Building a Scalable Messaging Fabric with JRuby and Storm

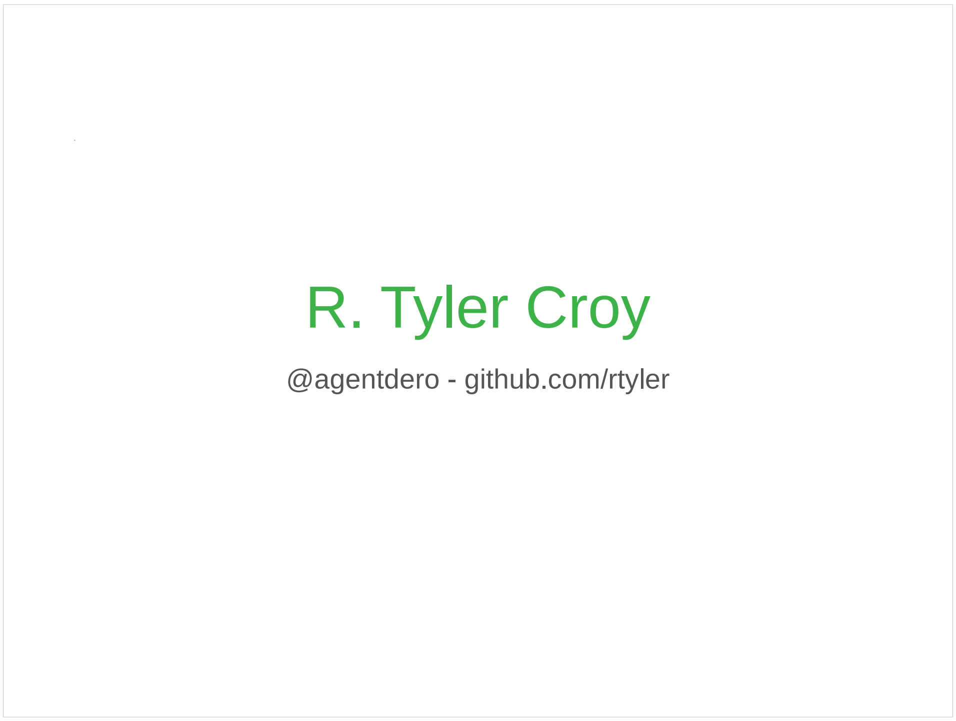
R. Tyler Croy - Ian Smith

Lookout, Inc.

August 1, 2014

Your hosts

Jump into who we are



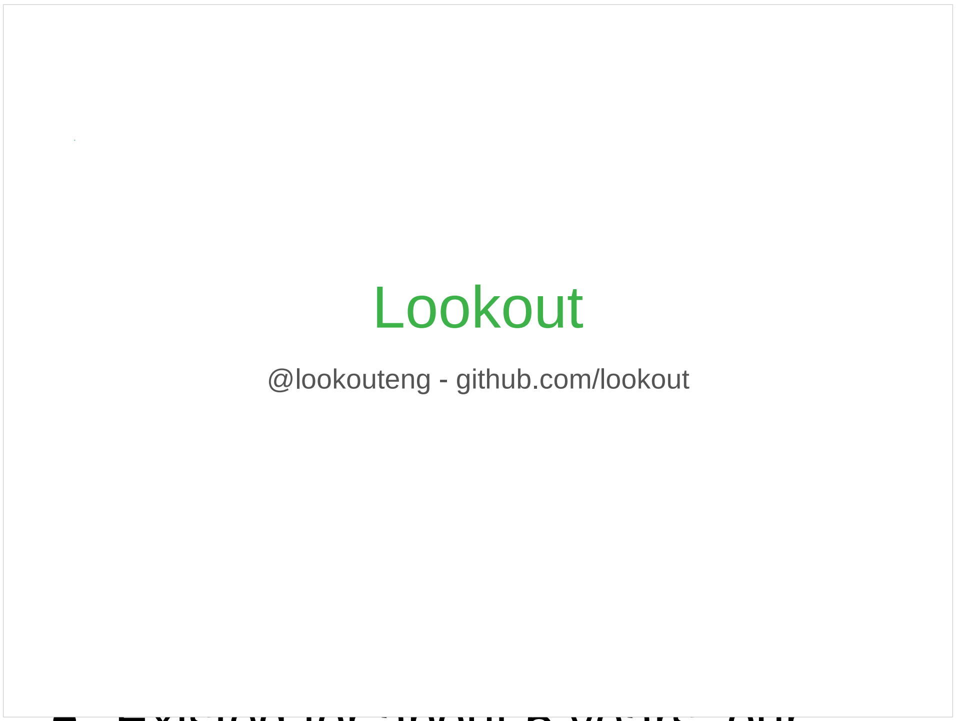
R. Tyler Croy

@agentdero - github.com/rtyler

- Ruby developer for 3.5 years, 4 years of Python before that
- Background in building scalable distributed web services

Ian Smith

@metaforgotten - github.com/ismith

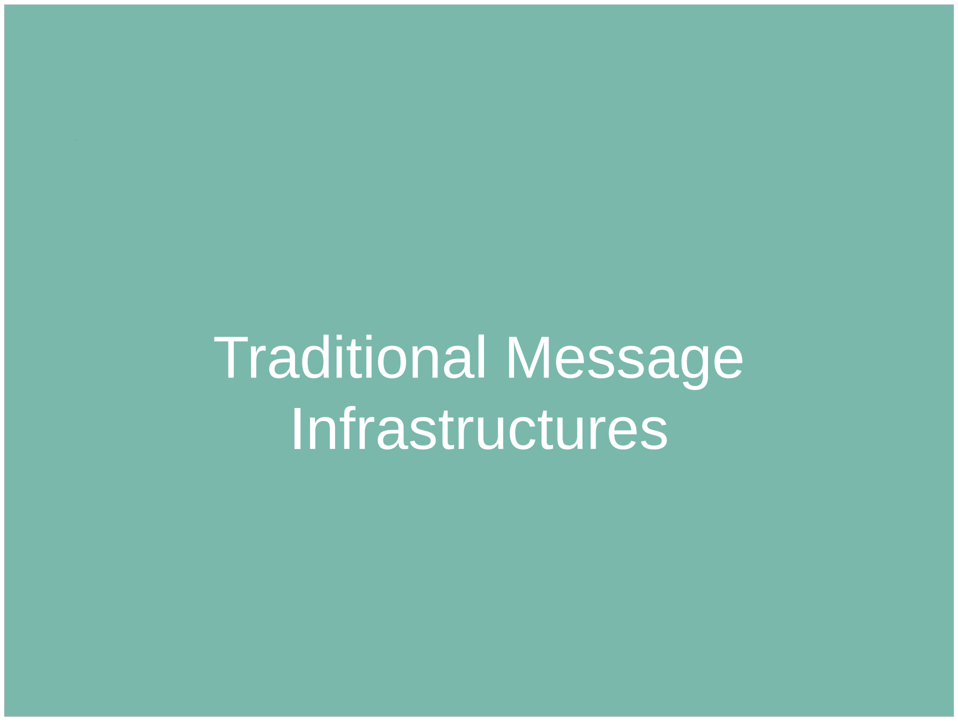

Lookout

@lookouteng - github.com/lookout

- Existed for about 5 years, our product's model relies on lots of backend infrastructure to provide mobile security faster/better/strong
- We've made the inverse change many companies have made over the past five years. Started with good roots in mobile, had to make lots of mistakes on the web
- Building a service-oriented infrastructure across a couple different business units (which is hard).
- Messaging between services has become an important component of that SOA, which led us to Storm ->

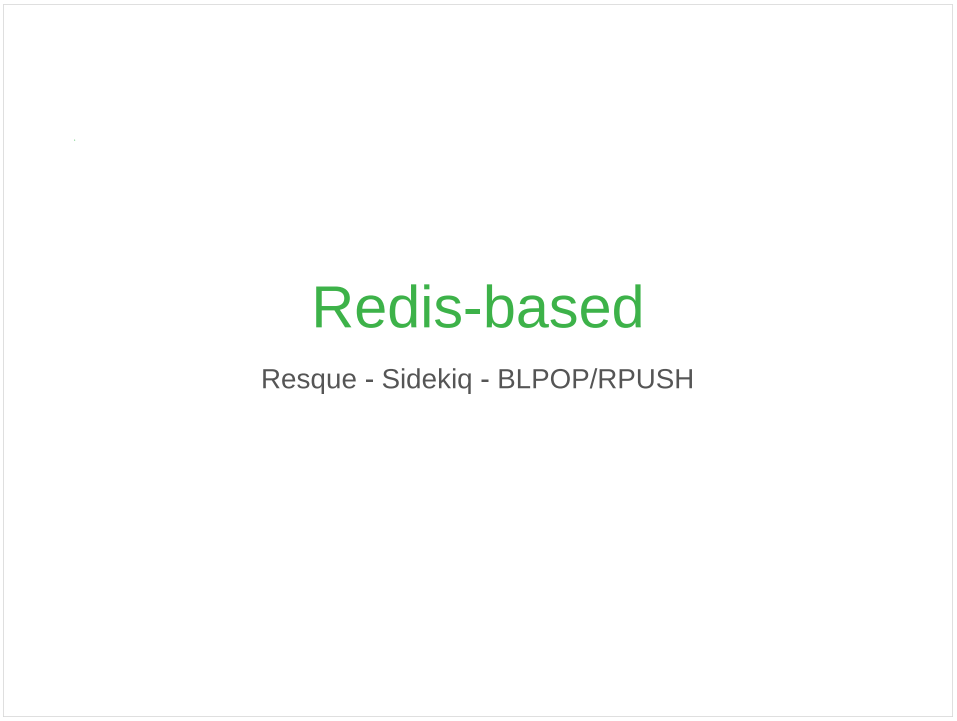
What/Why is Storm

- Storm is a message processing framework, but to really explain what or why we're talking about Storm, but first let’s talk about messaging



Traditional Message Infrastructures

First discuss the “old” way of messaging, i.e. shoveling messages to worker queues. Works for models where you’ve got a single known recipient of a message (e.g. deferred method call style invocations). “I need to send an email, I’m going to send a message to this queue to have something send an email for me so I don’t block this request”

