

# Sagas: distributed transactions without locks in Erlang

---

MARK ALLEN - ALERTLOGIC

MARK.ALLEN@ALERTLOGIC.COM

@BYTEMEORG

# Sagas

---

**mark mcbride**

@mccv

 Follow



them: is that written down?  
me: we communicate in the viking tradition. Let  
me tell you the saga of that system.

RETWEETS

878

LIKES

713

2:34 PM - 10 Feb 2014



17



878



713

# SAGAS

*Hector Garcia-Molina  
Kenneth Salem*

Department of Computer Science  
Princeton University  
Princeton, N J 08544

## Abstract

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions. To alleviate these problems we propose the notion of a saga. A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution. Both the concept of saga and its implementation are relatively simple, but they have the potential to improve performance significantly. We analyze the various implemen-

the majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors. Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database [Gray81a].

In most cases, LLTs present serious performance problems. Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database [Date81a, Ullm82a]. To make a transaction atomic, the system usually locks the objects accessed by the transaction until it com-

# Types of Sagas

---

- Backward Recovery
- Forward Recovery
- “Recovery Blocks”
- Parallel Sagas

# Sagas in Erlang

---

# The Big Idea

Fold over a list of closures...

# The Big Idea

Fold over a list of closures...  
...unless there's an error.



# The Big Idea

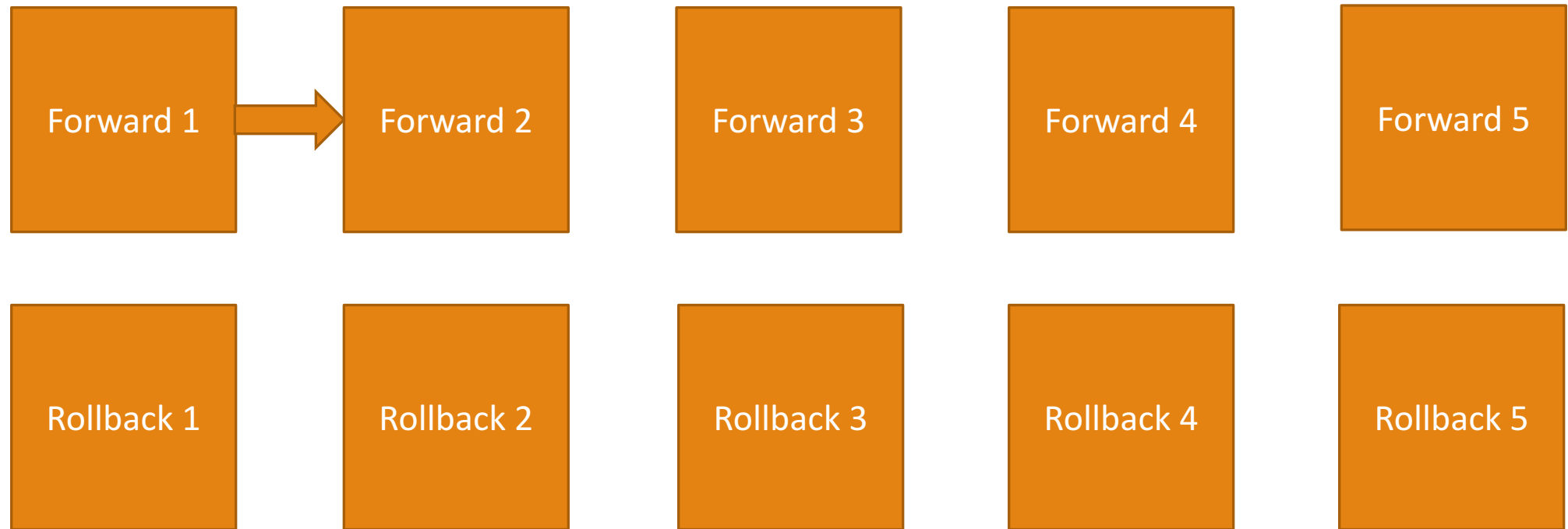
Fold over a list of closures...

...unless there's an error;

Then, fold over a list of closures.

