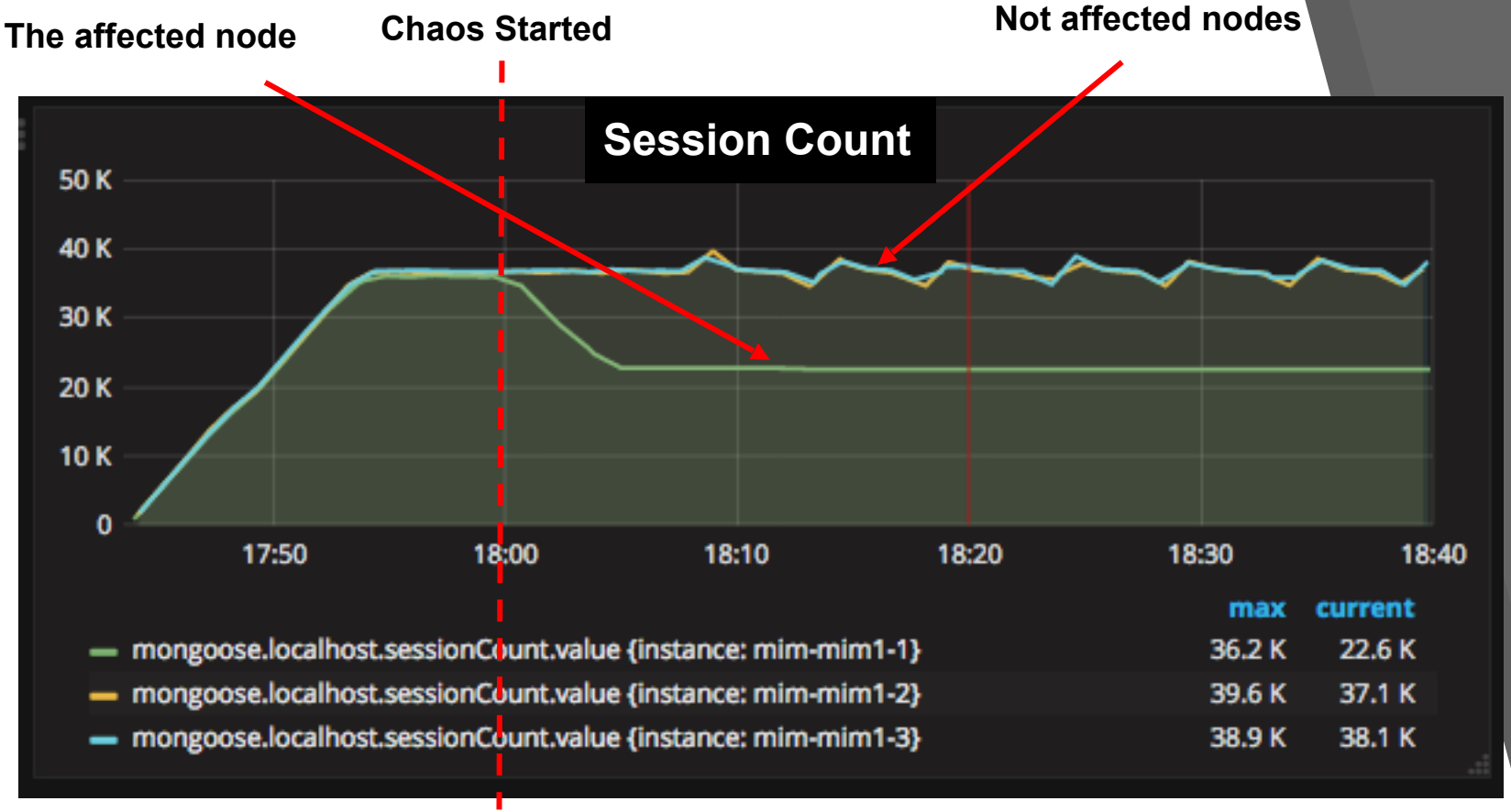
Delay on the connection to the Authentication Service won't prevent users from logging in