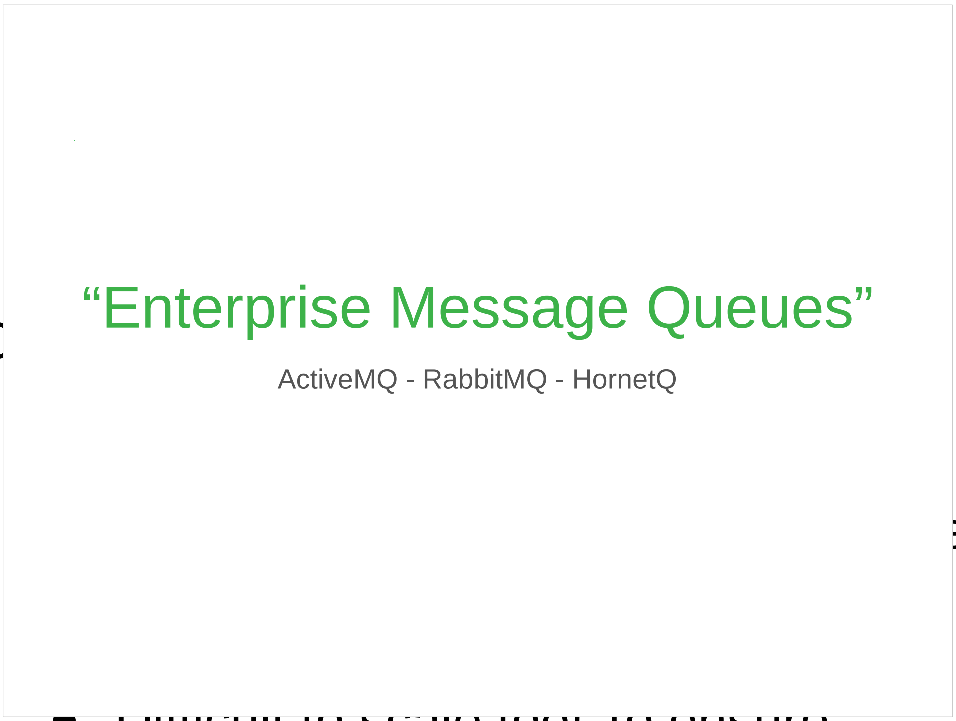


Redis-based

Resque - Sidekiq - BLPOP/RPUSH

Discuss the problems with scaling redis-based solutions.

- Hard to get good fault tolerance with Redis. Not distributed/clustered in a meaningful way. (Not investigated Redis Sentinel though)
- Hard to scale redis to varying kinds of workload (it’s a single CPU loop people)



“Enterprise Message Queues”

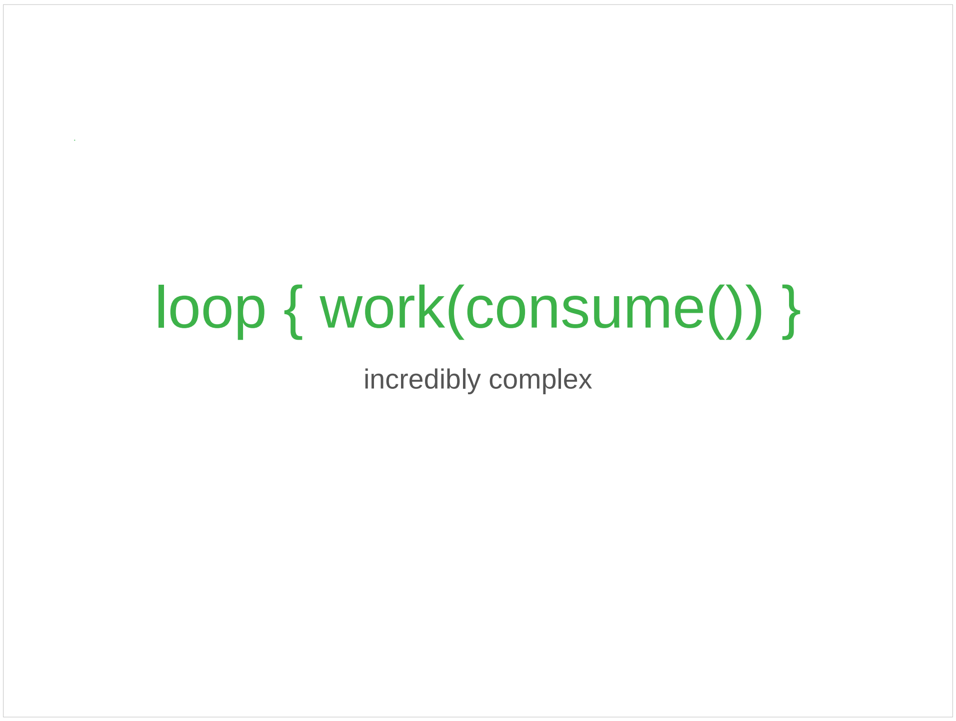
ActiveMQ - RabbitMQ - HornetQ

C

e

- Difficult to scale too! To ensure message delivery we would persist messages to MySQL using the built in MySQL backend. This bolted ActiveMQ’s scalability to that of MySQL (read: shitty application level sharding)
- Originally used STOMP for messaging protocol but ran into quirks with ActiveMQ’s support. Switching clients to JMS offered better performance however. Meant we were increasingly reliant on the JVM
- Using advanced features such as “virtual topics” led to many issues, but gave us the functionality we

Traditional Workers

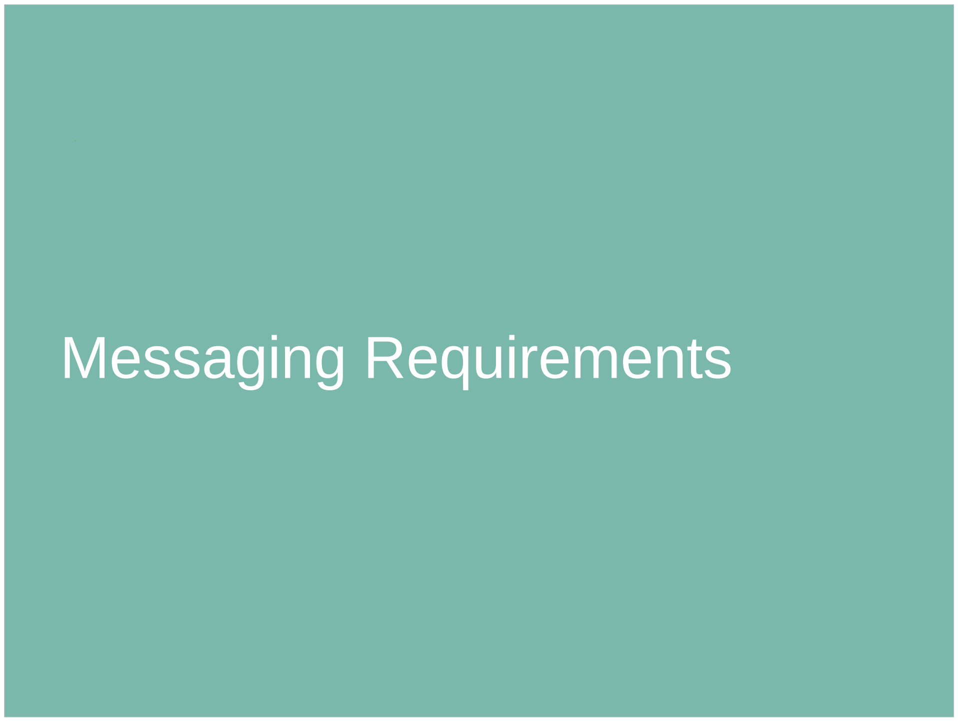


loop { work(consume()) }

incredibly complex

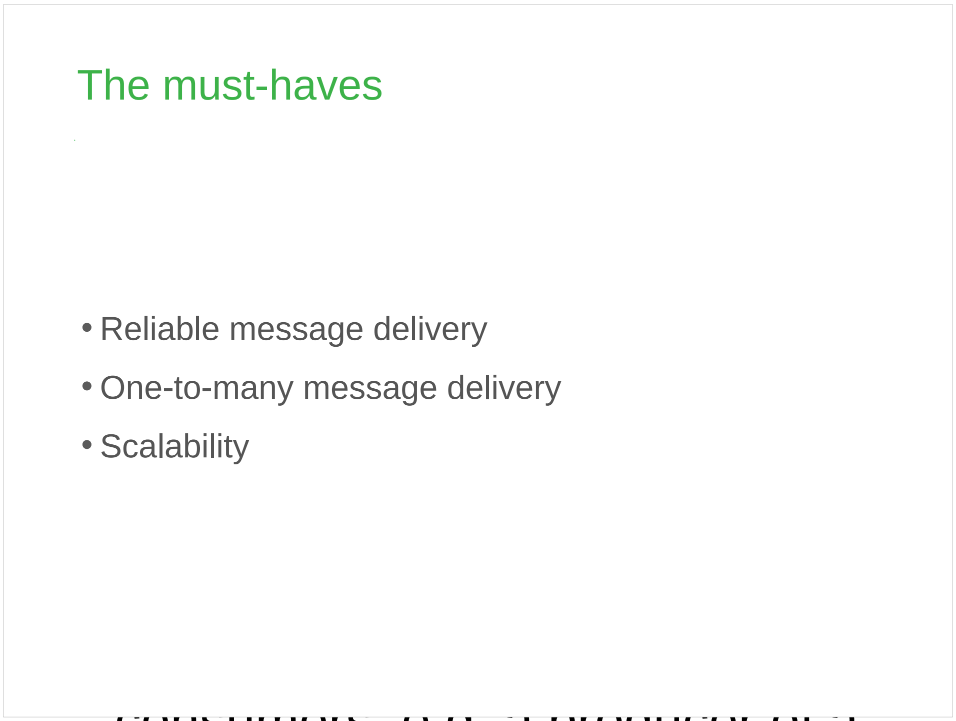
Need to go into detail on what most of our old style workers looked like, see: dumb ruby daemons

- Mostly simple daemons that would consume a single message per thread
- Mostly manual workload management, not really any auto scaling of consumers



Messaging Requirements

- After building all these other services we started to get a picture of what our real messaging requirements are