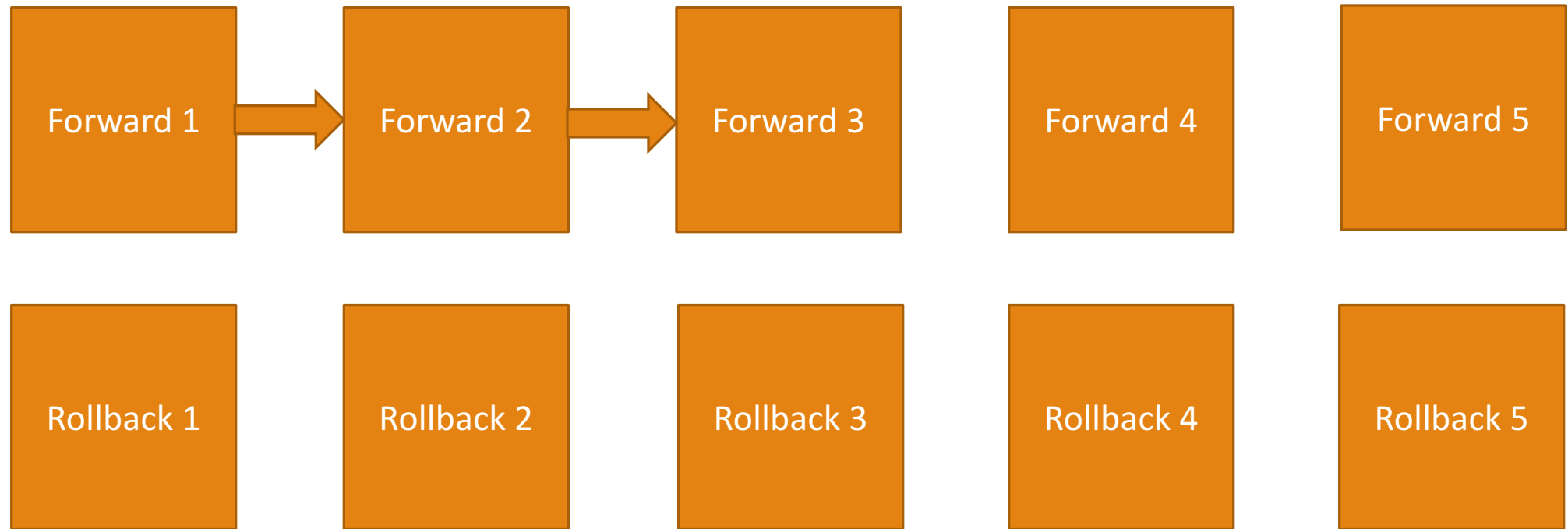
# Flows

---



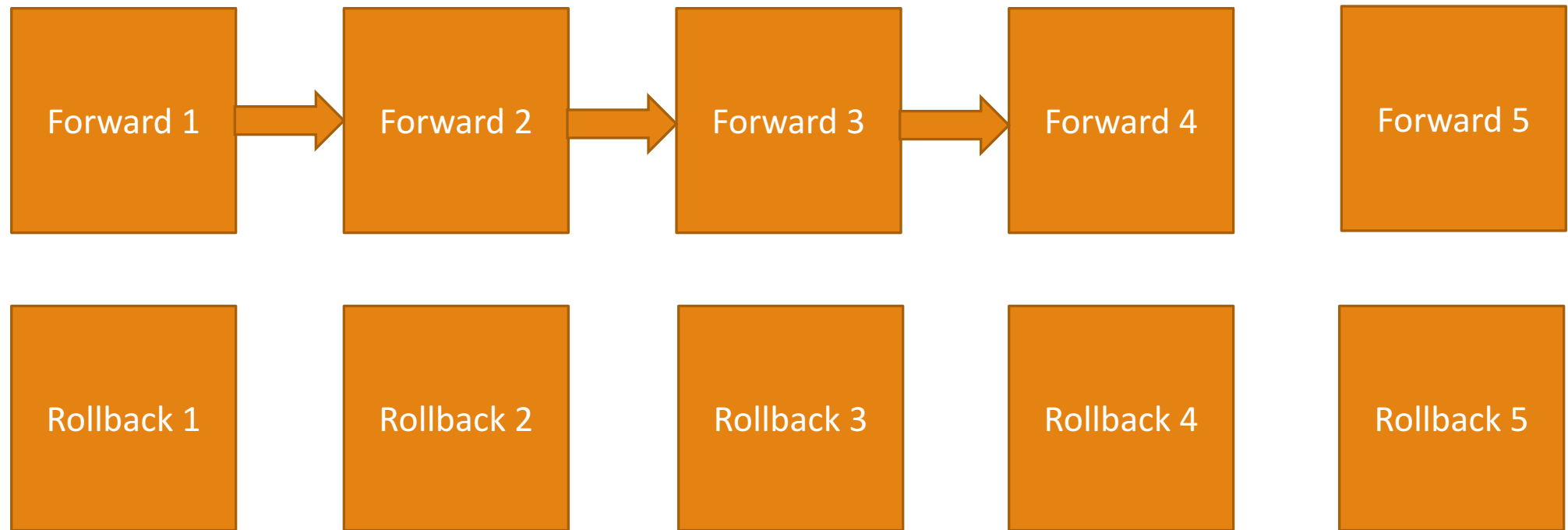
# Flows

---



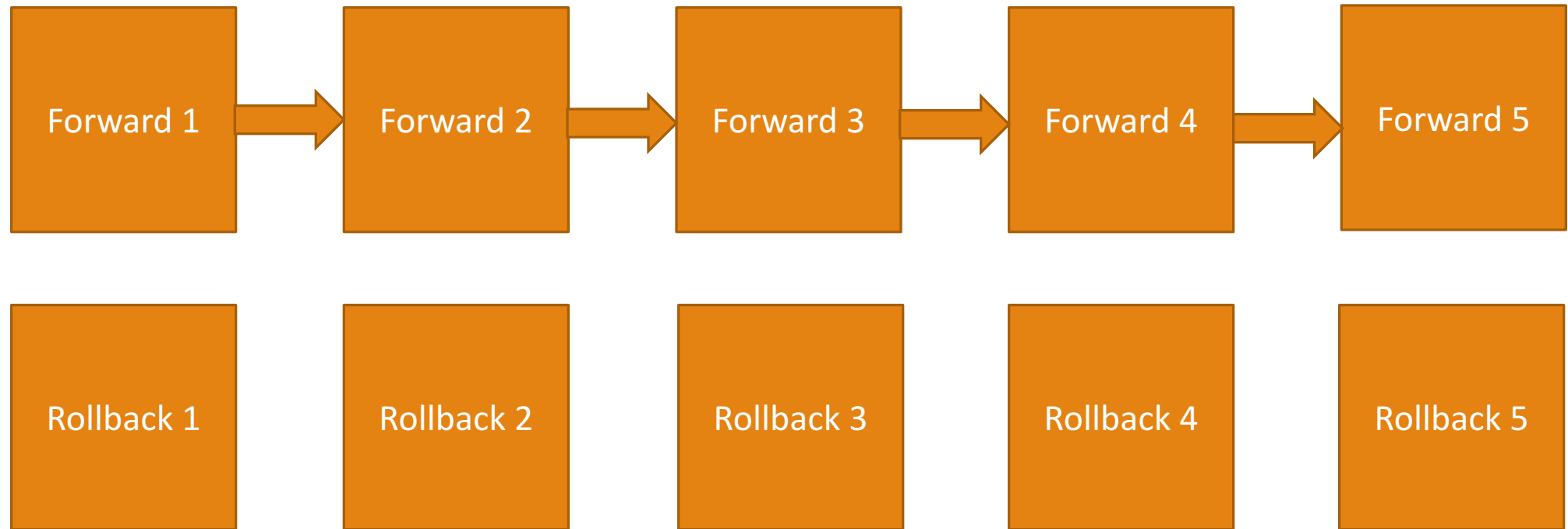
# Flows

---



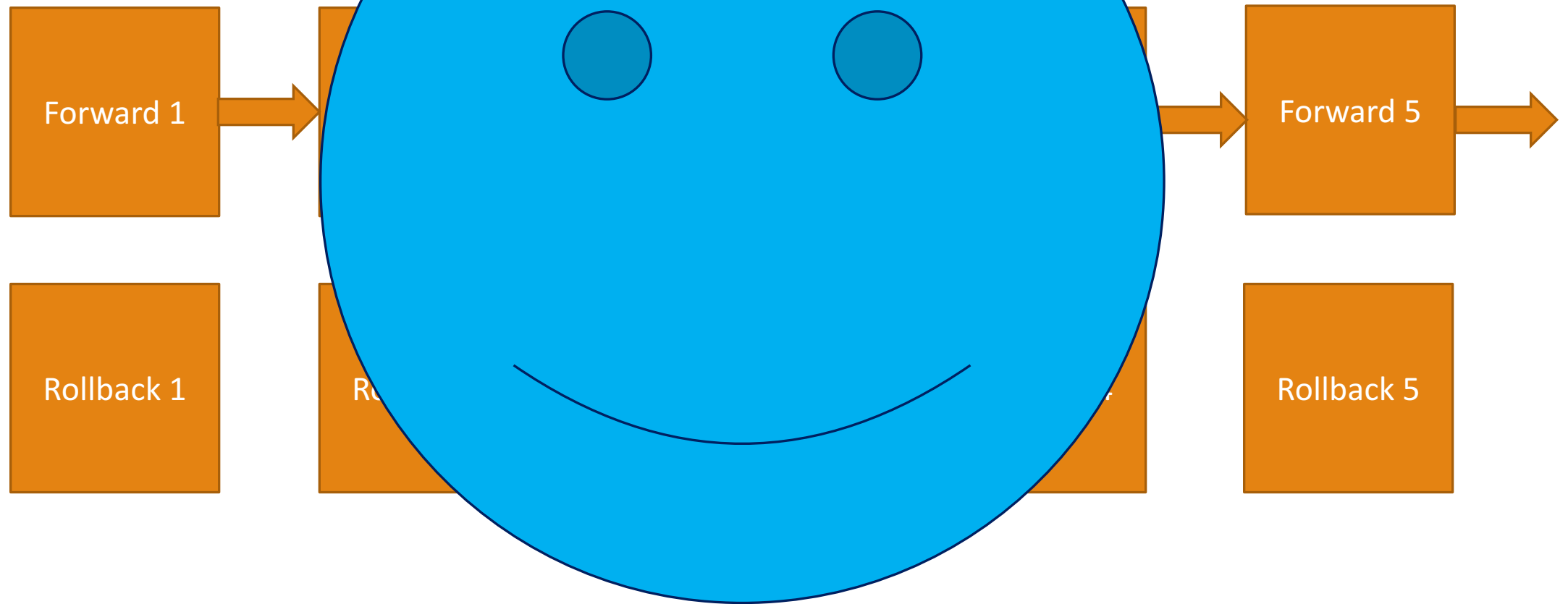
# Flows

---



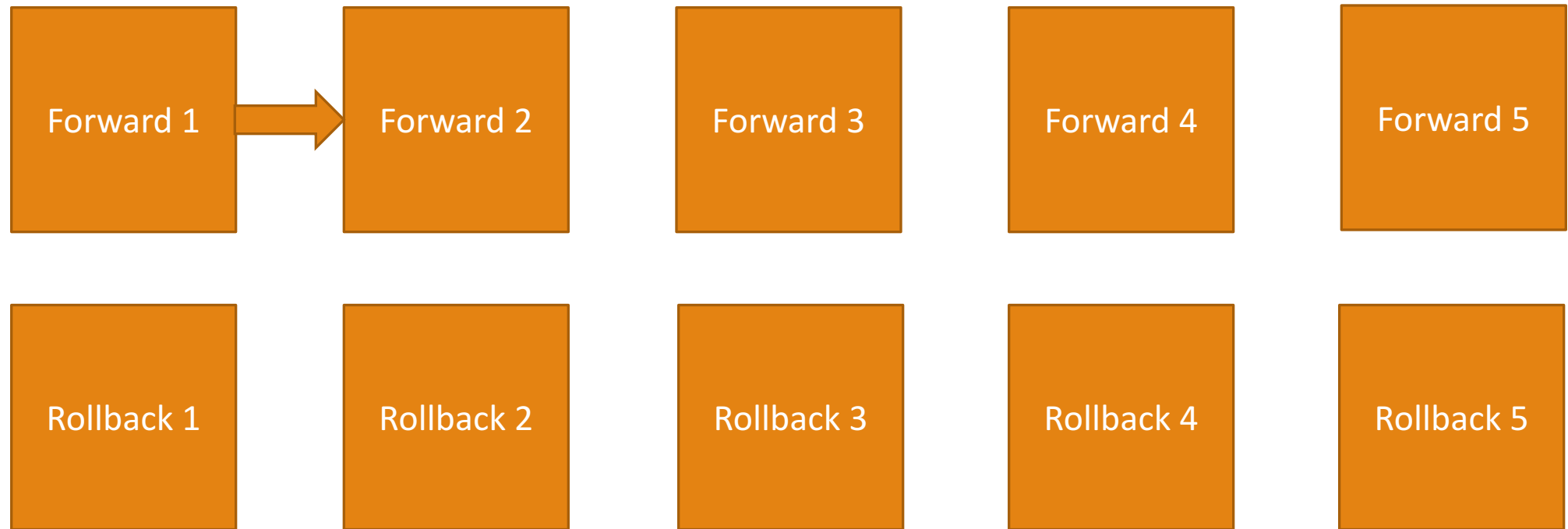
# Flows

---