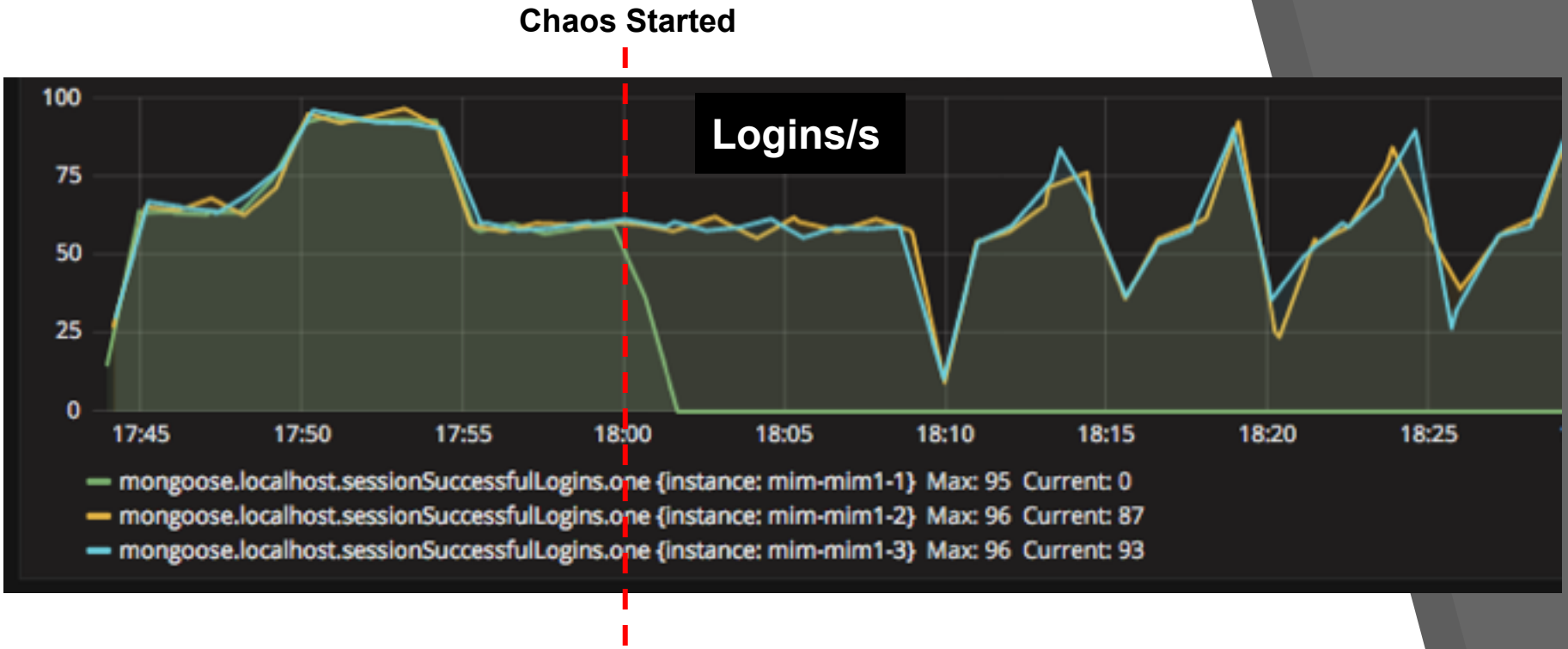
Hypothesis 2: Slow Network

```
# setup
delay = fn user, pass ->
  Process.sleep(100)
  :meck.passthrough([user, pass])
end
delay_auth = fn ->
  :ok = :meck.new(MongooseAuth, [:passthrough])
  :ok = :meck.expect(MongooseAuth, :authenticate,
    fn user, pass -> delay.(user, pass) end)
  Process.sleep(:infinity)
end
# run
fault = spawn(fn -> delay_auth.() end)
# stop
Process.exit(fault, :stop)
```