The must-haves

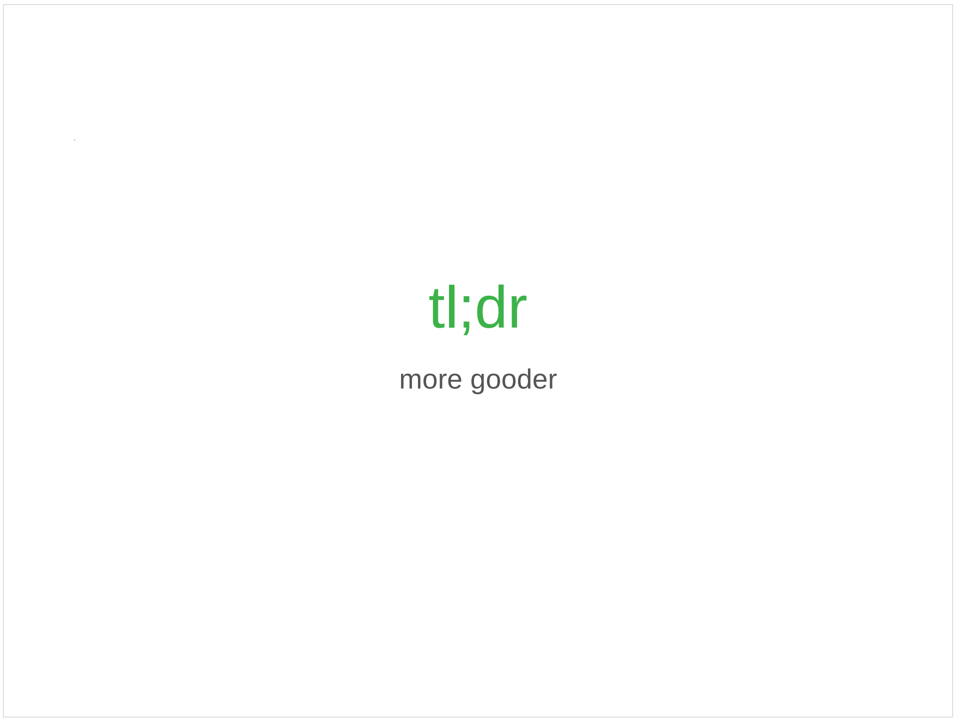
- Reliable message delivery
- One-to-many message delivery
- Scalability

consumers, e.g. a producer of a message need not know who will be consuming the message

- One-to-many delivery: a single message may need to be reliably consumed by multiple applications/consumers. Describe detection events and the consumers that might be interested in that (Security, Data, Consumer, etc)
- Having one-to-many allows newer consuming applications to be deployed without changing the producers (good!)
- Scalable/clusterable such that many services may use the messaging bus together

Kafka

This led us to Kafka

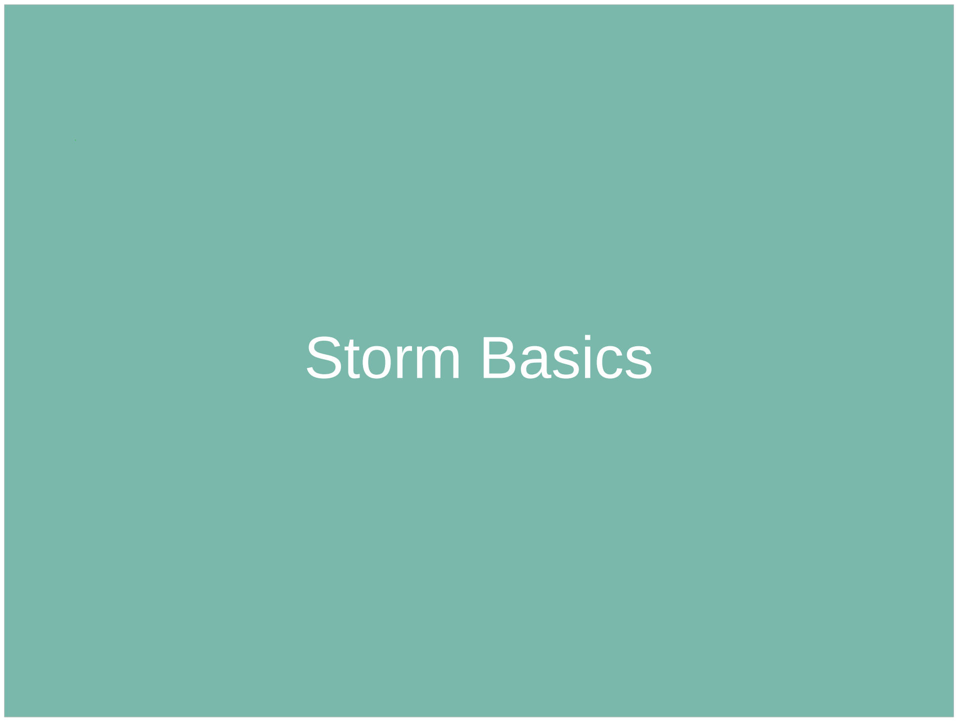


tl;dr

more gooder

Important features we need:

- Clustered
- Supports “consumer groups” which give us functionality similar to ActiveMQ’s “virtual topics” for managing different kinds of message consumers
- Designed for scalability with it’s log offset design



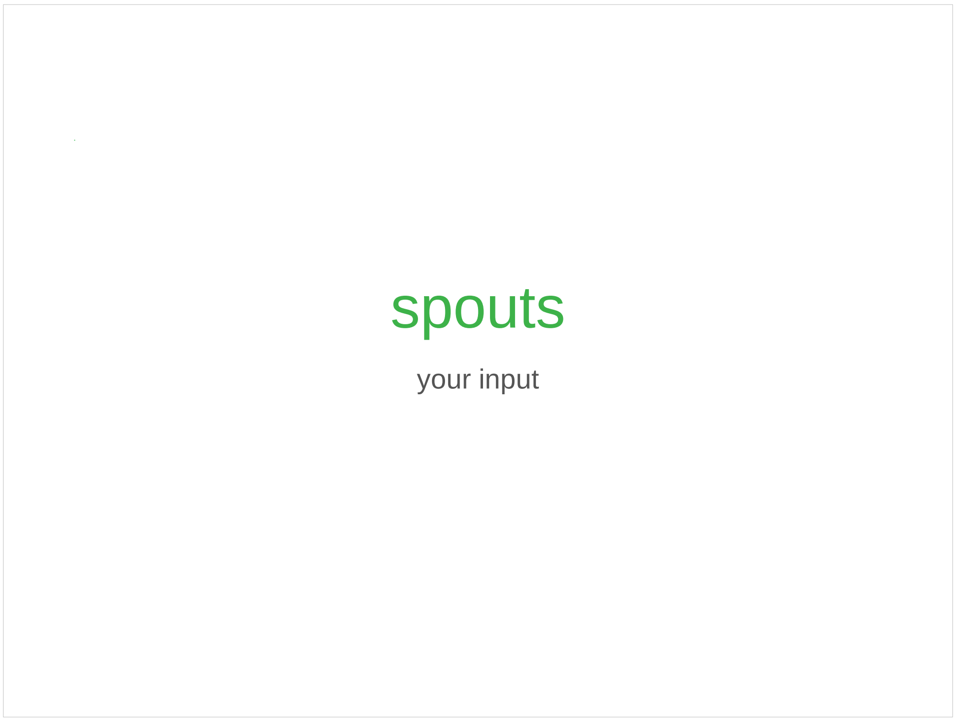
Storm Basics

There are some important terms and concepts that Storm introduces. It does not have many analogous tools that most developers might be familiar with.

tuples

the currency of Storm

All data is passed around in storm as a tuple, nothing special, just the basic data type for messages passing through Storm



spouts

your input

The way data gets into Storm is via spouts. This is some bit of code that feeds a stream of tuples in. These can be Kafka spouts, Redis pub/sub spouts or even just a spout which generates random streams of data

bolts

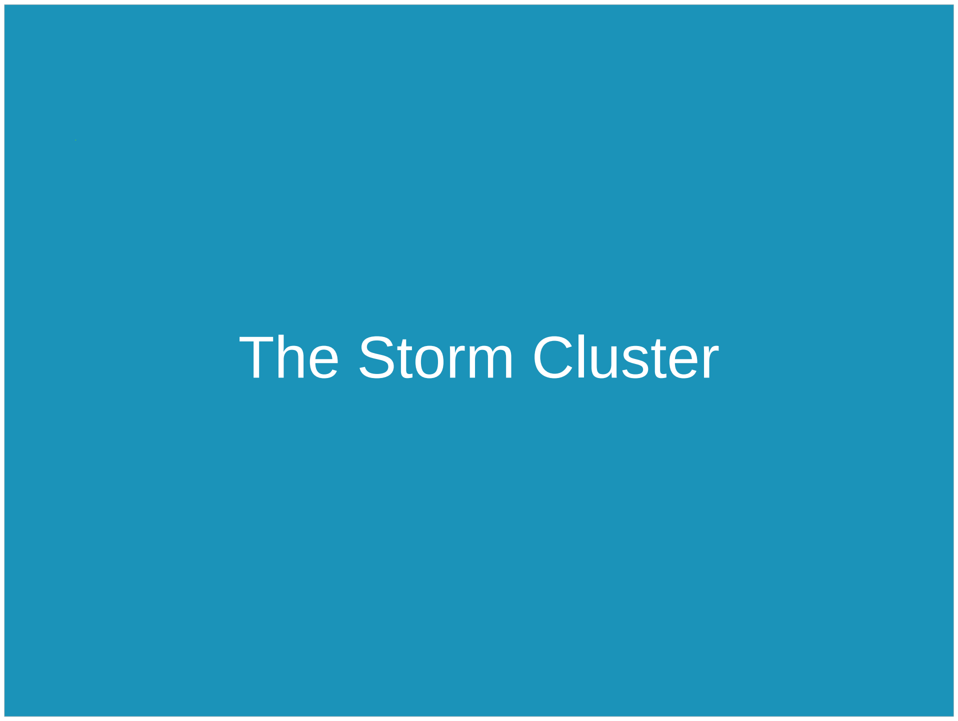
basic unit of operation

After a spout data is passed to a bolt, the worker unit within storm. These can be any kind of code whatsoever, they are passed a tuple of data at a time to work on by Storm