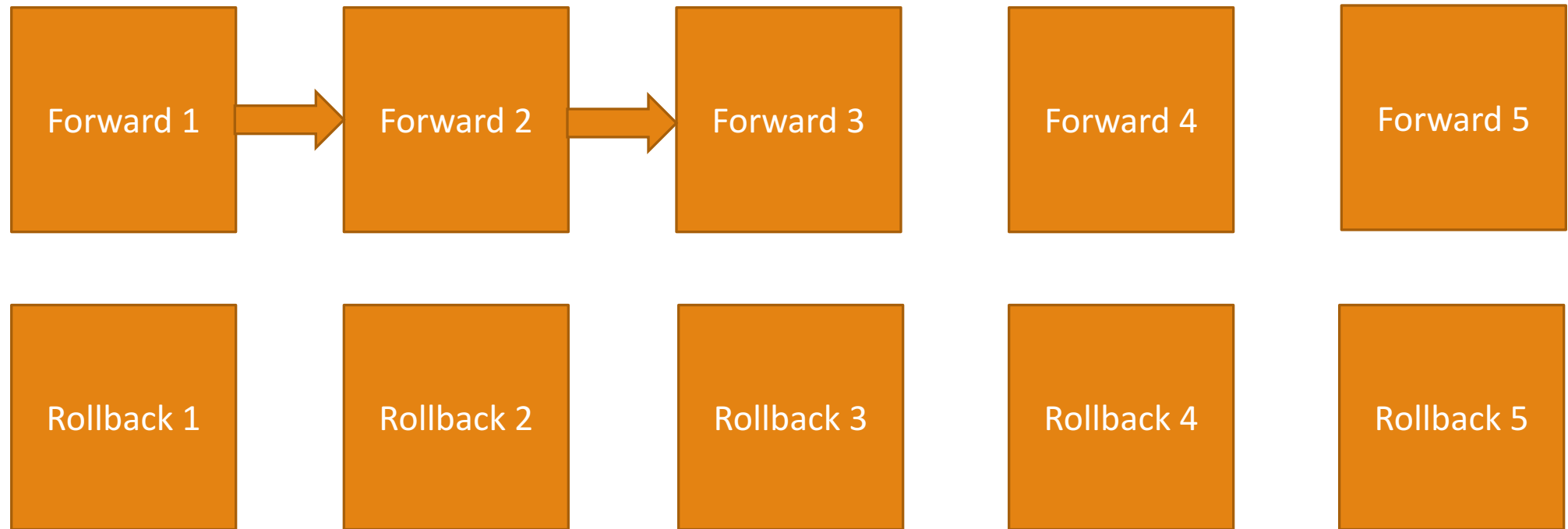
# Flows

---



# Flows

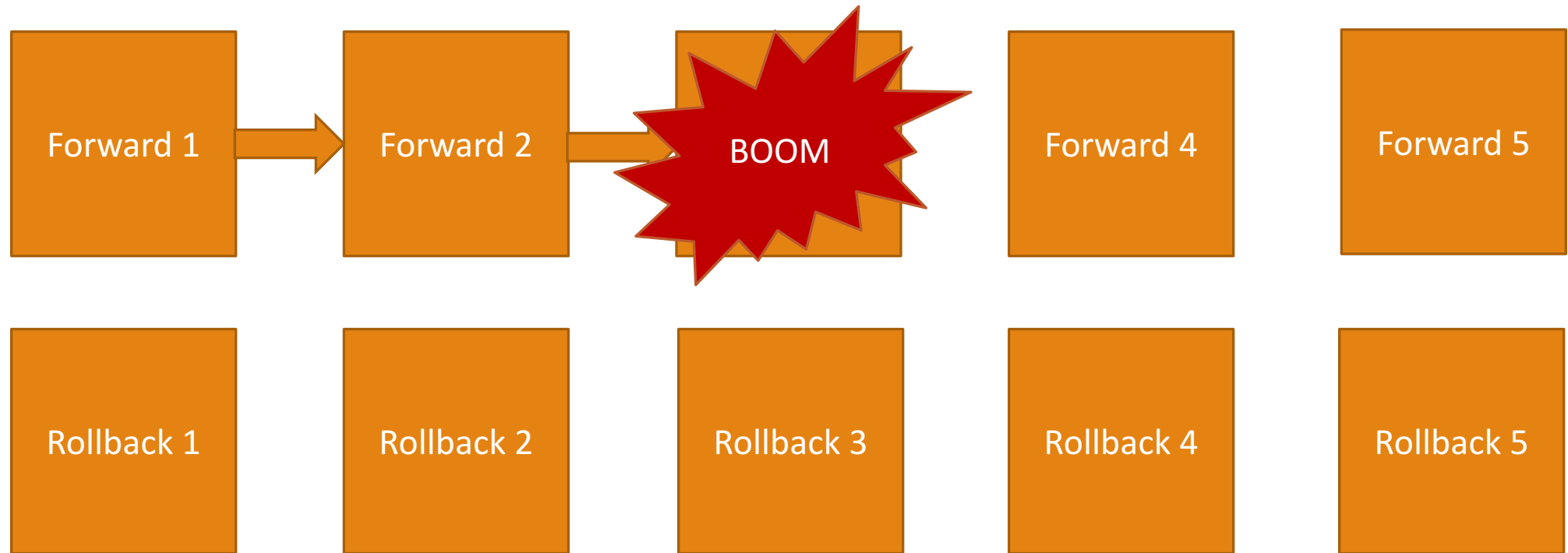
---





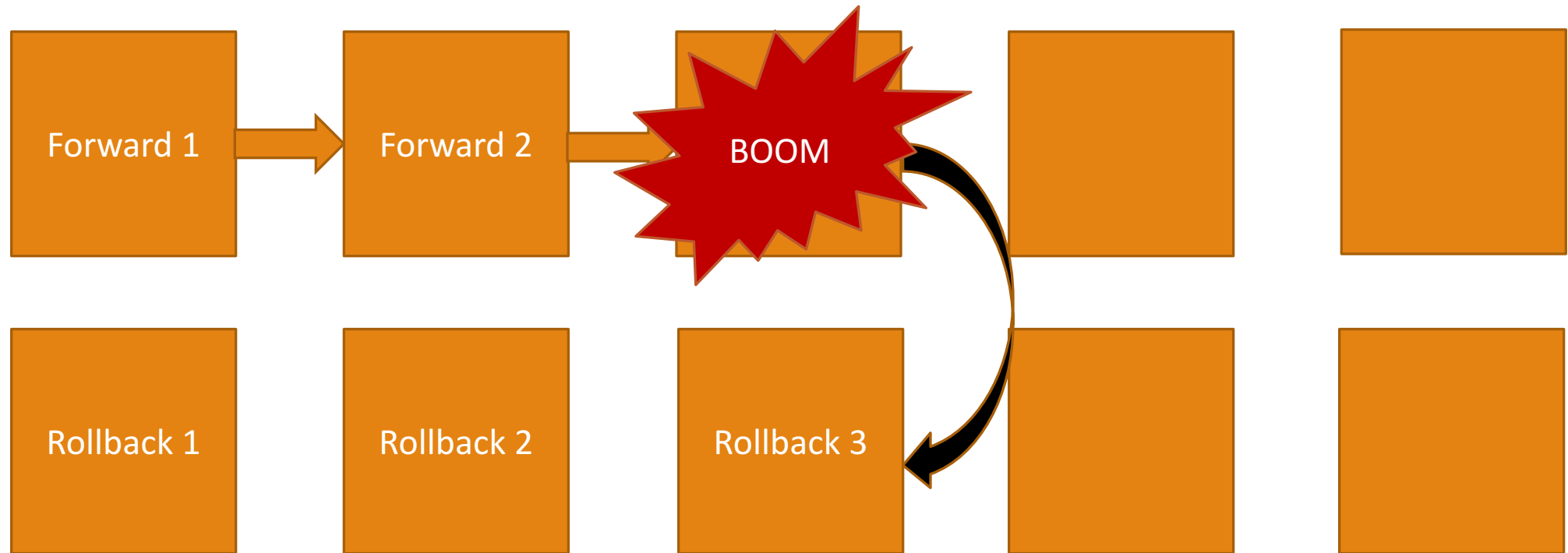
# Flows

---



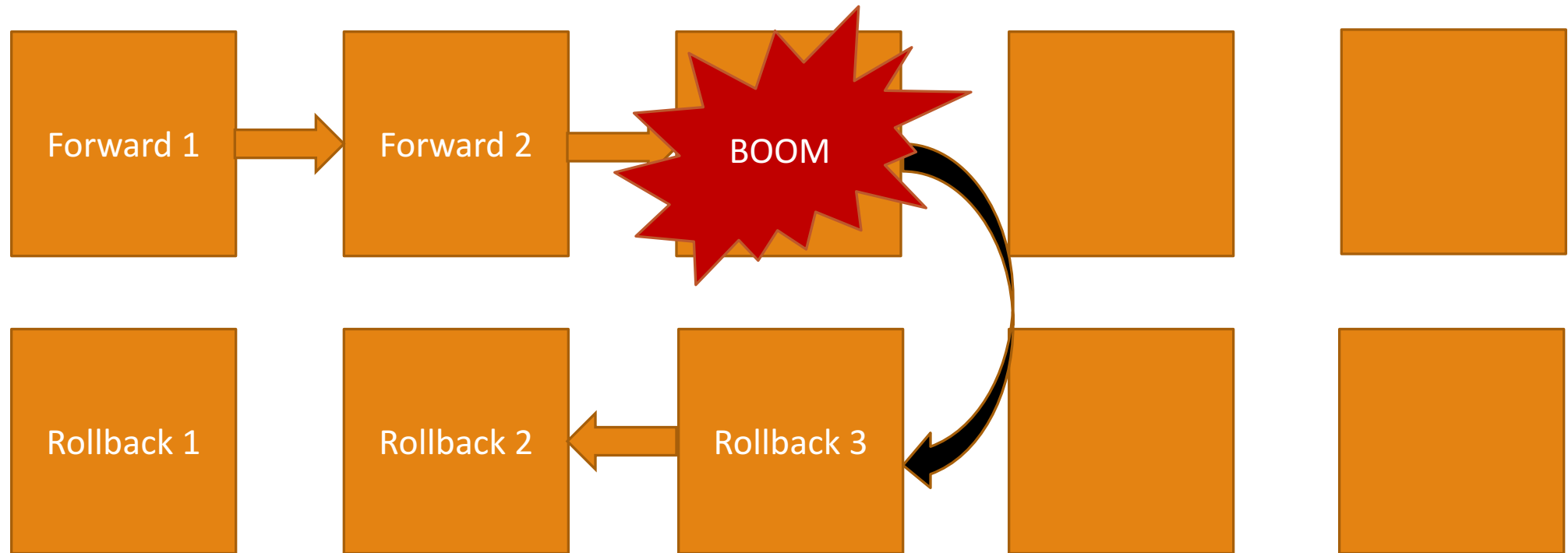
# Flows

---



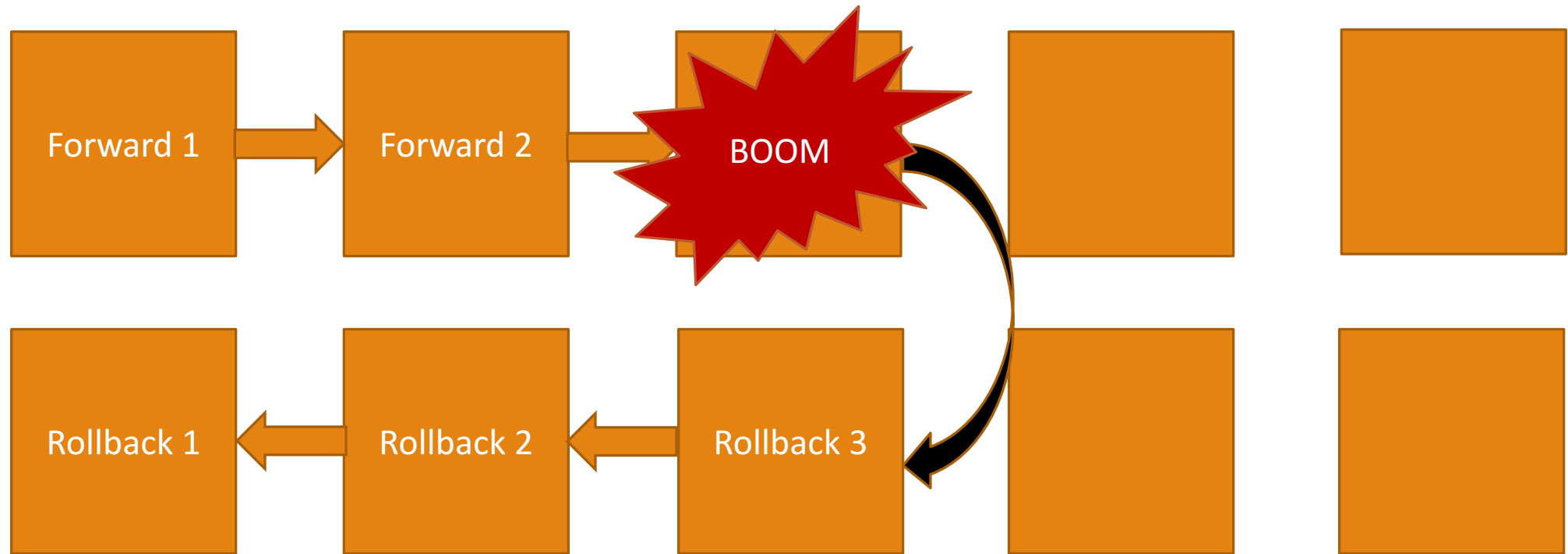
# Flows

---