Hypothesis 2: Slow Network

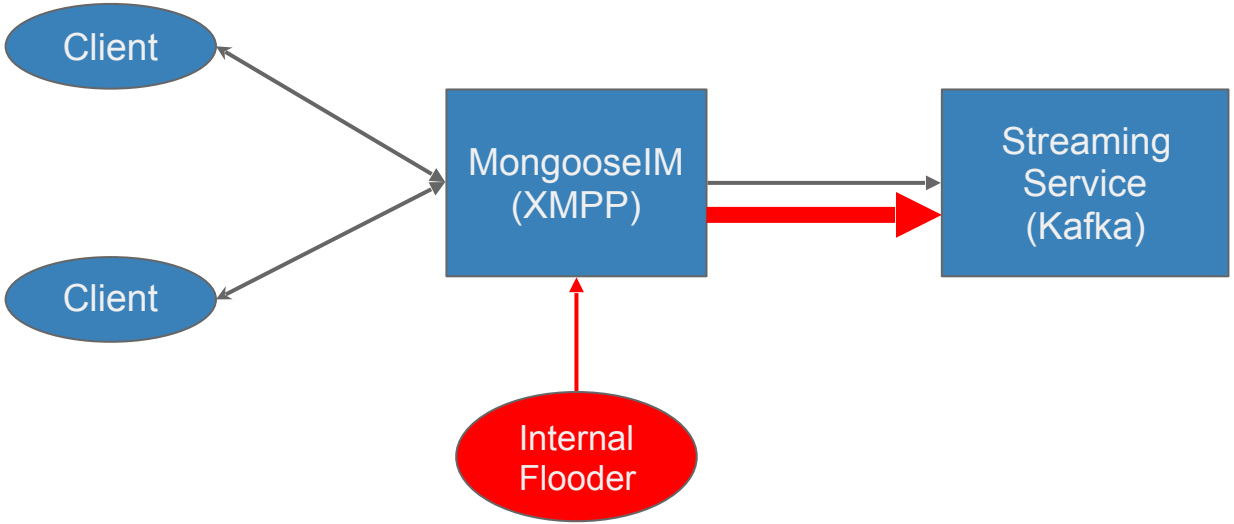


Hypothesis 2: Slow Network



Hypothesis 3: Message Rate Spike