topology

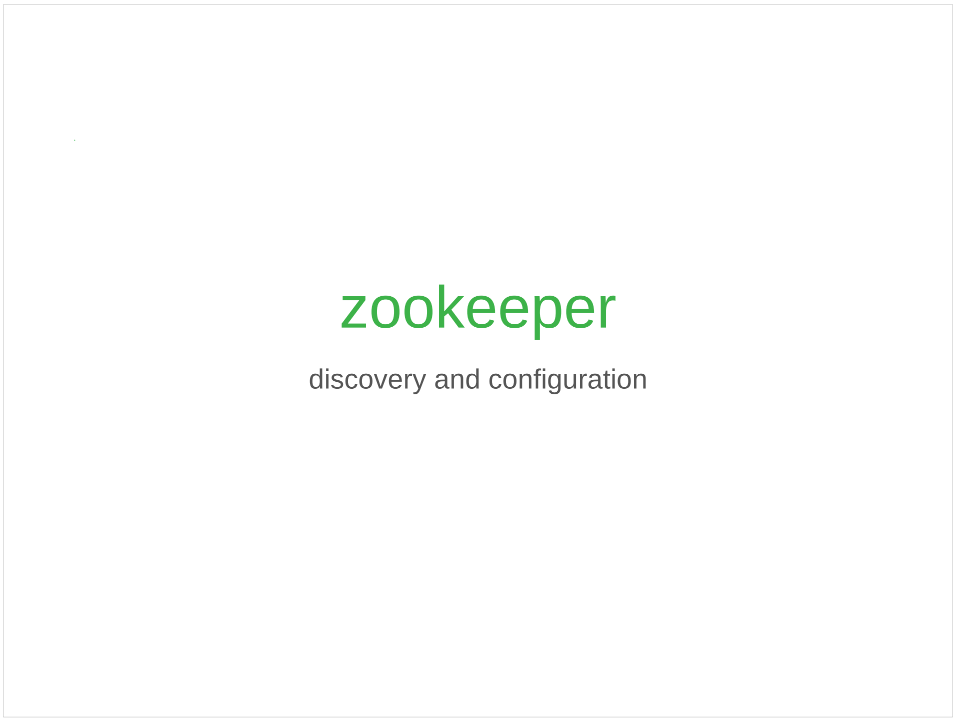
a directed graph of plumbing metaphors

All this comes together in a topology, a directed graoh of spouts and bolts. Discuss in this slide/section the basic topology deployment workflow: e.g. submitting a top, loading a top and then activating a top.



The Storm Cluster

Describing the Storm cluster itself is important as well. Storm is operated as a *cluster*, which contains a few different components



zookeeper

discovery and configuration

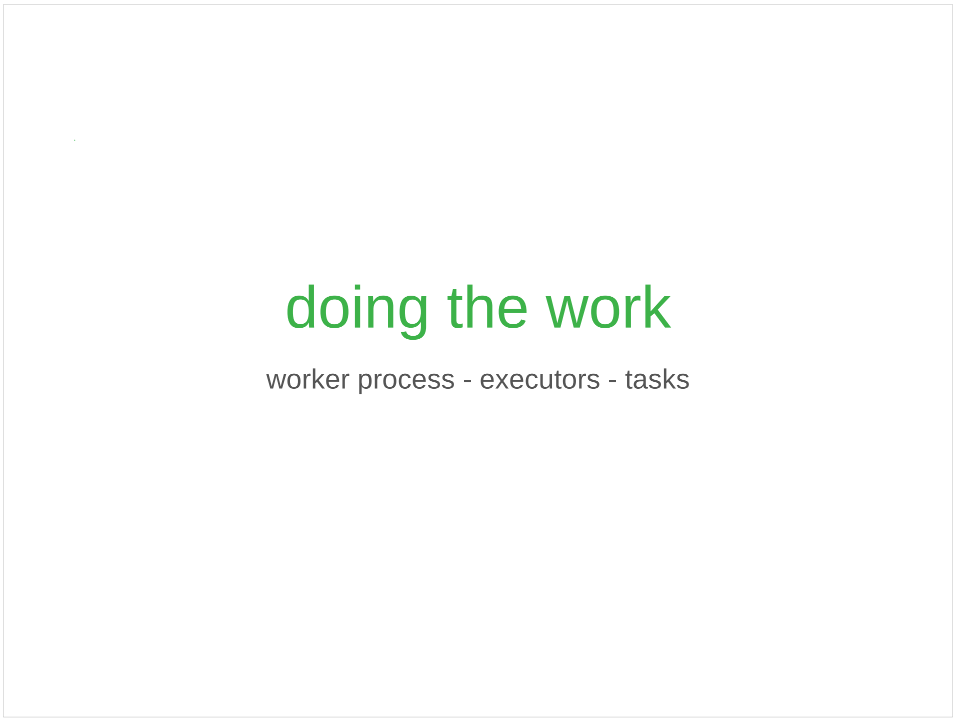
ZK is used for coordinating and discovering nodes

nimbus nodes

coordinate it

worker nodes

doing things with input



doing the work

worker process - executors - tasks

On a worker node work can be broken down into three buckets, there’s a worker process which contains executors (read: threads) and those executors will execute tasks (i.e. bits of a bolt or spout).

This model is important to keep in mind because it helps the Storm cluster distribute work across many nodes, even with varying workloads

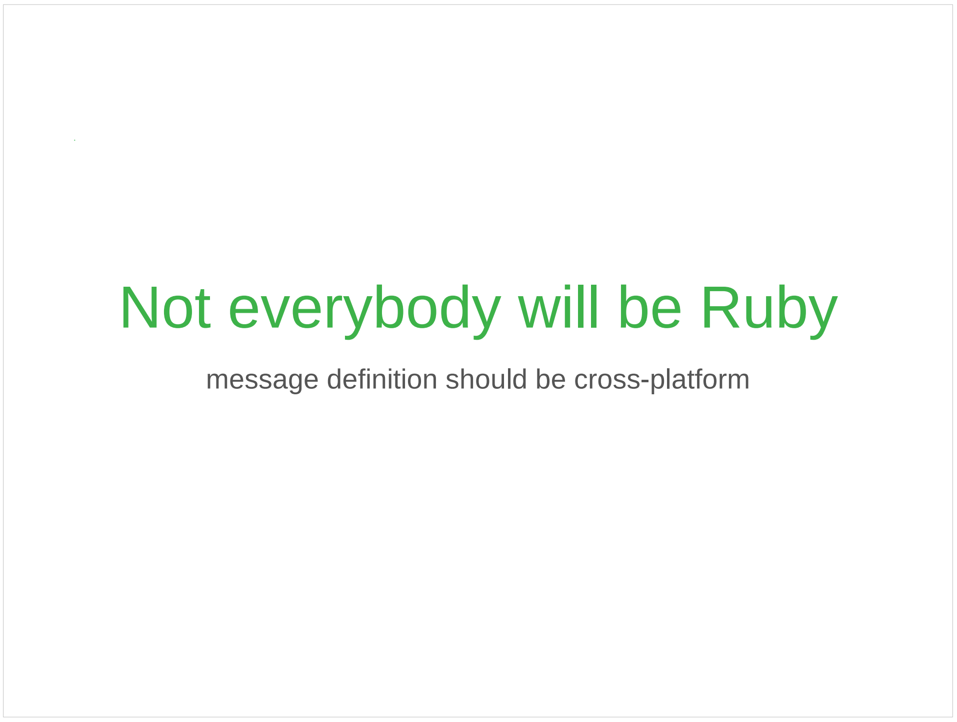
What/Why is Storm

- Storm is a message processing framework, with Kafka it gives us a foundation to build reliable and scalable messaging infrastructure

Message Design

With Kafka and Storm running and some ideas of how the pieces glue together, we must now discuss designing our messages. This is important because this is the API!

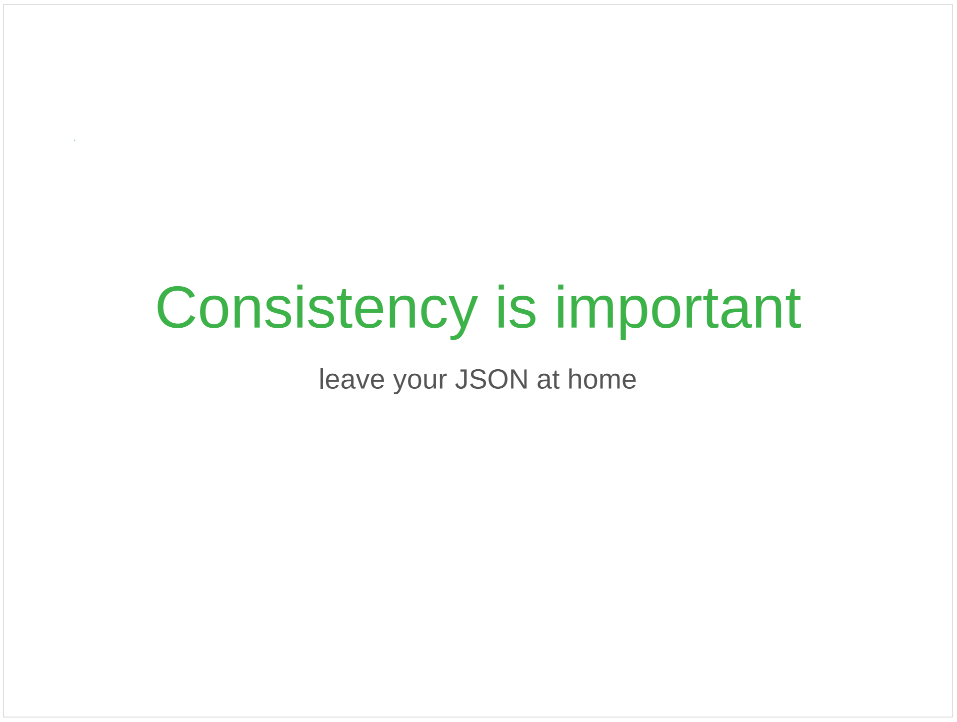
Everybody will likely choose different approaches for different reasons but in this section I want to explain the approach Lookout has taken



Not everybody will be Ruby

message definition should be cross-platform

Reality is a bummer! Unfortunately not all actors in our messaging fabric and likely not in yours either will be ruby based. No Marshal/dumping of objects! So what about JSON?



