facebook

Fighting Spam with Haskell

Simon Marlow

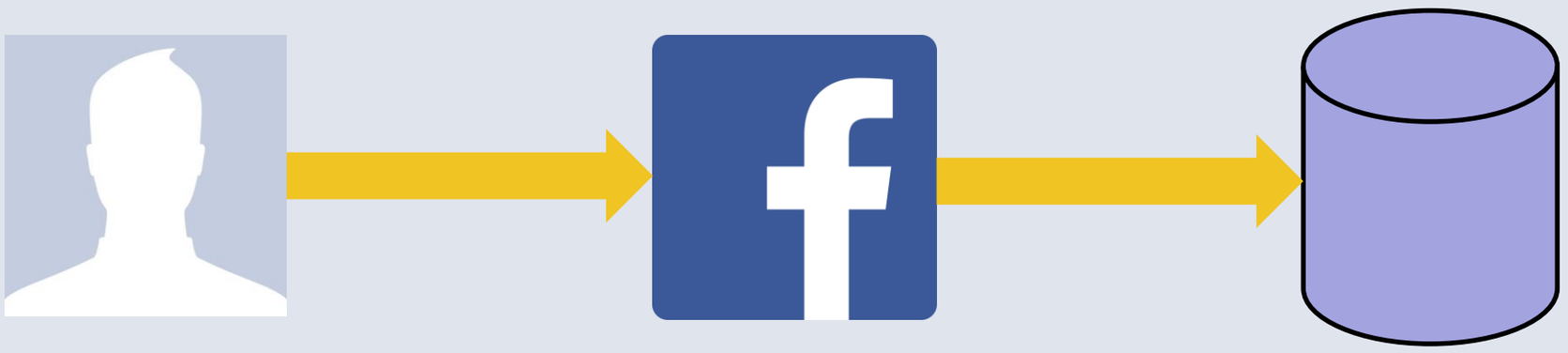
5 Sept 2015

Headlines

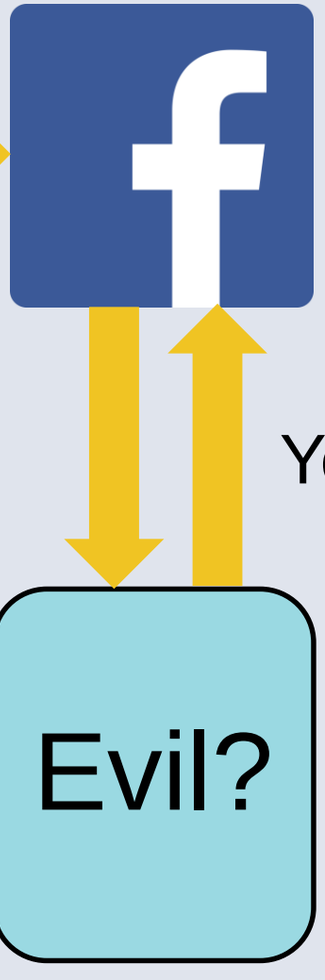
- Migrated a large service to Haskell
  - thousands of machines
  - every action on Facebook (and Instagram) runs some Haskell code
  - live code updates (<10 min)
- This talk
  - The problem we’re solving: abuse detection & remediation
  - Why (and how) Haskell?
  - Tales from the trenches

The problem

- There is spam and other types of abuse
  - Malware attacks, credential stealing
  - Sites that trick users into liking/sharing things or divulging passwords
  - Fake accounts that spam people and pages
- Spammers can use automation and viral attacks
- Want to catch as much as possible in a completely automated way
- Squash attacks quickly and safely