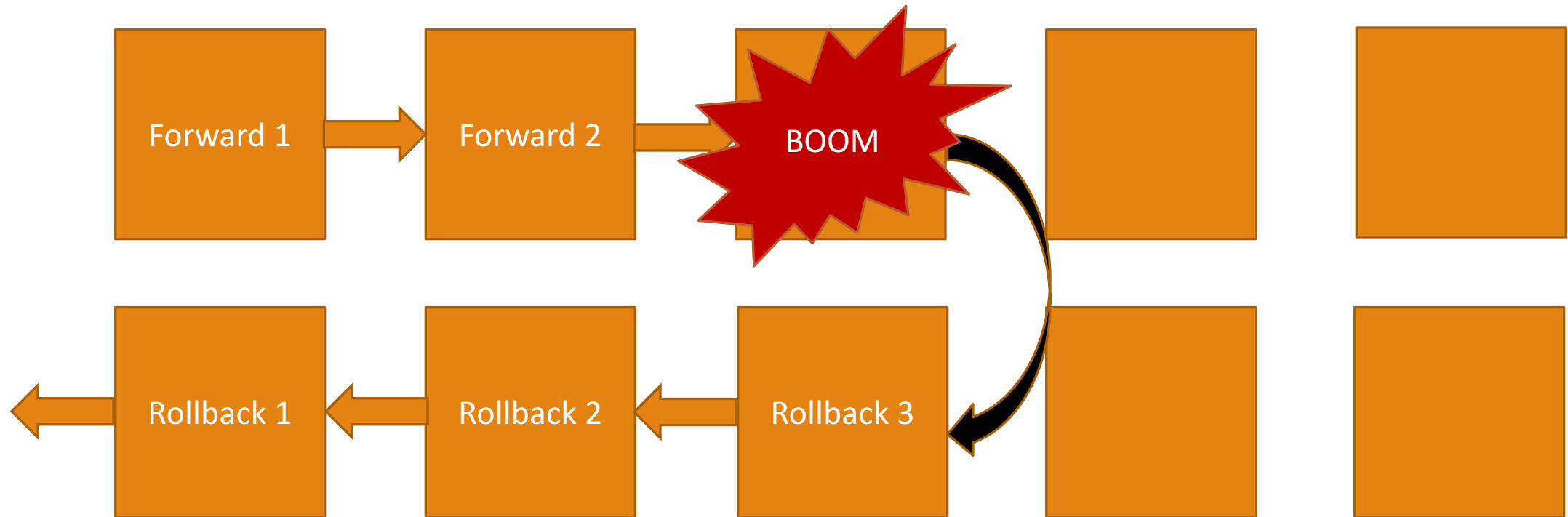
# Flows

---



# Flows

---






GitHub, Inc. github.com/mrallen1/gisla

Search NixOS packages VisuAlgo - vi...ugh animation Clip to OneNote Pullquote

This repository Search

Pull requests Issues Gist

mrallen1 / gisla

Unwatch 3

Star 22

Fork 0

<> Code

Issues 0

Pull requests 0

Projects 0

Wiki

Pulse

Graphs

Settings

A library that implements the sagas pattern for Erlang

Edit

16 commits

2 branches

1 release

1 contributor

MIT

Branch: master

New pull request

Create new file

Upload files

Find file

Clone or download

mrallen1 Set version 1.0.0

Latest commit b4fb223 on Sep 26, 2016

include	Change status -> result	4 months ago
src	Set version 1.0.0	4 months ago
.gitignore	Ignore rebar.lock	4 months ago
LICENSE	Initial commit	4 months ago
README.md	Rewrite README	4 months ago
rebar.config	rebar3 ftw	4 months ago