Consistency is important

leave your JSON at home

Message validation is important, using stock JSON or poorly formatted XML requires a lot more code to validate properly

Protocol Buffers	Thrift
Avro	Home-grown

A few options exist for defining messages consistently between platforms, We ended up selecting protocol buffers for its good

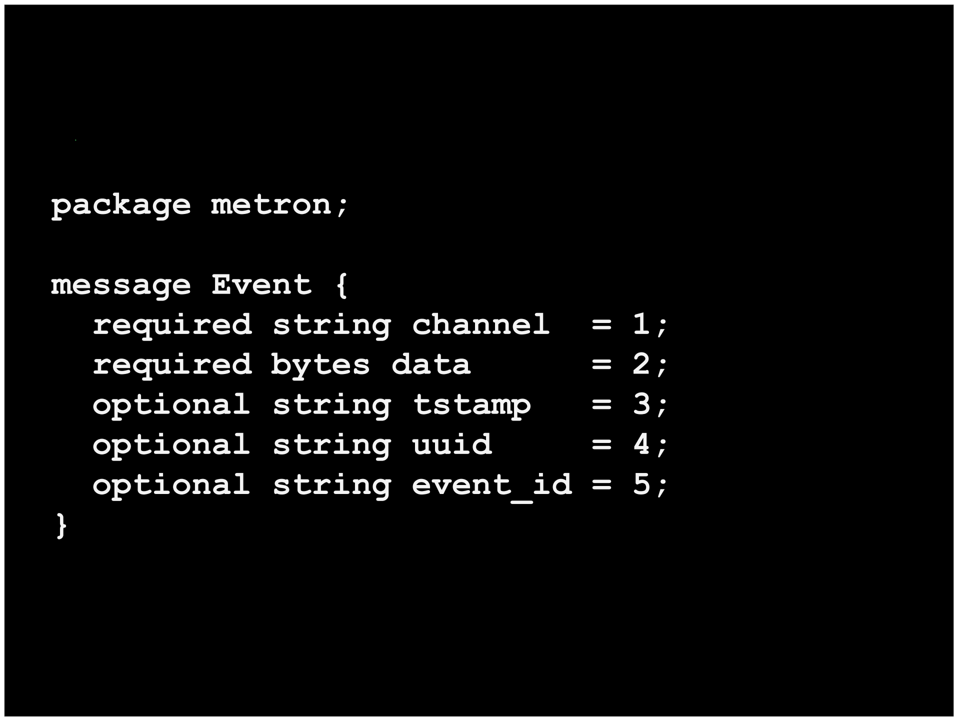
Working with everything else

Lots of our code runs in web applications, background processes and of course on mobile devices. Most of our interesting messages come straight from mobile devices

To help with this we built a tool called Metron

Metron

Metron is a simple simple Simple Sinatra app that authenticates devices and accepts a message payload from them. It also determines the original routing for that message into the messaging infrastructure

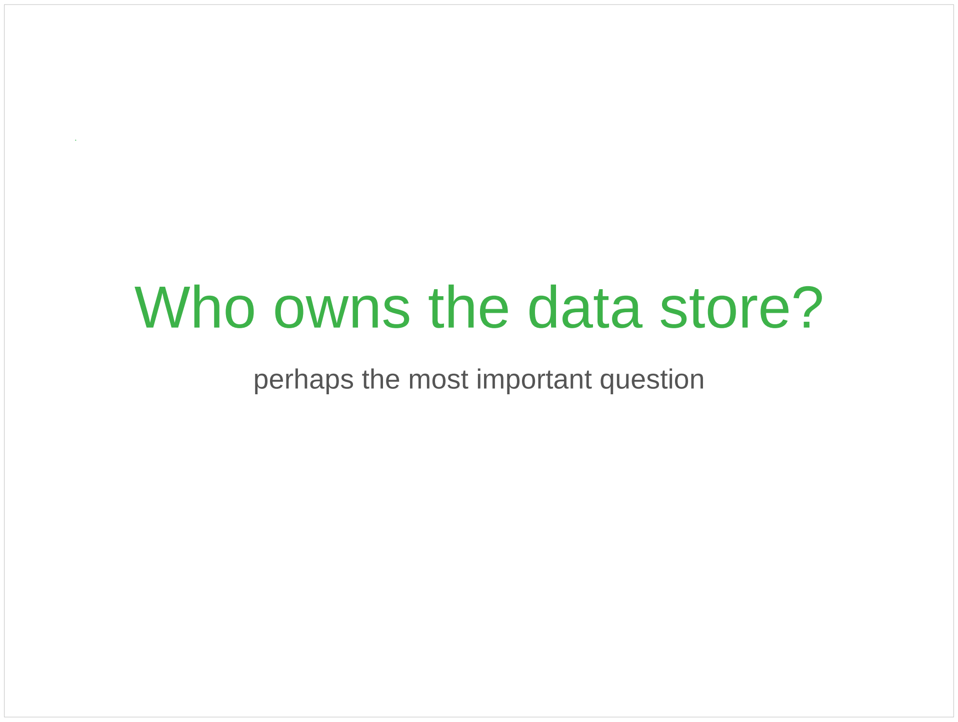


```
package metron;

message Event {
  required string channel  = 1;
  required bytes data      = 2;
  optional string tstamp   = 3;
  optional string uuid     = 4;
  optional string event_id = 5;
}
```

A paired down version of the message declaration in Protobuf syntax. It's important to note that the actual message is completely opaque to metron!

Storm and your applications

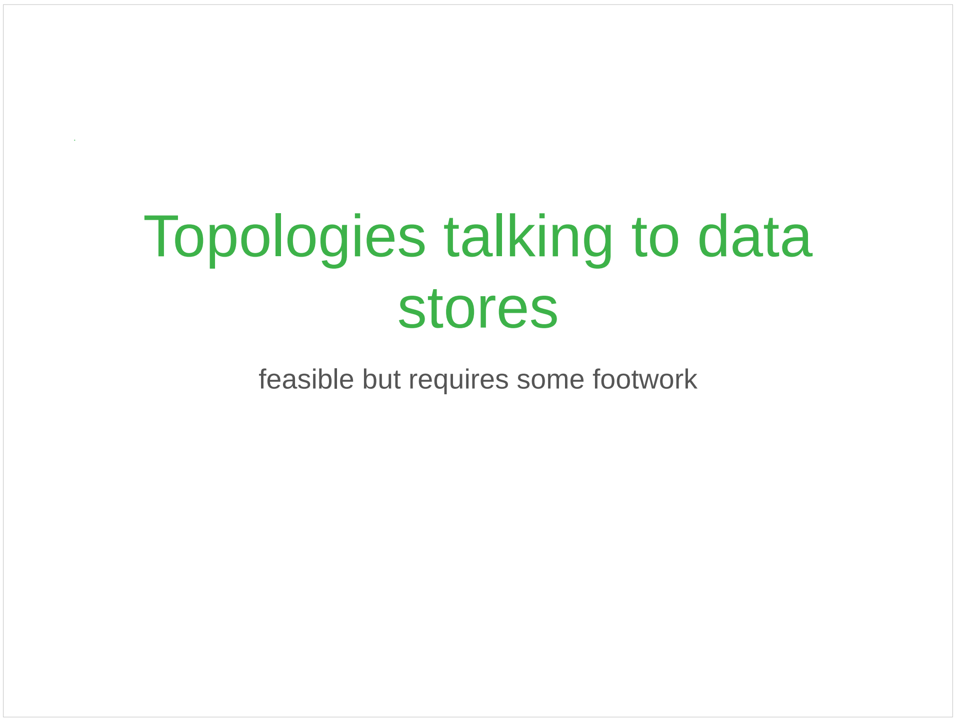
With Metron helping to get messages in, we then can glue Storm topologies and our other applications together. Most of our use-cases actually fit into a model of service-to-service messaging, relying on Kafka and Storm to drive composite behavior between services. In an SOA world, many services may work together to provide a singular product experience



Who owns the data store?

perhaps the most important question

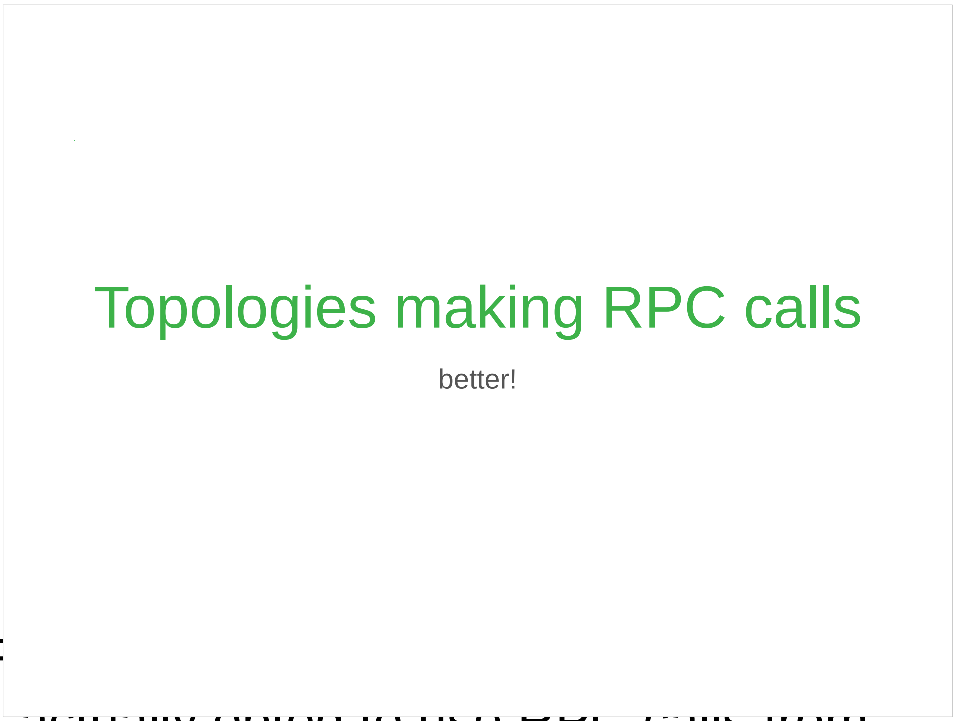
With functionality in your Storm cluster and some more traditional web service, whose responsibility is persisting data?



Topologies talking to data stores

feasible but requires some footwork

Topologies talking to data stores is doable. Access control for nodes in the storm cluster and exposing that information to the topology itself is a little tricky. Since Storm schedules the work across multiple worker nodes, you need to make sure all the information a topology might need is on every node.



Topologies making RPC calls

better!

F

actually opted to use RPC calls from topologies to applications.

- Building RPCs is better understood domain
- Far easier to dictate consistent access controls
- Many of our use cases might have a synchronous and asynchronous path. E.g. send one email versus send 1000
- Allows development teams loads of freedom when implementing an rpc call
- Need to discuss what work should stay in topologies versus moving into rpc, its a blurry line.