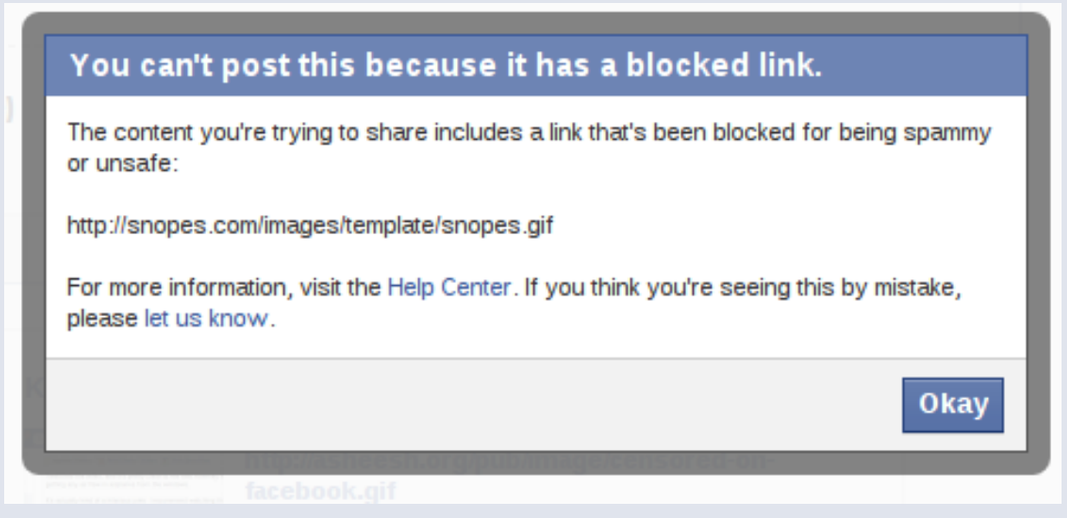


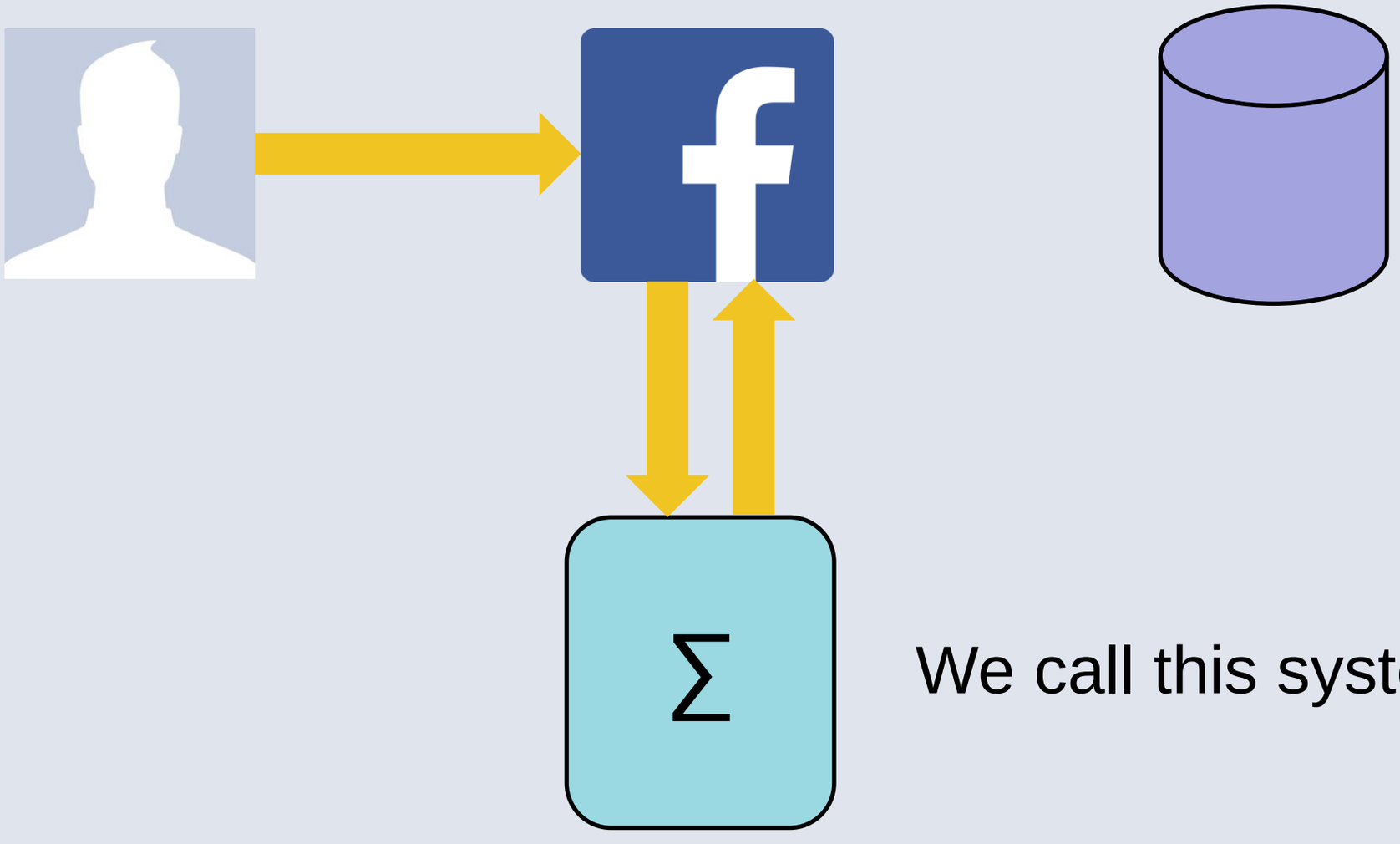




Yes!

Evil?


You can't post this because it has a blocked link.
The content you're trying to share includes a link that's been blocked for being spammy or unsafe:
http://snopes.com/images/template/snopes.gif
For more information, visit the Help Center. If you think you're seeing this by mistake, please let us know.
Okay



We call this system Sigma

Sigma :: Content -> Bool

- Sigma classifies tens of billions of actions per day
  - Facebook + Instagram

- Sigma is a *rule engine*
  - For each action type, evaluate a set of rules
  - Rules can block or take other action
  - Manual + machine learned rules
  - Rules can be updated live

- Highly effective at eliminating spam, malware, malicious URLs, etc. etc.

How do we define rules?

Example

- Fanatics are spamming their friends with posts about Functional Programming!
- Let’s fix it!

Example

- We want a rule that says
  - If the person is posting about Functional Programming
  - And they have >100 friends
  - And more than half of their friends like C++
  - Then block, else allow

Need info about the content

Need to fetch the friend list

Need info about each friend

Our rule, in Haskell

```
fpSpammer :: Haxl Bool
fpSpammer =
```

- Haxl is a monad
- “Haxl Bool” is the type of a computation that may:
  - do data-fetching
  - consult input data
  - maybe throw exceptions
  - finally, return a Bool

Our rule, in Haskell

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP
 where
  talkingAboutFP =
   strContains “Functional Programming” <$> postContent
```

- postContent is part of the input (say)

```
postContent :: Haxl Text
```

Our rule, in Haskell

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP .&&
  numFriends .> 100
 where
  talkingAboutFP =
    strContains “Functional Programming” <$> postContent
```

```
(.&&) :: Haxl Bool -> Haxl Bool -> Haxl Bool
(.>)  :: Ord a => Haxl a -> Haxl a -> Haxl Bool
numFriends :: Haxl Int
```

Our rule, in Haskell

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP .&&
  numFriends .> 100 .&&
  friendsLikeCPlusPlus
 where
  talkingAboutFP =
    strContains “Functional Programming” <$> postContent

friendsLikeCPlusPlus = do
  friends <- getFriends
  cppFriends <- filterM likesCPlusPlus friends
  return (length cppFriends >= length friends `div` 2)
```

Observations

- Our language is Haskell + libraries
  - Embedded Domain-Specific Language (EDSL)
  - Users can pick up a Haskell book and learn about it
  - Tradeoff: not exactly the syntax we might have chosen, but we get to take advantage of existing tooling, documentation etc.
- Focus on expressing functionality concisely, avoid operational details
- “pure” semantics
  - no side effects – easy to reason about
  - scope for automatic optimisation

Efficiency

- Rules are data + computation
- Fetching remote data can be slow
- Latency is important!
  - We’re on the clock: the user is waiting

- So what about efficiency?

Fetching data efficiently is all that matters.

1. Fetch only the data you need to make a decision
2. Fetch data concurrently whenever possible

Let’s deal with (1) first.

Example

- We want a rule that says
  - If the person is posting about Functional Programming
  - And they have >100 friends
  - And more than half of their friends like C++
  - Then block, else allow

Fast

Slow

Very slow

- Avoid slow checks if fast checks already determine the answer

.&& is short-cutting

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP .&&
  numFriends .> 100 .&&
  friendsLikeCPlusPlus
 where
  talkingAboutFP =
    strContains “Functional Programming” <$> postContent

friendsLikeCPlusPlus = do
  friends <- getFriends
  cppFriends <- filterM likesCPlusPlus friends
  return (length cppFriends >= length friends `div` 2)
```

- Programmer is responsible for getting the order right
- (tooling helps with this)