README.md

# Gisla

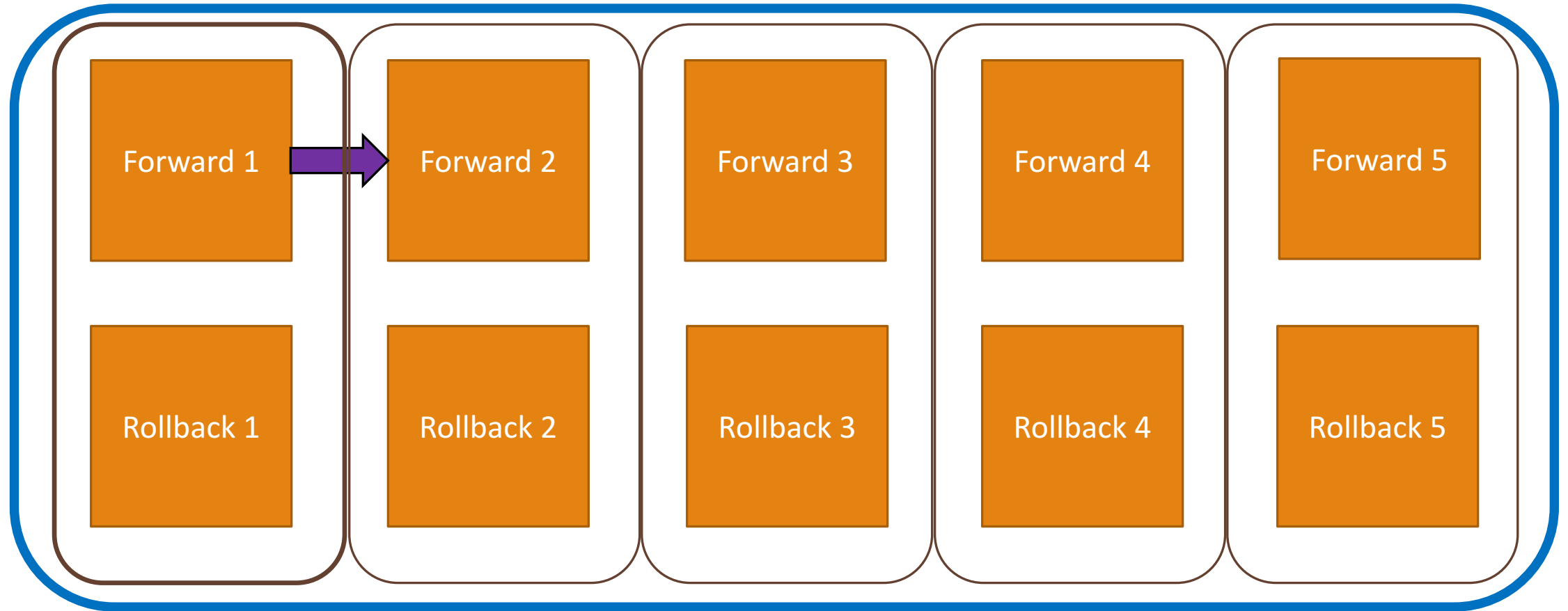
# Terminology

---

- Operation
- Step
- Transaction

# Terminology

---







**Tyler Treat**

@tyler\_treat

Following



Sagas are a great pattern for highly available microservices, but don't screw it up with slow and fragile messaging.

LIKES

7



3:47 PM - 25 May 2017 from [Broomfield, CO](#)



1



7



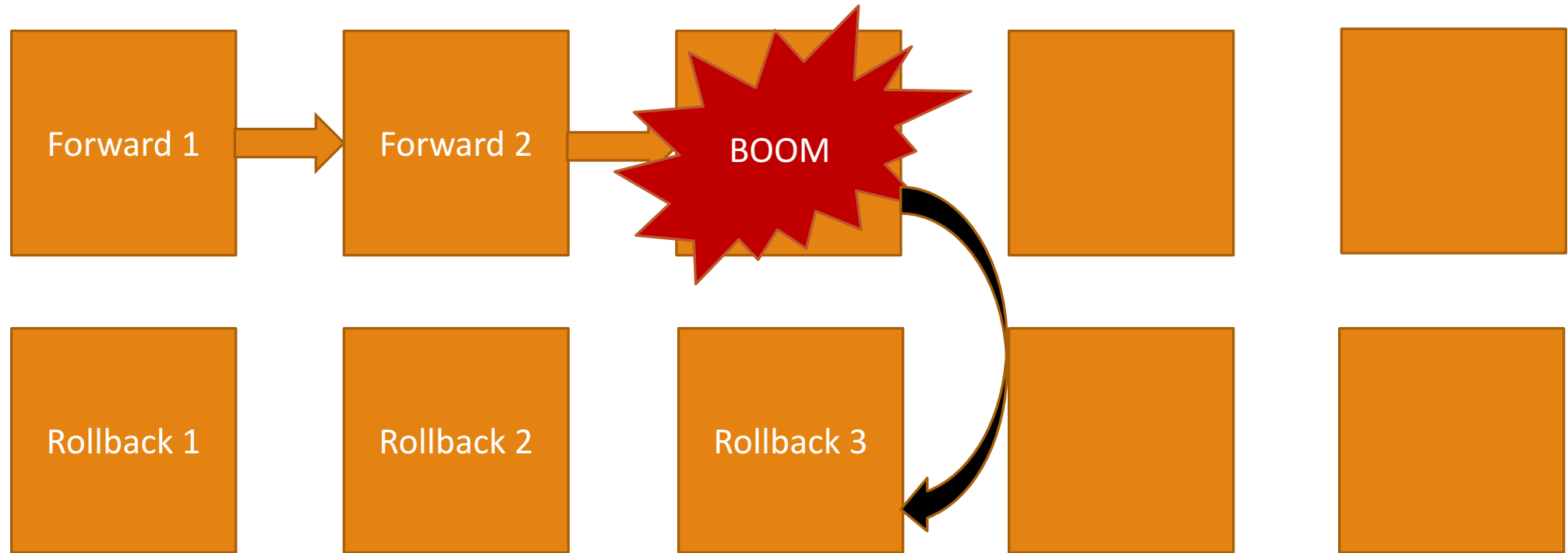
# Implicit Assumptions and Requirements

Compensating  
Closures  
Cannot Abort

# Idempotent Requests

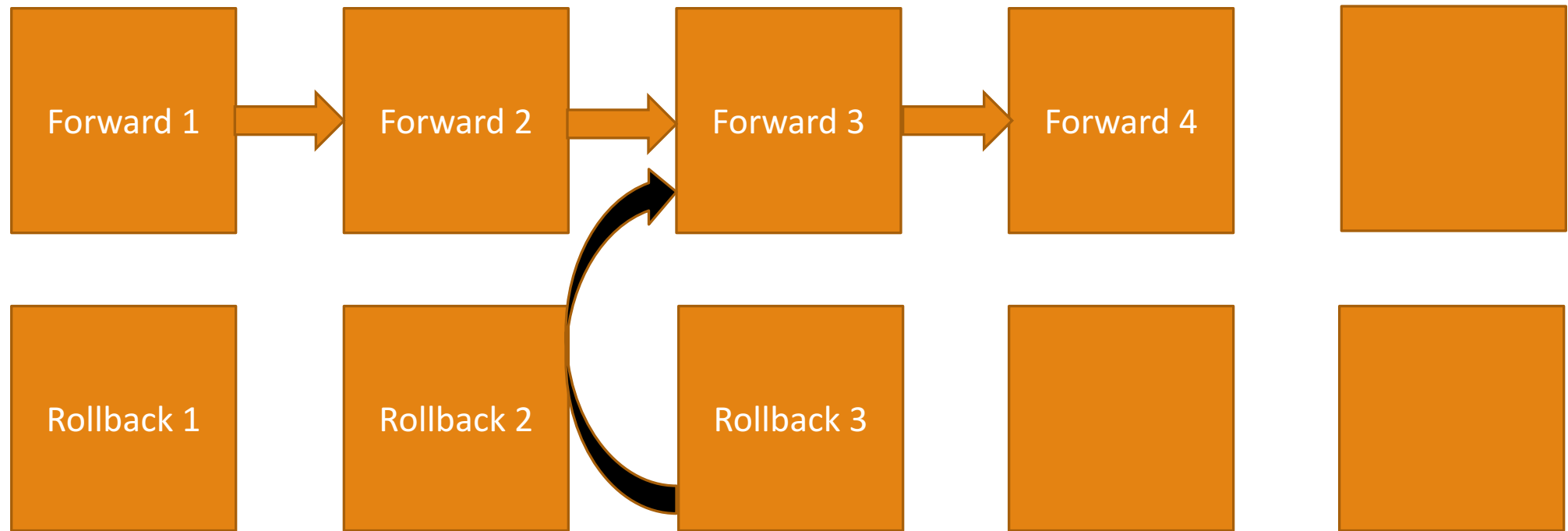
# What if??