Phew

There's a lot to mentally fit into your head! The learning curve is steeper than I would like but once you orient your thinking around Storm's model, everything falls into place relatively quickly

Ian takes the rest

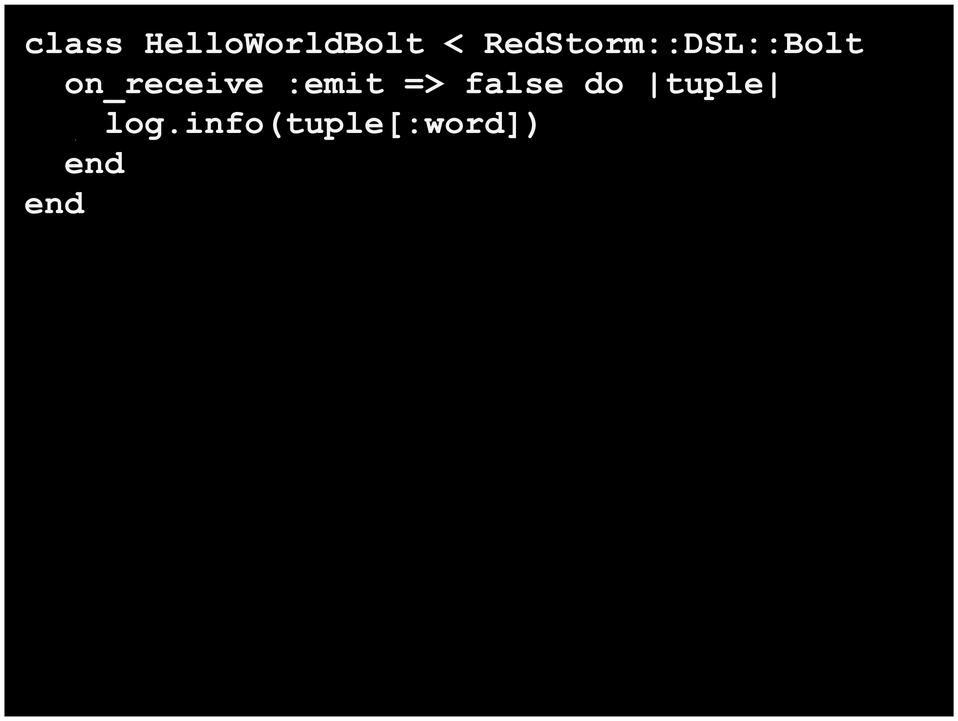
Developing Storm Topologies with JRuby

You could use ShellBolts - a bolt and a script in a non-JVM language (pipes & JSON)

But! Ruby *is* a JVM language, via JRuby

RedStorm
https://github.com/colinsurprenant/redstorm

Native, Trident & DSL



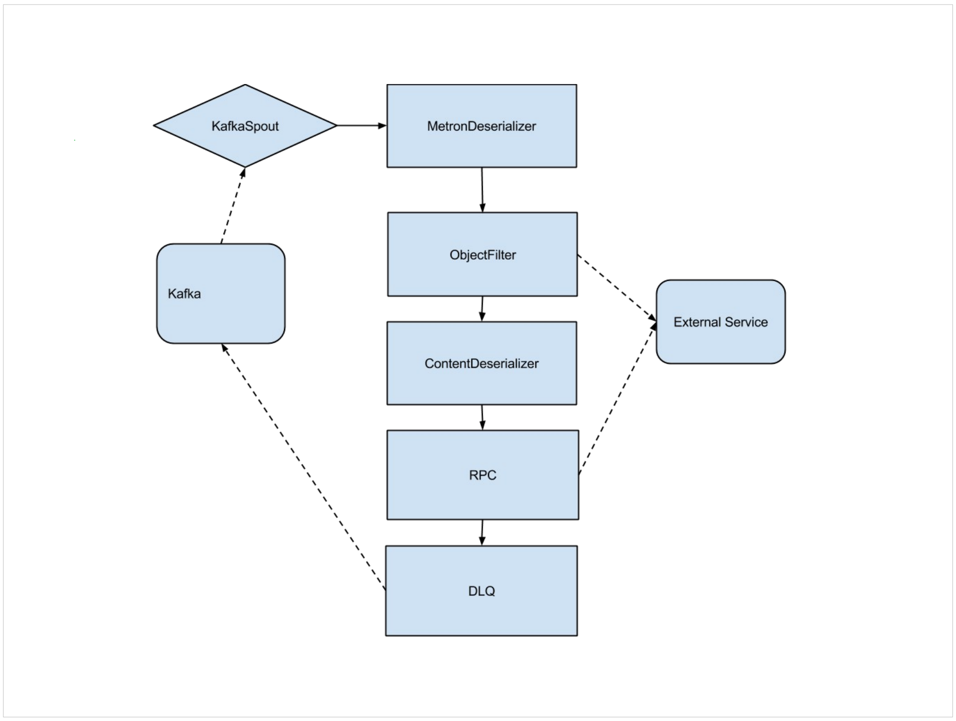
```
class HelloWorldBolt < RedStorm::DSL::Bolt
  on_receive :emit => false do |tuple|
    log.info(tuple[:word])
  end
end
```

Defining a new bolt or topology is pretty simple

```
class HelloWorldTopology
  spout HelloWorldSpout do
    output_fields :word
  end

  bolt HelloWorldBolt do
    source HelloWorldSpout, :shuffle
  end

  bolt AnotherBolt do
    source HelloWorldBolt, :shuffle
  end
end
```

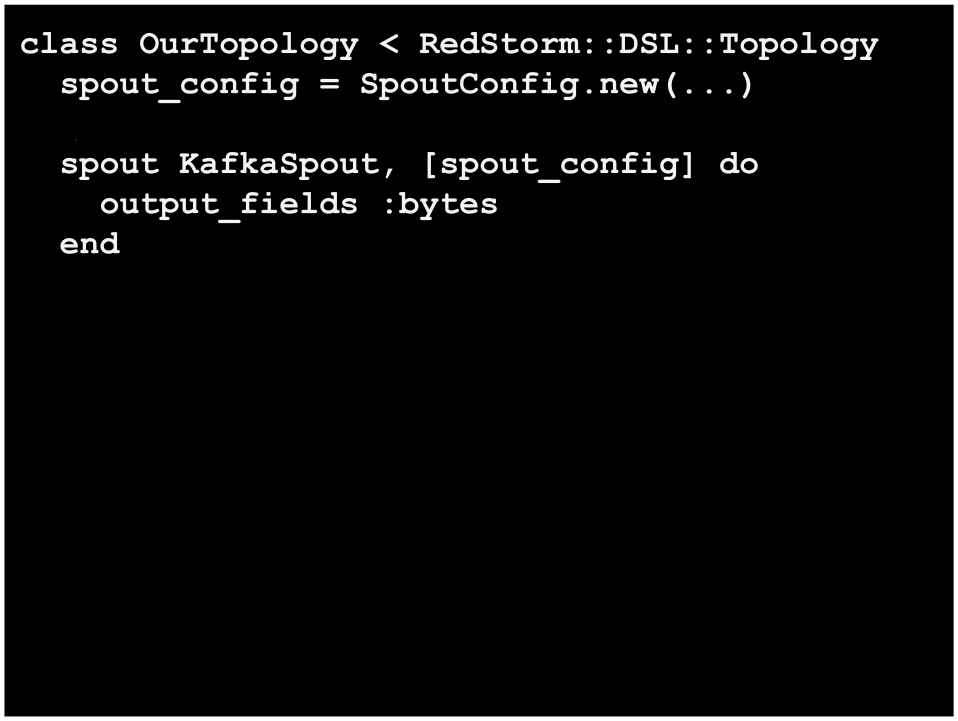
Defining a new bolt or topology is pretty simple

Topology Design


KafkaSpout
MetronDeserializer
ObjectFilter
Kafka
External Service
ContentDeserializer
RPC
DLQ

```
class OurTopology < RedStorm::DSL::Topology
  def self.topology_name
    “#{self.name}_#{commit_hash}”
  end
```

Define the topology class, with the commit hash in the name to disambiguate

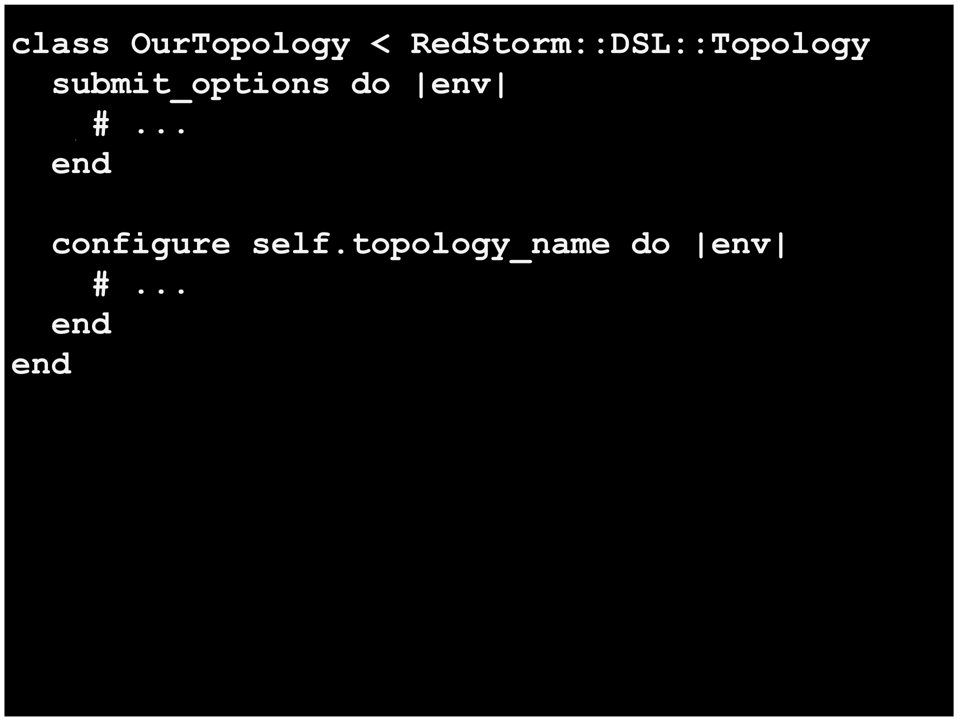


```
class OurTopology < RedStorm::DSL::Topology
  spout_config = SpoutConfig.new(...)

  spout KafkaSpout, [spout_config] do
    output_fields :bytes
  end
```