We can speculate

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP .&&
    do a <- numFriends .> 100
       b <- friendsLikeCPlusPlus
       return (a && b)
 where
  talkingAboutFP =
    strContains “Functional Programming” <$> postContent

  friendsLikeCPlusPlus = do
    friends <- getFriends
    cppFriends <- filterM likesCPlusPlus friends
    return (length cppFriends >= length friends `div` 2)
```

avoid shortcutting behaviour by explicitly evaluating both conditions

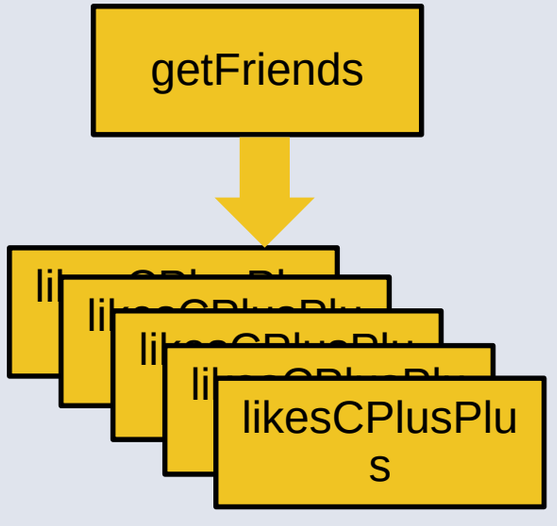
Concurrency

- Multiple independent data-fetch requests must be executed concurrently and/or batched
- Traditional languages and frameworks make the programmer deal with this
  - threads, futures/promises, async, callbacks, etc.
  - Hard to get right
  - Our users don’t care
  - Clutters the code
  - Hard to refactor later

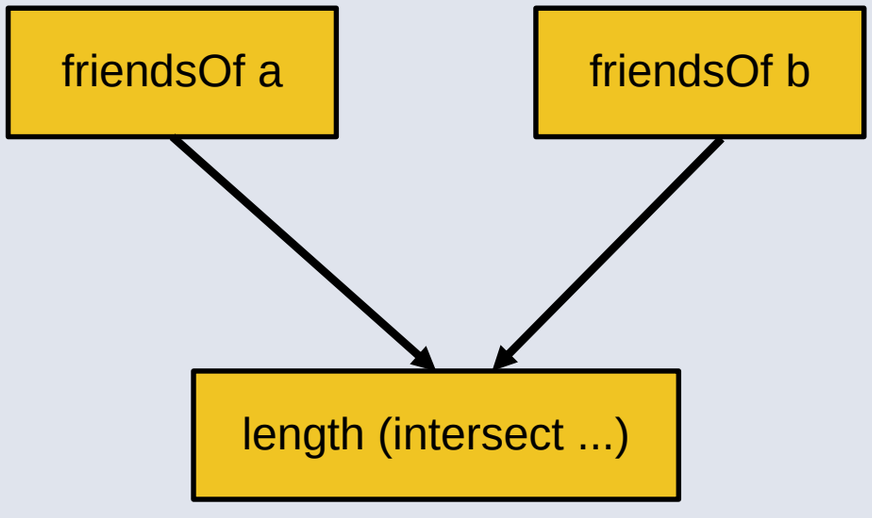
Haxl's advantage

- Because our language has no side effects, the framework can handle concurrency automatically
- We can exploit concurrency as far as data dependencies allow
- The programmer doesn't need to think about it

```
friendsLikeCPlusPlus = do
  friends <- getFriends
  cppFriends <- filterM likesCPlusPlus friends
  ...
```



```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```



How does Haxl work?

Step 1

- Haxl is a Monad
- The implementation of (>>=) will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked (Seq BlockedRequest) (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

This is the result of a computation

Blocked indicates that the computation requires this data.

Haxl may need to do IO

Monad instance

```
instance Monad Haxl where
  return a = Haxl $ return (Done a)

  Haxl m >>= k = Haxl $ do
    r <- m
    case r of
      Done a       -> unHaxl (k a)
      Blocked br c -> return (Blocked br (c >>= k))
```

If m blocks with continuation c, the continuation for m >>= k is c >>= k

So far we can only block on *one* data-fetch

- Our example will block on the first friendsOf request:

```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```

blocks here

- How do we allow the Monad to collect multiple data-fetches, so we can execute them concurrently?

First, rewrite to use *Applicative* operators

```
numCommonFriends a b =
  length <$> (intersect <$> friendsOf a <*> friendsOf b)
```

- Applicative is a weaker version of Monad

```
class Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

- When we use Applicative, Haxl can collect multiple data fetches and execute them concurrently.

Applicative instance

```
instance Applicative Haxl where
  pure = return

  Haxl f <*> Haxl x = Haxl $ do
    f' <- f
    x' <- x
    case (f',x') of
      (Done g,         Done y       ) -> return (Done (g y))
      (Done g,         Blocked br c ) -> return (Blocked br (g <$> c))
      (Blocked br c,   Done y       ) -> return (Blocked br (c <*> return y))
      (Blocked br1 c, Blocked br2 d) -> return (Blocked (br1 <> br2) (c <*> d))
```

- <*> allows both arguments to block waiting for data
- <*> can be nested, letting us collect an arbitrary number of data fetches to execute concurrently

(Some) Concurrency for free

- Applicative is a standard class in Haskell
- Lots of library functions are already defined using it
- These work concurrently when used with Haxl
- e.g.

```
sequence :: Monad m => [m a] -> m [a]
mapM     :: Monad m => (a -> b) -> m [a] -> m [b]
filterM  :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

```
friendsLikeCPlusPlus = do
  friends <- getFriends
  cppFriends <- filterM likesCPlusPlus friends
  ...
```

Back to our example

- These behave the same:

```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```

This is the version we want to write

```
numCommonFriends a b =
  length <$> (intersect <$> friendsOf a <*> friendsOf b)
```

This is the version we want to run

- Data dependencies tell us we can translate one into the other

Applicative Do

- We implemented this transformation in the compiler
- Users just turn it on:

```
{-# LANGUAGE ApplicativeDo #-}
```

- ... and get automatic concurrency/batching when using “do”
- Semantics-preserving with existing code (if certain standard properties hold), but provides better performance in some cases
- Extension pushed upstream to GHC (17/9/2015), will be in 8.0.1

Does this work in practice?

- In Sigma, our most common request executes hundreds of fetches in under ten rounds.