---



# What if??

---



# Futures in Erlang

---

Why does Erlang  
need futures?!




It doesn't!

Except...




# Asynchronous Saga Operations


GitHub, Inc. [github.com/mrallen1/criswell](#)

Search NixOS packages VisuAlgo - vi...ugh animation Clip to OneNote Pullquote

 This repository

[Pull requests](#) [Issues](#) [Gist](#)

 **mrallen1 / criswell**

[Unwatch](#) 3 [Star](#) 4 [Fork](#) 0

[Code](#) [Issues 0](#) [Pull requests 0](#) [Projects 0](#) [Wiki](#) [Pulse](#) [Graphs](#) [Settings](#)

"Future events such as these will affect you in the future" - Futures for Erlang

Edit

3 commits


1 branch

0 releases


1 contributor

Branch: master [New pull request](#)

[Create new file](#) [Upload files](#) [Find file](#) [Clone or download](#)

 **mrallen1** Add await/1 and a promise() type def Latest commit 097895b 15 days ago

<a href="#">include</a>	Add await/1 and a promise() type def	15 days ago
<a href="#">src</a>	Add await/1 and a promise() type def	15 days ago
<a href="#">test</a>	Add a basic test	15 days ago
<a href="#">.gitignore</a>	Initial commit	15 days ago
<a href="#">LICENSE</a>	Initial commit	15 days ago
<a href="#">README.md</a>	Add await/1 and a promise() type def	15 days ago

 **README.md**

# Criswell

# Implementation

Semantics?

# Property Based Tests

---

# What is it?

---

- Write one or more invariants
- Test values are automatically generated and the invariant tested
- Failures are shrunk to the smallest possible failure case
- Modify code (or test case) and restart test
- Tests are usually timeboxed
- Extremely useful to build confidence in complex scenarios

# Academic history

---

## QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

### QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen  
Chalmers University of Technology  
koen@cs.chalmers.se

John Hughes  
Chalmers University of Technology  
rjmh@cs.chalmers.se

#### ABSTRACT

QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in which the tool was successfully used, and also point out some pitfalls to avoid. Random testing is especially suitable for functional programs because properties can be stated at a fine grain. When a function is built from separately tested components, then random testing suffices to obtain good coverage of the definition under test.

#### 1. INTRODUCTION

Testing is by far the most commonly used approach to ensuring software quality. It is also very labour intensive, accounting for up to 50% of the cost of software development. Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-checks, it usually works'), in practice it is still a major part of functional program development.

The cost of testing motivates efforts to automate it, wholly or partly. Automatic testing tools enable the programmer to complete testing in a shorter time, or to test more thoroughly in the available time, and they make it easy to repeat tests after each modification to a program. In this paper we describe a tool, QuickCheck, which we have developed for testing Haskell programs.

Functional programs are well suited to automatic testing. It is generally accepted that pure functions are much easier to test than side-effecting ones, because one need not be concerned with a state before and after execution. In an imperative language, even if whole programs are often pure functions from input to output, the procedures from which they are built are usually not. Thus relatively large units

monad are hard to test), and so testing can be done at a fine grain.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an automatically checkable criterion of doing so. We have chosen to use formal specifications for this purpose. We have designed a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases. The specification language is embedded in Haskell using the class system. Properties are normally written in the same module as the functions they test, where they serve also as checkable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [11], which competes surprisingly favourably with systematic methods in practice. However, it is meaningless to talk about random testing without discussing the distribution of test data. Random testing is most effective when the distribution of test data follows that of actual data, but when testing reusable code units as opposed to whole systems this is not possible, since the distribution of actual data in all subsequent reuses is not known. A uniform distribution is often used instead, but for data drawn from infinite sets this is not even meaningful – how would one choose a random closed  $\lambda$ -term with a uniform distribution, for example? We have chosen to put distribution under the human tester's control, by defining a *test data generation language* (also embedded in Haskell), and a way to observe the distribution of test cases. By programming a suitable generator, the tester can not only control the distribution of test cases, but also ensure that they satisfy arbitrarily complex invariants.

An important design goal was that QuickCheck should be *lightweight*. Our implementation consists of a single pure



# Implementations for

---

- Go
- C/C++
- Clojure
- Scala
- Haskell (of course)
- ... lots of others ...
- Erlang/Elixir ([watch Thomas Arts talk at ElixirConf EU 2015](#))

# Erlang implementations

---

- Quviq Commercial QuickCheck (<http://www.quviq.com>)
- Free (but not open source) “QuickCheck mini” – doesn’t do statem
- PropEr (<https://github.com/manopapad/proper>)
- Triq (<https://github.com/triqng/triq>)

# Generators

---

- All standard basic types:
  - integers,
  - floats,
  - atoms,
  - binaries,
  - strings,
  - lists
- Lists of basic types
- User defined generators using ?LET and ?SUCHTHAT

# Propositions

---

- ?FORALL(Variable, generator\_function(), test\_function(Variable))

# Resources

---

- <https://github.com/mrallen1/gisla>
- <https://github.com/mrallen1/criswell>
- Sagas paper: <http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf>
- Caitie McCaffrey, Papers We Love on Sagas: <https://youtu.be/7dc4TI5ZHRg?t=27m16s>
- Caitie McCaffrey, Distributed Sagas: <https://speakerdeck.com/caitiem20/distributed-sagas-a-protocol-for-coordinating-microservices>
- These slides: <https://speakerdeck.com/mrallen1/sagas-distributed-transactions-without-locks>