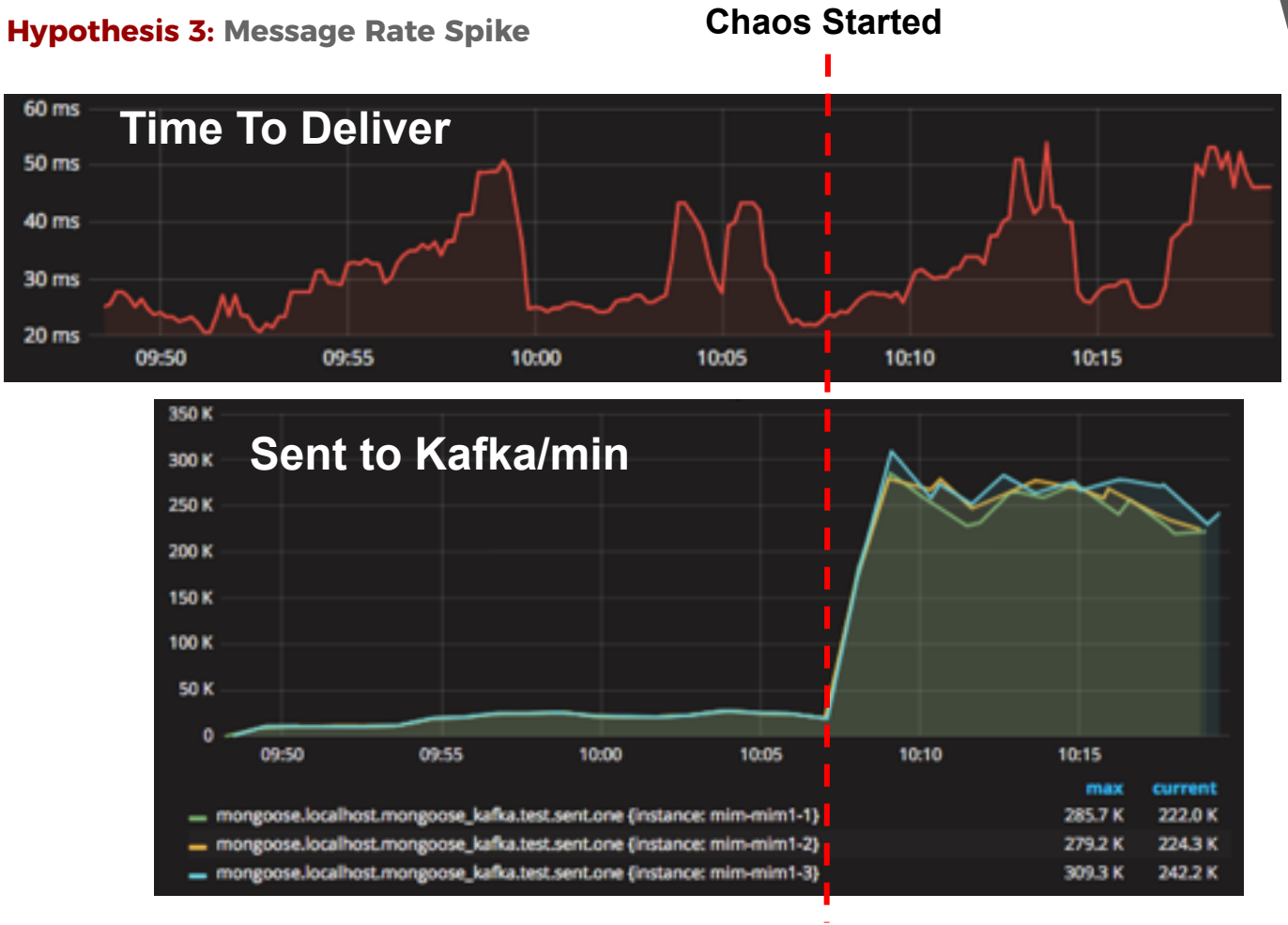
Spikes in message rate sent to Kafka won't disrupt the service