- Performance problems that come up in code reviews (and production) tend to be about fetching too much data, almost never about concurrency

Caching

Caching

- In addition to concurrency, caching is crucial.
  - “fetch only the data you need”
- e.g. if we have “numFriendsOf x” in two places in the codebase, we should only fetch it once
  - programmers are free to fetch data wherever, without fear of duplication, or having to plumb data around
- Means that we could safely common-up identical expressions

```
friendCondition x y = do
  numFriendsOf x .> 100 .&&
  numFriendsOf x .> numFriendsOf y
```

Side benefit of caching: *replayability*

- cache is a complete record of what was fetched.
- If we save the cache, we can re-run the request at any time without fetching any data.
  - Keep it as a unit test
  - Debug an error that occurred in production
  - Analyse performance

But caching by itself isn't enough.

- We often want to re-use decisions in multiple places.

```
anySpammer =
   fpSpammer .|| otherSpamType .|| ...
```

- Clear and concise... but we're recomputing fpSpammer
- Caching will avoid re-fetching everything that fpSpammer needs
- But it might be expensive to recompute
- Answer: *memoization*
- Everything top-level gets automatically memoized
- Manual memoization available for other things

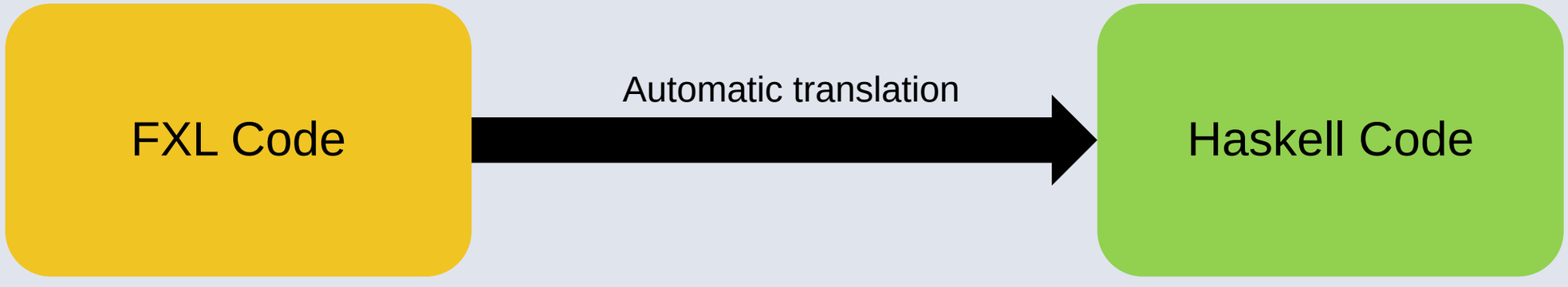
Haxl project: experience

The Haxl project

- We had an existing system and home-grown DSL, called FXL, and lots of code written in it
- Started April 2013
- By July 2015 we had deleted all the FXL code and replaced it with Haskell, and trained our engineers to use Haskell.

1. Existing source code

- hundreds of thousands of lines of existing FXL code
- Impractical and error-prone to translate code by hand
- Wrote a tool to do the migration



- Source code still FXL (during the migration), run the tool each time the code changes.

2. Migrating running code to Haskell

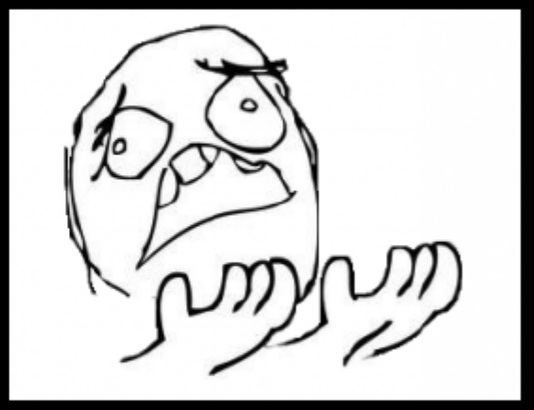


2. Migrating running code to Haskell

- hundreds of different requests (one for each write action)
- had to ensure that each one:
  - performed well enough
  - gave exactly the same answers
    - (otherwise we introduce false positives/negatives)
- As each request type is ready, we want to switch it over to running on Haskell in production

```
fxlsh> Round(0.5)
1


haxlsh> round 0.5
0
```



At scale, the edge cases happen all the time

- Invalid Unicode
- Invalid arguments to primitives & library functions
- Exception behaviour, values of exceptions
- Is “NaN” a valid number in JSON? What about “infinity”?
- Floating-point:
  - round 0.5
  - printing floating-point values
- divide-by-zero throws in FXL
- semantics of \s in regex with Unicode
- etc. etc.