Add a Kafka spout

```
class OurTopology < RedStorm::DSL::Topology
  bolt ContainerDeserializerBolt do
    source KafkaSpout, :shuffle
  end

  bolt ObjectFilterBolt do
    source ContainerDeserializerBolt
  end

  # ...
```

Not much here; just one bolt passing to another



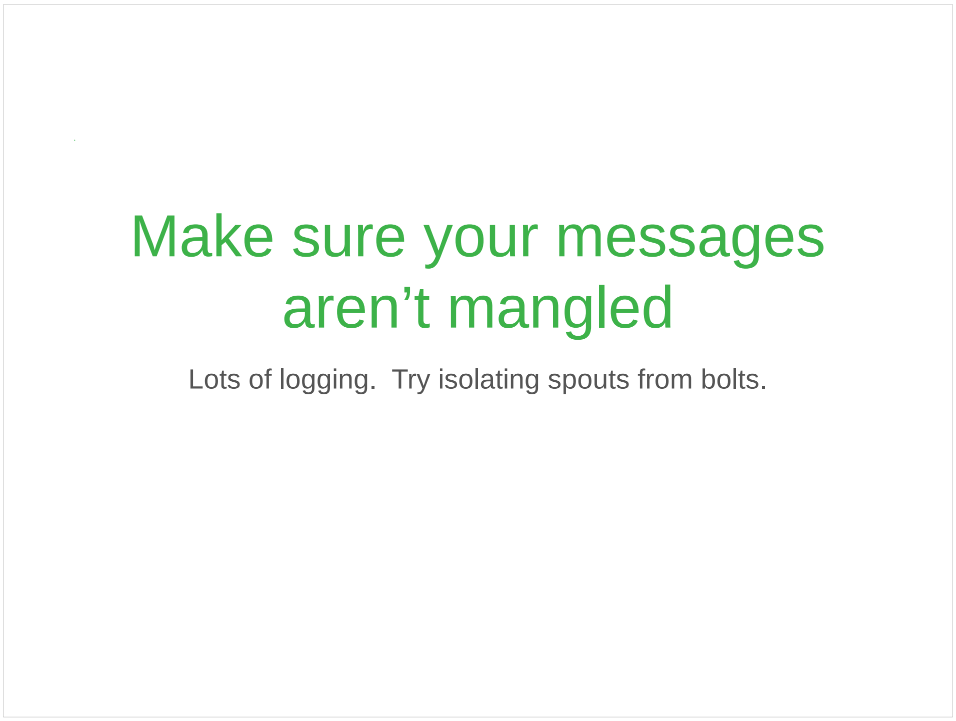
```
class OurTopology < RedStorm::DSL::Topology
  submit_options do |env|
    # ...
  end

  configure self.topology_name do |env|
    # ...
  end
end
```

Not much here; just one bolt passing to another

```
class OurBolt < RedStorm::DSL::Bolt
  output_fields :bytes, :dlq

  on_init do
    @connection = # ...
  end

  on_receive do |tuple|
   ...
  end
end
```

What does a bolt look like?

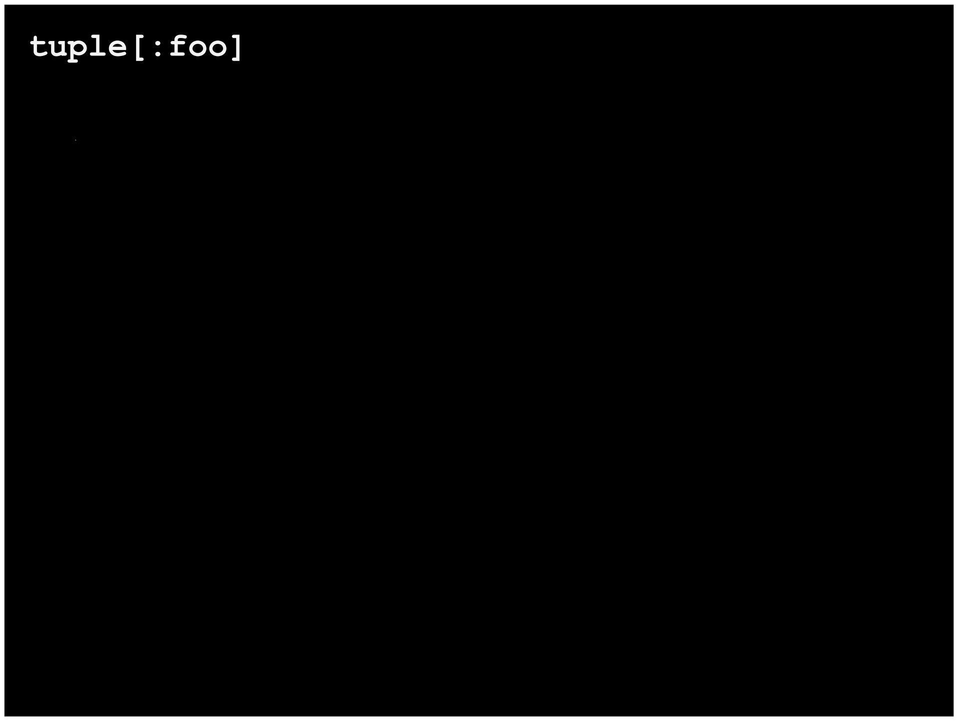
Lessons Learned / Pitfalls



Make sure your messages aren’t mangled

Lots of logging. Try isolating spouts from bolts.

E.g., our test of a factoried object being passed between bolts. Solved by wrapping in base64.

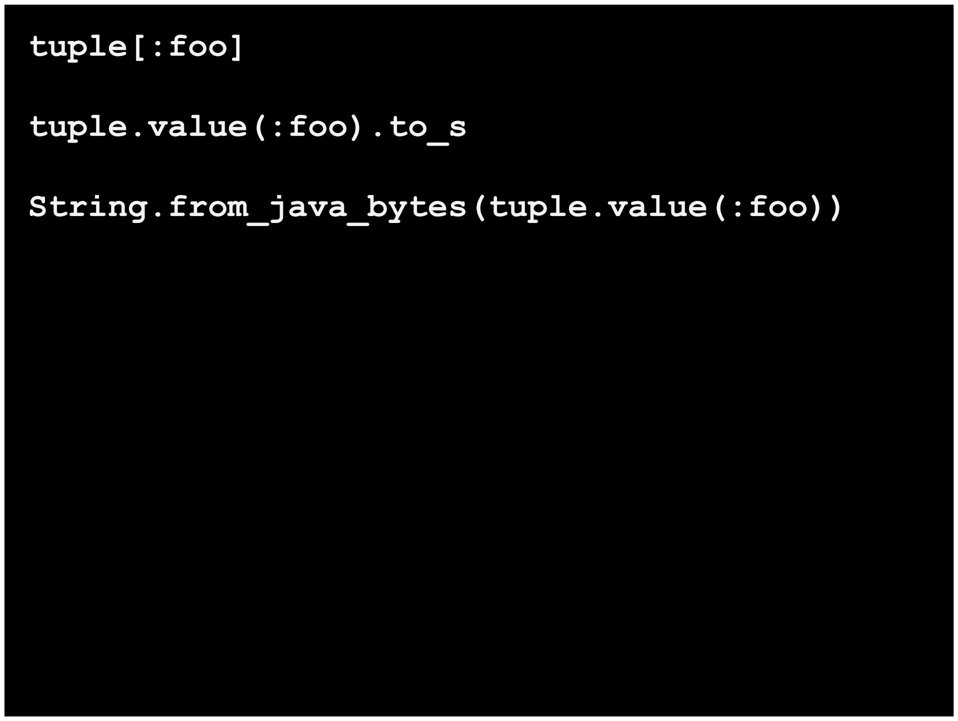


```
tuple[:foo]
```

Defining a new bolt or topology is pretty simple

```
tuple[:foo]

tuple.value(:foo).to_s
```

Defining a new bolt or topology is pretty simple

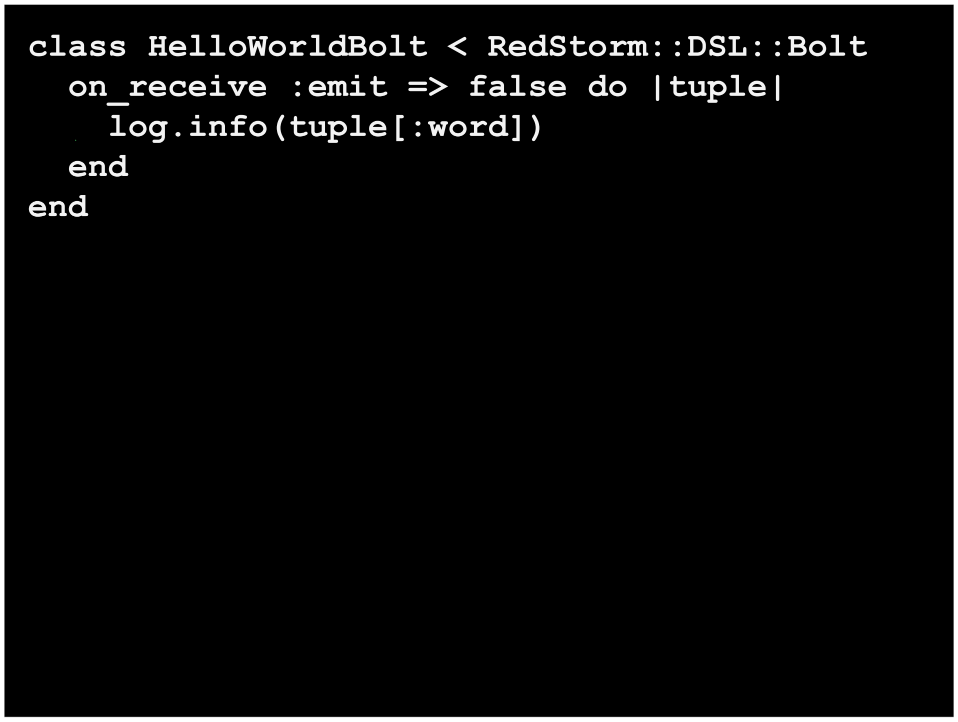


```
tuple[:foo]

tuple.value(:foo).to_s

String.from_java_bytes(tuple.value(:foo))
```

Defining a new bolt or topology is pretty simple

The DSL doesn’t subclass directly

Use methods, not blocks



```ruby
class HelloWorldBolt < RedStorm::DSL::Bolt
  on_receive :emit => false do |tuple|
    log.info(tuple[:word])
  end
end
```