Hypothesis 3: Message Rate Spike

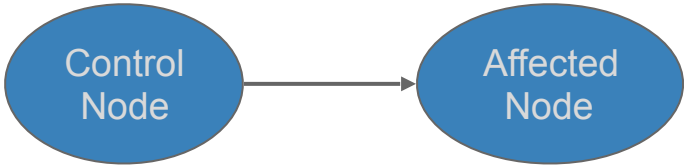
```
# setup
produce = fn ->
  msg = :crypto.strong_rand_bytes(100)
  MongooseKafka.produce("host", "topic", msg)
end
flood = fn f ->
  for _ <- 1..50_000, do: produce.()
  Process.sleep(1000 * :rand.uniform(5))
  f.(f)
end
# run
fault = spawn(fn -> flood.(flood) end)
# stop
Process.exit(fault, :stop)
```

Hypothesis 3: Message Rate Spike



Fault Injection: no recompilation

Direct or Remote



- RPC

```
:rpc.call(:aff_node@localhost, PoolSup, :which_children, [])
```

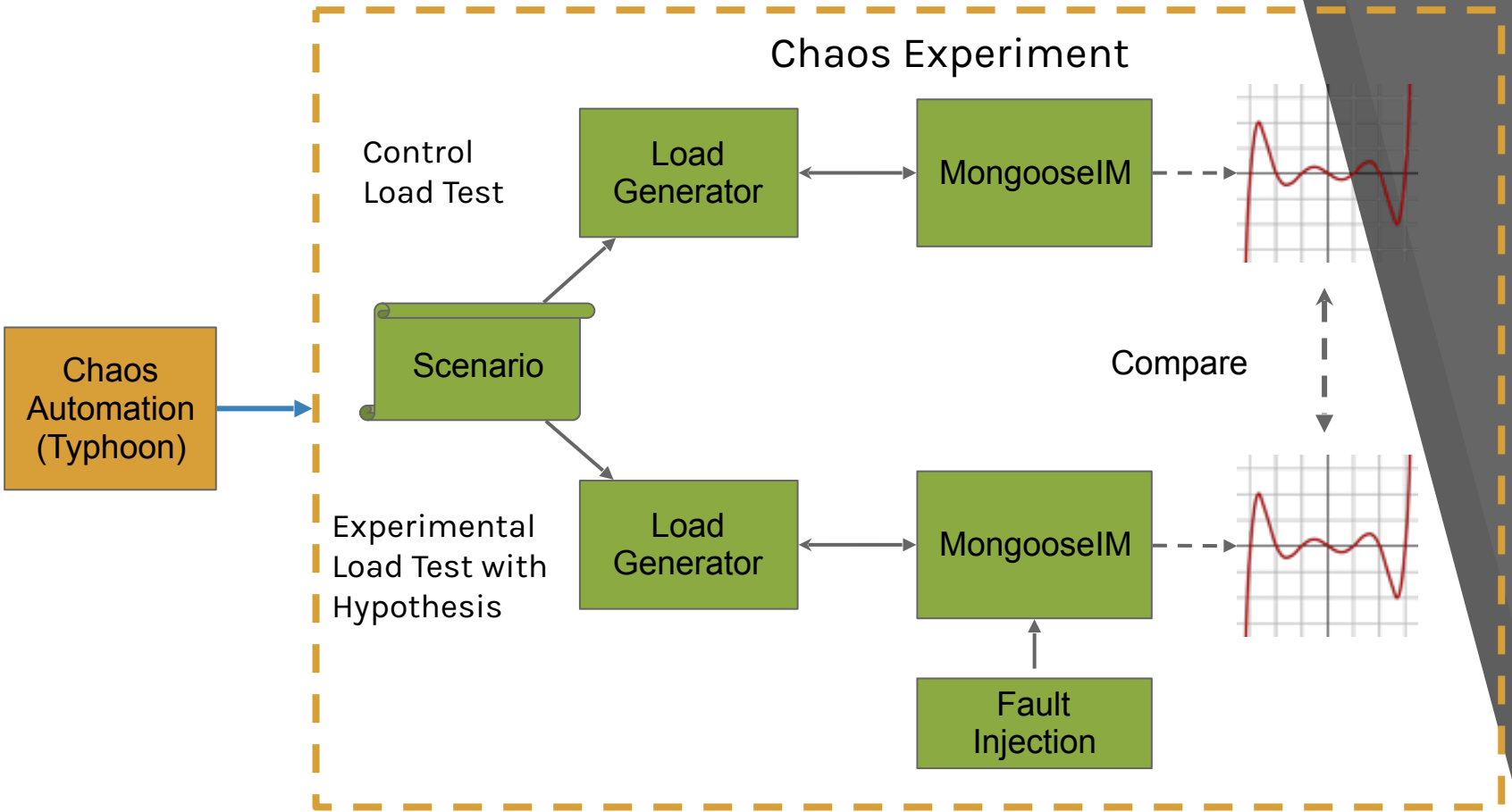
- Remote processes

```
Node.spawn(:aff_node, fn -> … end)
```

3.

MongooseIM Chaos Engineering Automation

Automation of Chaos Engineering



Typhoon: Elixir Application

```
apps
├── typhoon
│   └── lib
│       └── typhoon
│           ├── chaos_manager
│           ├── fault_injection
│           └── infrastructure
└── typhoon_web
```