3. Migrating the users

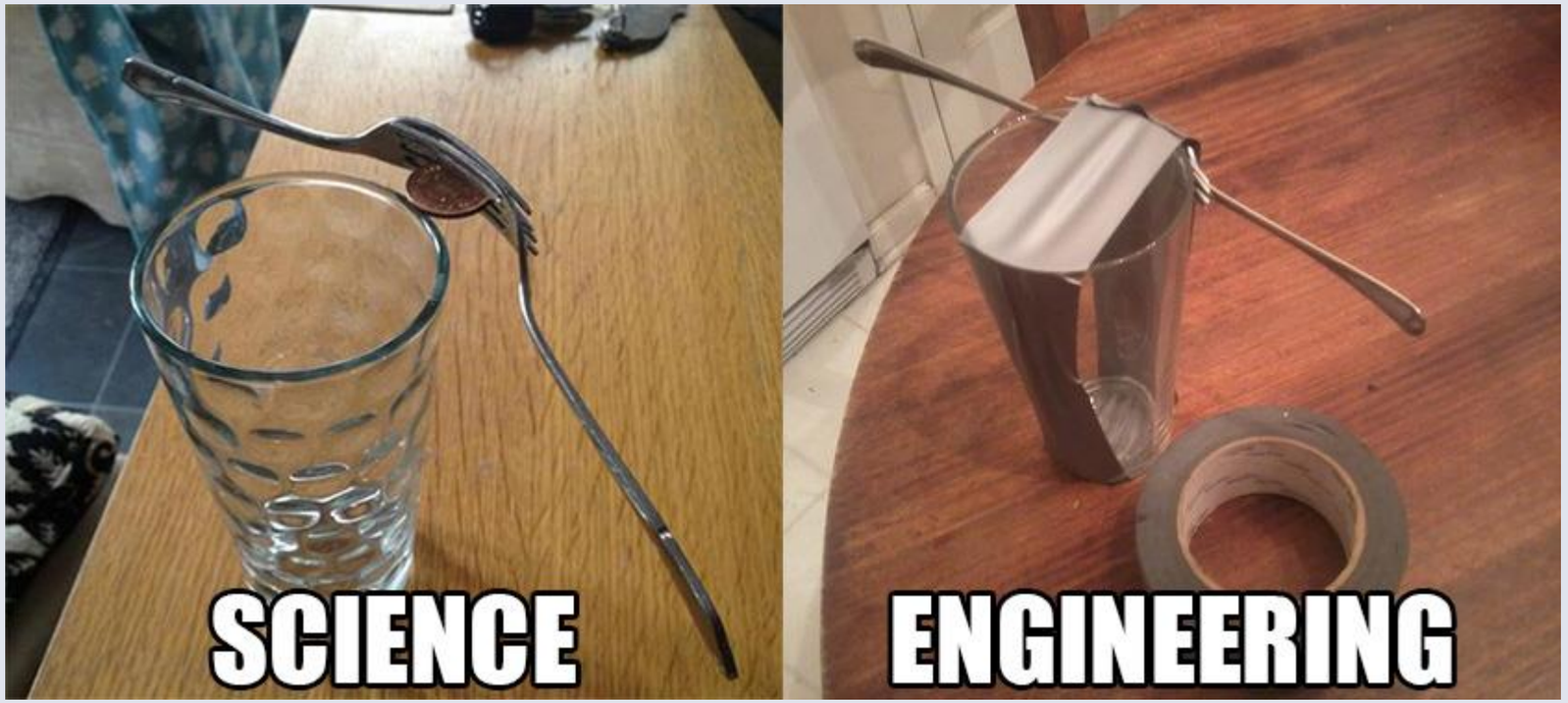
- Dozens of FXL users in multiple geographical locations
- Wrote a lot of teaching material
- Ran multi-day hands-on workshops
- Created internal Facebook group for questions (“Haxl Therapy”)
- Haxl team helped with code reviews



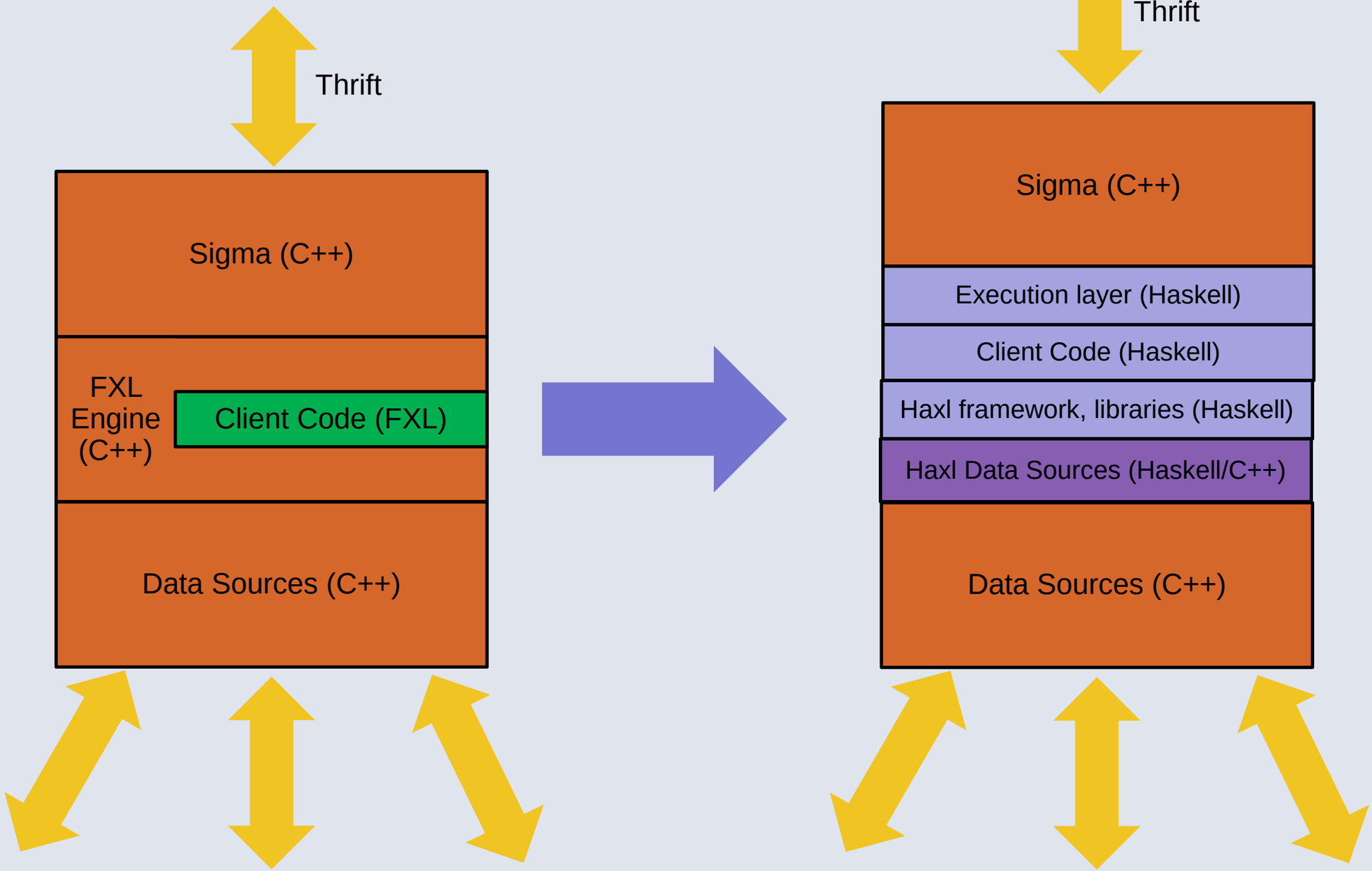
How did users cope with the switch?

- Still committing happily
- Some struggles with do-notation vs. fmap, <$>, =<<
  - “How do I convert a Haxl t to t?”
- Users started embracing the new features
  - Started building abstractions, adding types, creating tests
- Unblocks some large-scale rewrites and redesigns of subsystems

Does it *work*?