Defining a new bolt or topology is pretty simple

```
class GenericBolt < RedStorm::DSL::Bolt
  def on_receive(tuple)
    log.info(tuple[:word])
  end
end
```

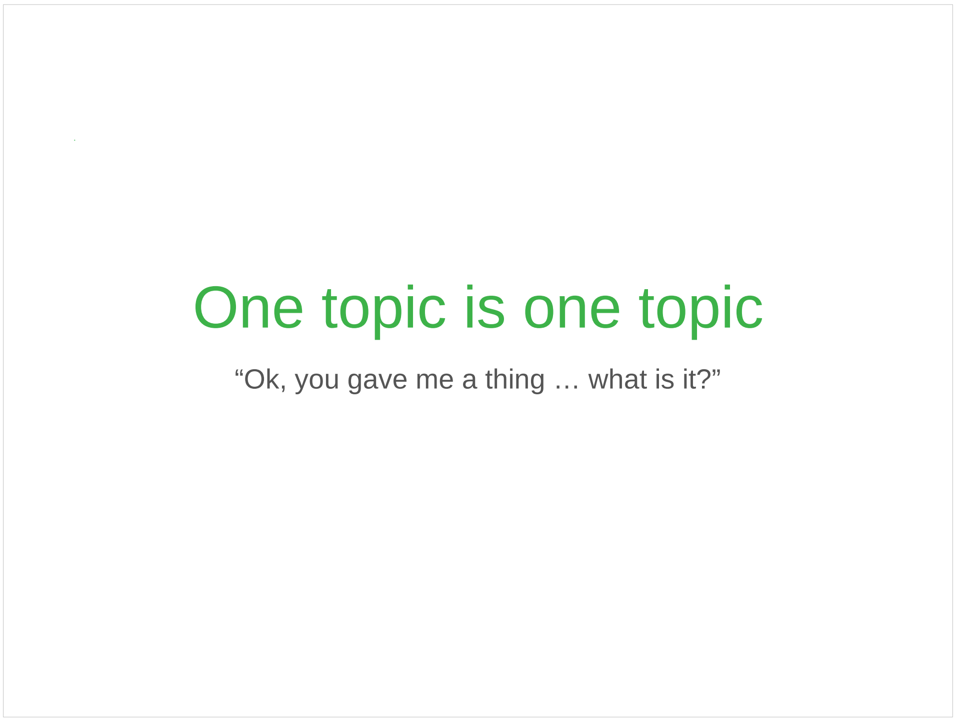
Generic bolt code can be shared, and subclassed

```
# Topology-specific subclass
class HelloWorldBolt < GenericBolt
  on_receive :on_receive

  def log
   # topology-specific logging code
  end
end
```

Generic bolt code can be shared, and subclassed

Make sure you ack post-emit

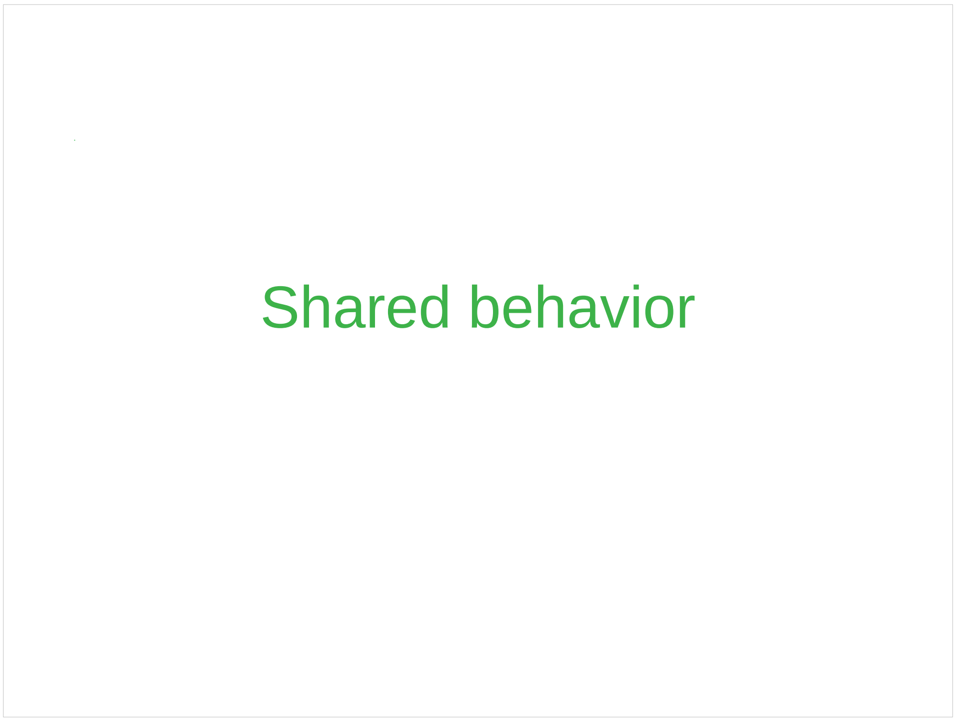
You might emit multiple tuples



One topic is one topic

“Ok, you gave me a thing … what is it?”

One datatype <-> one topic/queue



Shared behavior

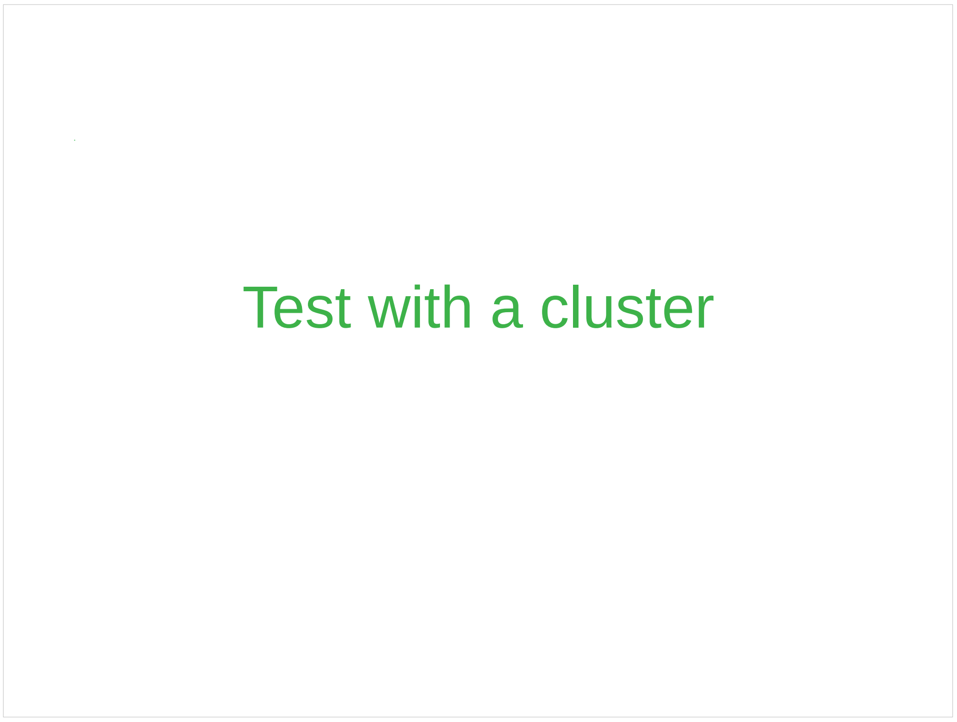
One datatype <-> one topic/queue

```
class Lookout::Bolt < RedStorm::DSL::Bolt
  # Wrap these calls in an exception handler

  def execute(*args)
    Raven.capture { super }
  end

  # Same with #prepare, #cleanup
end
```

Generic bolt code can be shared, and subclassed

```
class Lookout::Bolt < RedStorm::DSL::Bolt
  def log
    # Custom logging
  end
end
```

Generic bolt code can be shared, and subclassed

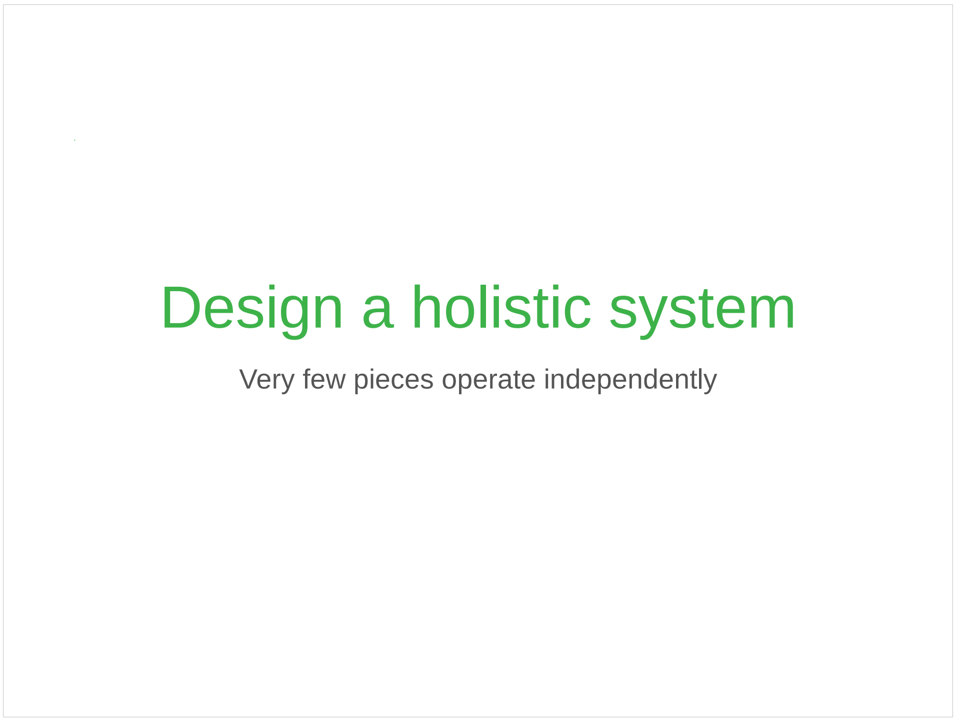


Test with a cluster

When using the locsl testing topology or even testing with a single node cluster, you may see divergent behavior to what you're going ti get in production

Submit in inactive mode

Reduce downtime



Design a holistic system

Very few pieces operate independently

When designing topologies, consider the larger system and its goals. Does the topology need to feed back into Kafka to drive other behaviors? What are the upstream and downstreams of this work?

Questions?

Thanks