Typhoon: Elixir Application

Infrastructure

```
%TestSetup{}
%TestTopology{}
```

Fault Injection

```
Fault Protocol
%MyFault{}
```

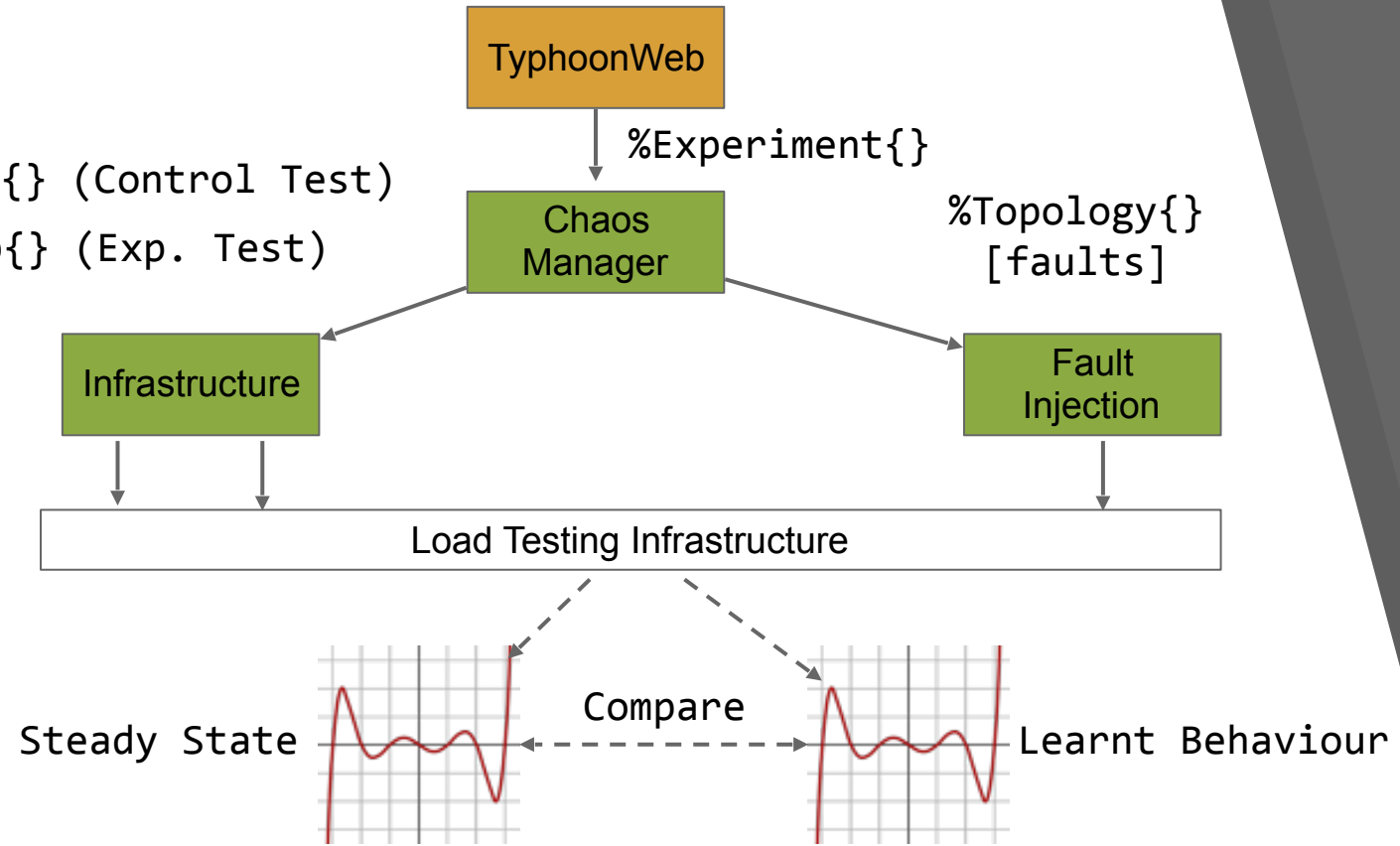
Chaos Manager

```
%Experiment{
    :control_test_id,
    :experimental_test_id,
    :setup,
    :faults,
    :faults_offsets,
}
```

Typhoon: Elixir Application

```
Load Test Config: …
Faults: …
```

1. %Setup{} (Control Test)
2. %Setup{} (Exp. Test)



Typhoon: Fault Injection

```
defprotocol FaultInjection.Fault do
  @doc "Applies the fault to the load test run by `test_id`"
  @spec apply(struct(), test_id()) :: :ok | {:error, term()}
  def apply(fault, test_id)
end
```

Typhoon: Fault Injection

```
defmodule FaultInjection.Fault.MyFault do
  embedded_schema do
    field(:param1, :integer)
    field(:param2, :string)
  end

  def changeset(struct, attrs) do
    struct
    |> cast(attrs, [:param1, :param2])
    |> validate_required([:param1, :param2])
  end

  defimpl FaultInjection.Fault do
    def apply(fault, test_id), do: send self, %{fault: MyFault}
  end
end
```

CHAOS ENGINEERING

is for everyone - go and explore it

APPLY

it to your system using the most basic techniques available

AUTOMATE

if it works for you add it to your continuous integration pipeline

THANK YOU!

szymon.mentel@erlang-solutions.com
@szymonmetel
github.com/mentels
medium.com/@mentels