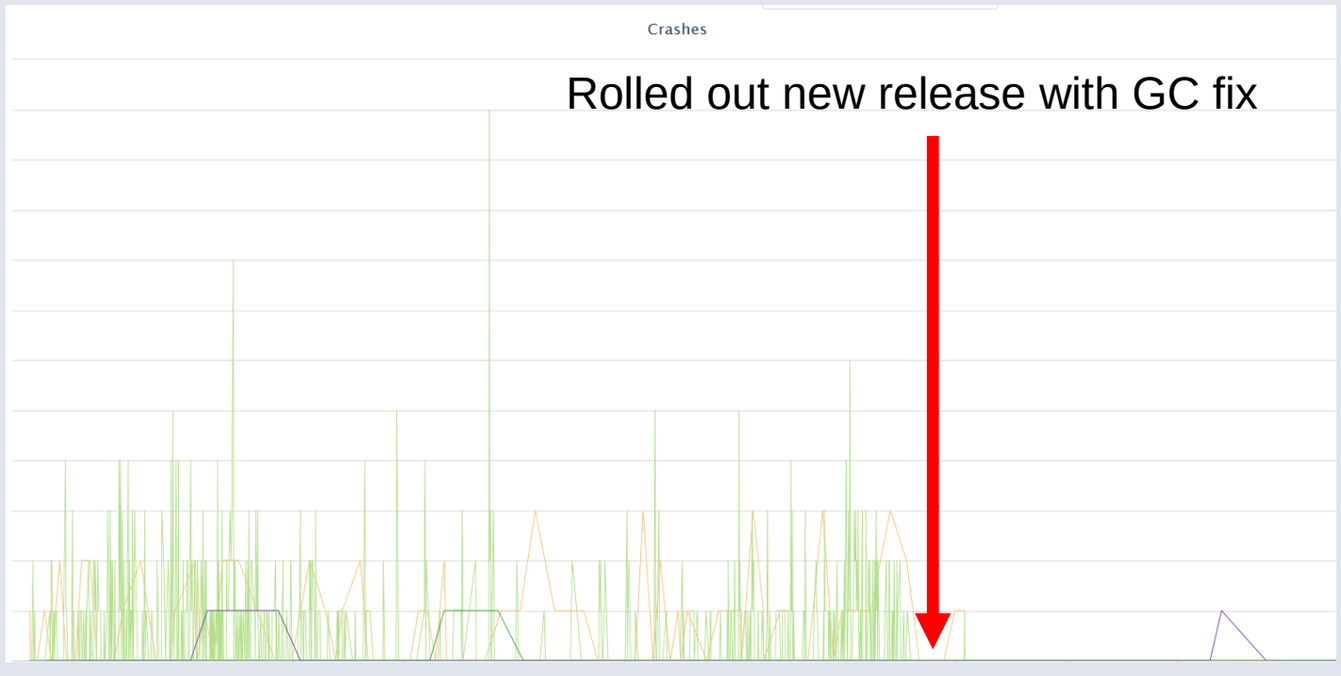


- Is it stable enough?
- Is performance good enough?
- Did we have to hack the compiler?
- What about build systems, packages, cabal hell, etc. etc.
- How do we do live updates?


Thrift
Sigma (C++)
FXL Engine (C++)
Client Code (FXL)
Data Sources (C++)
Thrift
Sigma (C++)
Execution layer (Haskell)
Client Code (Haskell)
Haxl framework, libraries (Haskell)
Haxl Data Sources (Haskell/C++)
Data Sources (C++)

Found one bug in the GHC garbage collector

- There was a multi-year-old bug in the GC that caused our machines to crash every few hours under constant load.



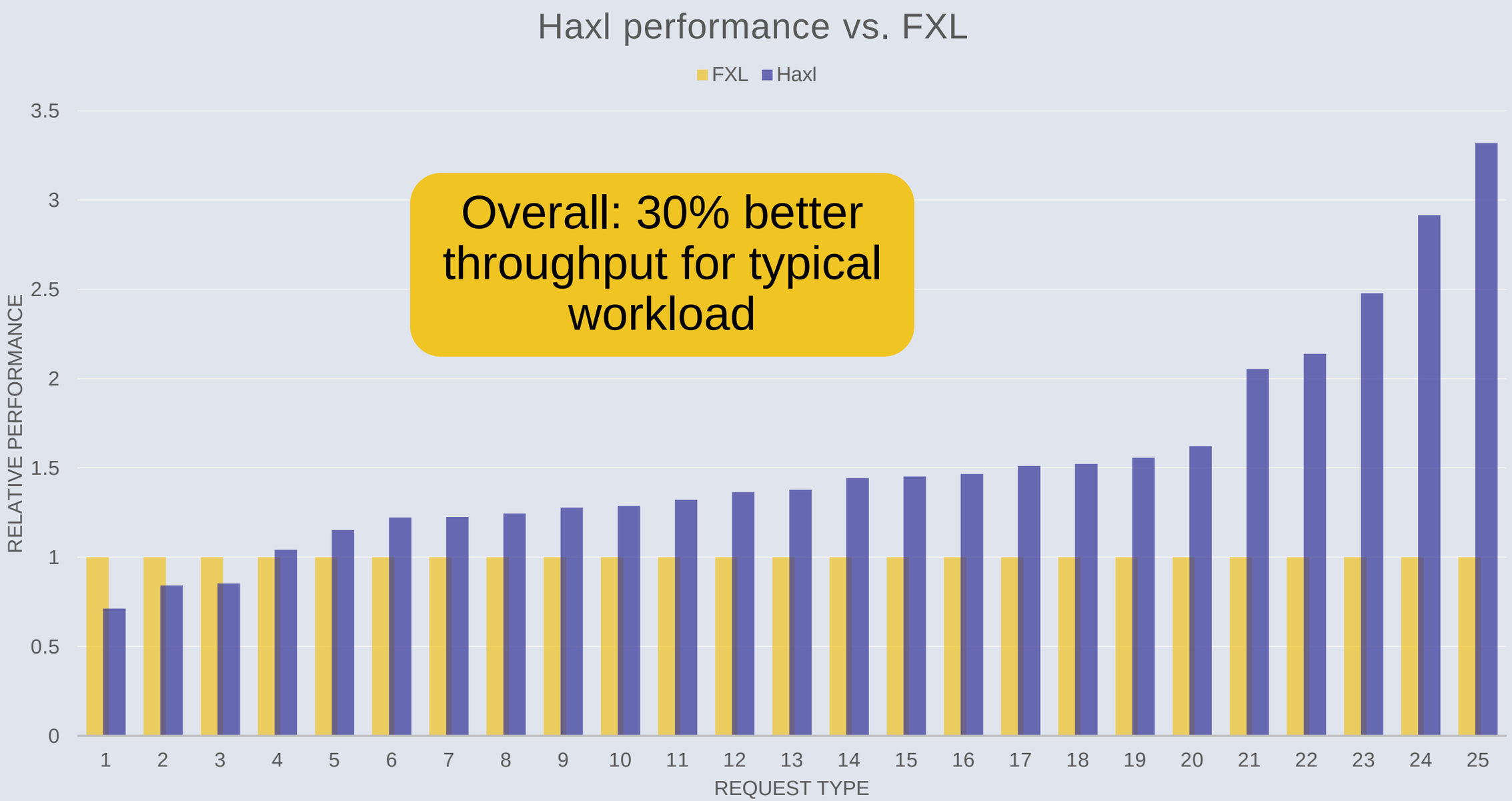
- ~~This is the only runtime bug we've found~~
  - (we found one more that only affected shutdown)

Stability

- The Haskell code just doesn’t crash. (*)
  - which is good, because diagnosing a crash in Haskell is Very Hard
- (*) except for FFI code
  - One FFI-related memory bug was particularly hard to find

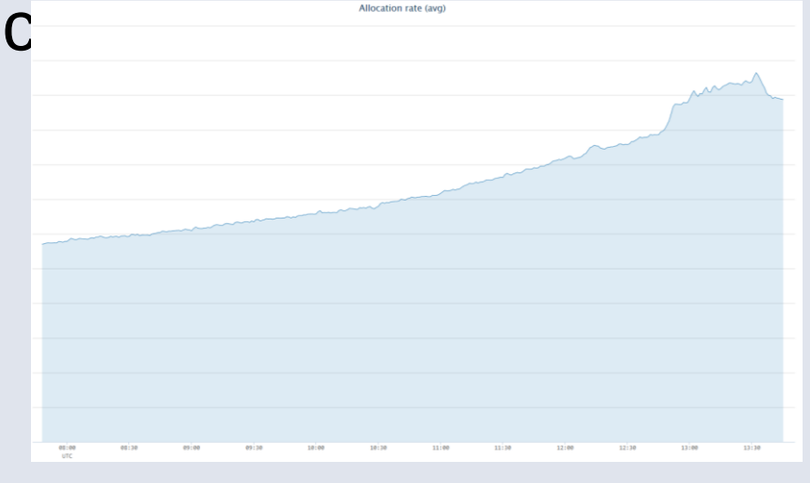
Performance

Performance



Good monitoring is essential

- Noisy environment:
  - multiple sources of changes
  - dependencies on other services
  - traffic isn’t constant
  - spam/malware attacks are bursty
- Hooked up GHC’s garbage collector stats API to Facebook’s monitoring infrastruc





Resource limits

- Server resources are finite, and we have latency bounds
- Our job is to keep the system responsive, so we cannot allow individual requests to hog resources
- Fact of life: individual requests *will* sometimes hog resources if allowed to
  - Usually: doing some innocuous operation on untypically large data
  - e.g. regex engines sometimes go out to lunch

Allocation limits

- So we implemented allocation limits in GHC

```
setAllocationCounter    :: Int64 -> IO ()
getAllocationCounter    :: IO Int64

enableAllocationLimit   :: IO ()
disableAllocationLimit :: IO ()
```

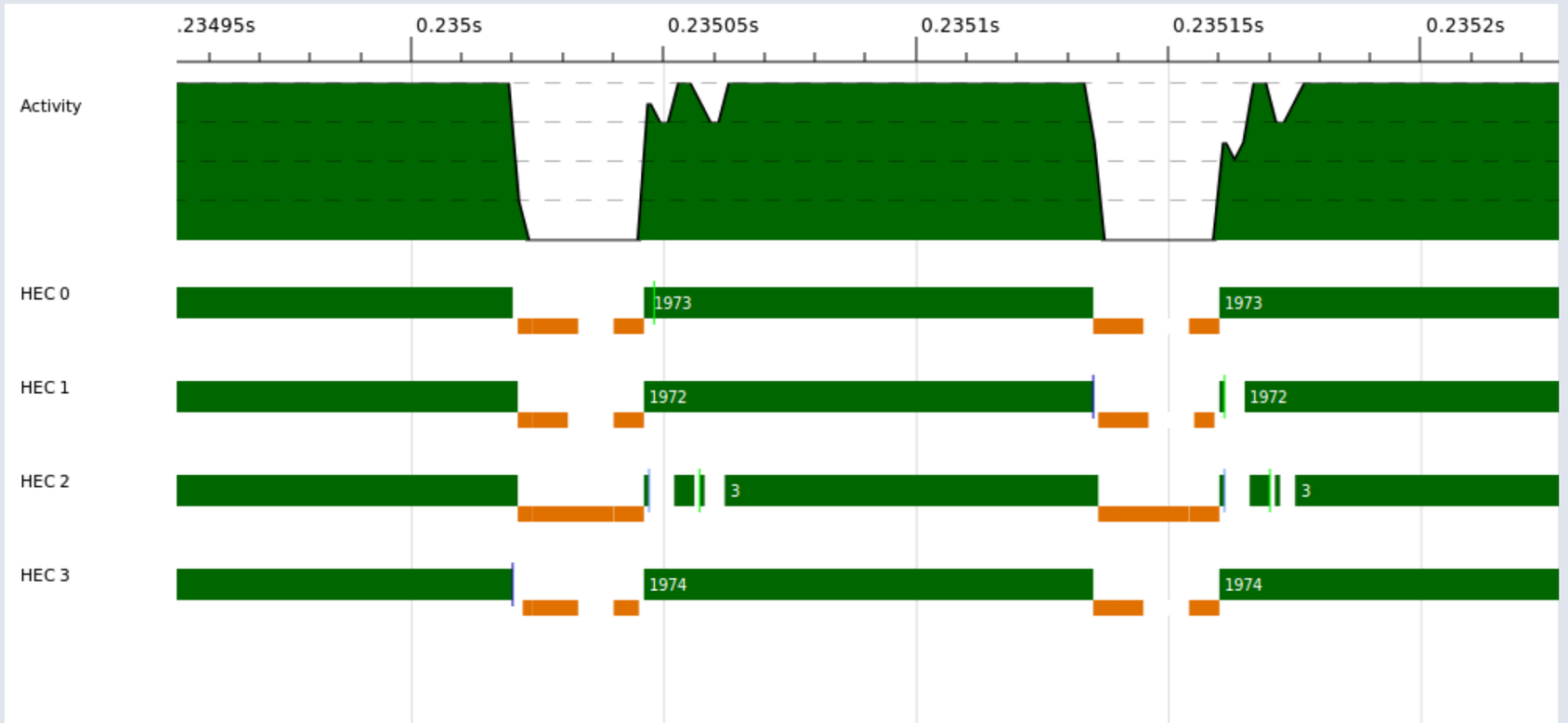
- Counter is per-thread, ticks down with memory allocation
- Triggers an exception when the counter reaches zero
- Easy in Haskell, very difficult in C++

How important are allocation limits?



GC vs. latency

- GHC has a stop-the-world parallel collector



- obviously to meet our latency goals we cannot GC for too long

GC vs. latency

- So how do we manage this?

- Fixed number of worker threads + allocation limits
  - effectively puts a bound on the amount of work we are doing at any given time

- Very little persistent state (a few MB).

- A handful of GC improvements, all upstreamed

Hot-swapping code

Fast deployment

- The faster we can get new rules into production, the more spam we catch before people see it, the faster we stop viral malware, etc.
- (Not all changes need immediate deployment: code review is the norm)
- “Code in the repo is what is running in production”
- Deployment typically on the order of a few minutes

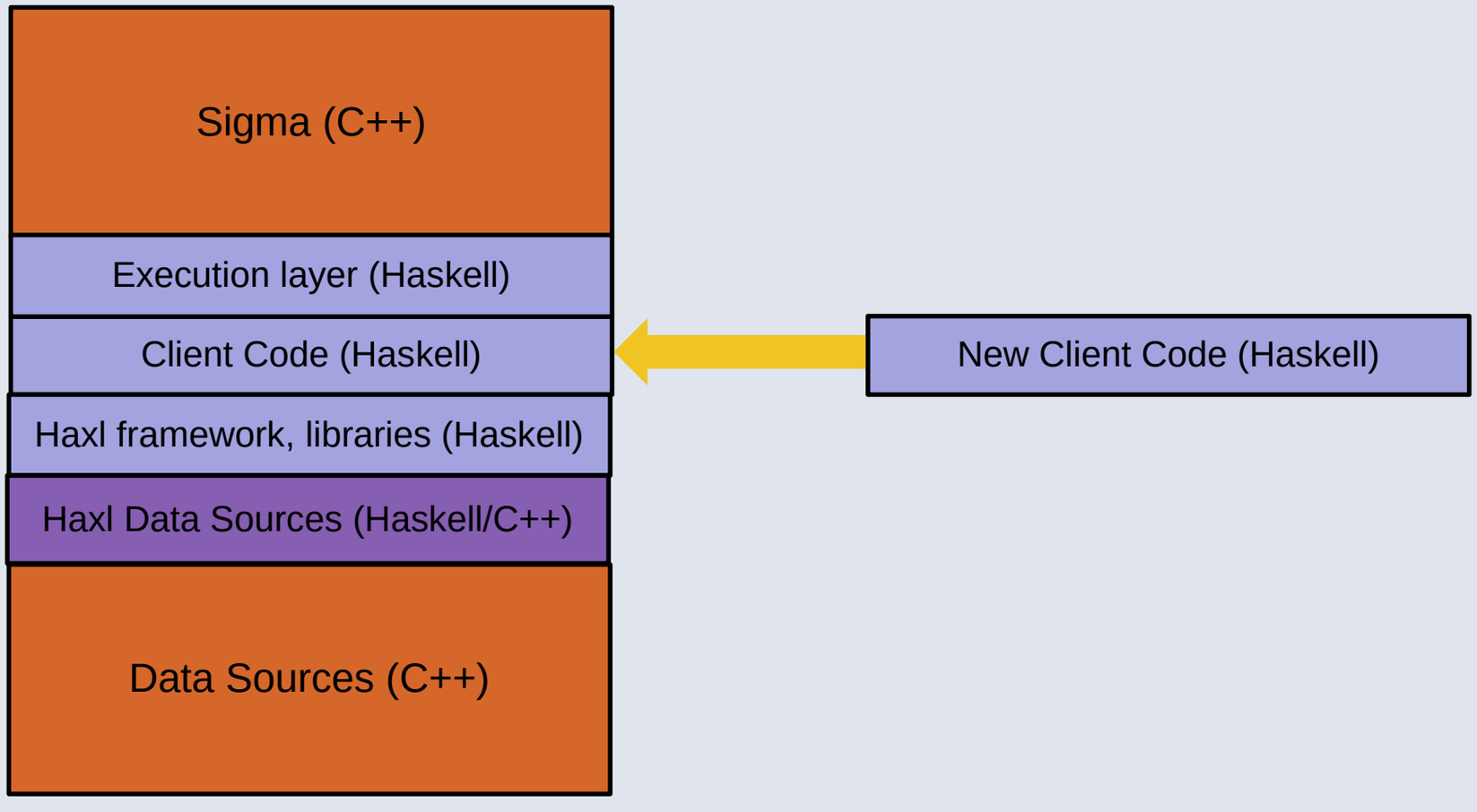
How can we deploy new code?

- Haskell has an optimising compiler, runs native code
- Haskell code needs to be compiled and distributed to servers
- Servers need to start running the new code somehow

Restarting the process doesn’t work

- Takes a while to start up
- Caches would be cold
- A rolling restart would take too long

Live updates



Live Updates

- Main idea
  - load the new code directly into the running process
    - needs a dynamic linker
  - Start taking requests using the new code
    - When all requests running on the old code have finished, remove it from the process

- GHC’s runtime has a built-in linker
- We added support for unloading objects with GC integration

Other things

Don’t have time to talk about...

- Build systems
  - (we use Stackage LTS + cabal + FB build system)
- FFI with C++
  - Wrote a tool to mangle C++ function names
  - Don’t forget: catch all your exceptions in C++
- Customized GHCi
  - we built our own customized GHCi that uses the Haxl monad by default, and has Hoogle integration for our codebase

facebook

Questions?

The Haxl Team, past and present

Jonathan Coens
Bartosz Nitka
Aaron Roth
Kubo Kováč
Katie Miller
Jon Purdy
Zejun Wu
Jake Lengyel
Andrew Farmer
Louis Brandy
Noam Zilberstein
Simon Marlow