Retcon

Imposing eventual consistency on disparate data sources

Thomas Sutton

2015-05-21

Introduction

Overview

1. I'll introduce the **business problem**;
2. Sketch the **architecture**;
3. Describe **version one** and some of its **shortcomings**;
4. Describe **version two**; and
5. Note some of the **lessons** I learned.

Business Problem

Context

Anchor is a hosting company and, like many companies, we have **lots of systems** which **share data** used to operate our services:

- Customer details
- User accounts
- Service details
- Usage and billing details
- Configuration

Some have **no system of record** and others do, but it's too **slow**, too **unreliable**, or too difficult to use directly.

Problem

As a result many of our systems maintain their own private copies of the data they need:

lots of copies of lots of data in lots of places

Worse, many of these systems will happily update their own local copies!

lots of **mutable** copies of lots of data in lots of places

We've historically used a range of caches, batch updates, and manual processes to keep this all up to date, but that isn't really scalable.

We created Retcon to help fill this gap.

Goals

Our goals for the project were:

1. To *reduce coupling* between systems;
2. To *replace* existing application-specific solutions;
3. To *propagate changes quickly* between systems; and
4. Do so safely, reliably, and automatically.

Requirements

- **Generic in the middle** - Retcon shouldn't need any application-specific logic or special cases; it's all just data.
- **Flexible at the boundaries** - Retcon will interact with systems outside of our control and will have to use awkward interfaces.
- **Reliable** - If it can't do the right thing in every case, at the very least Retcon shouldn't do the wrong thing.
- **Automatic** - Retcon will replace existing manual and semi-manual process; it should require *less* manual processing than the previous systems.
- **Timely** - Retcon should propagate changes between systems in a timely fashion.

Solution

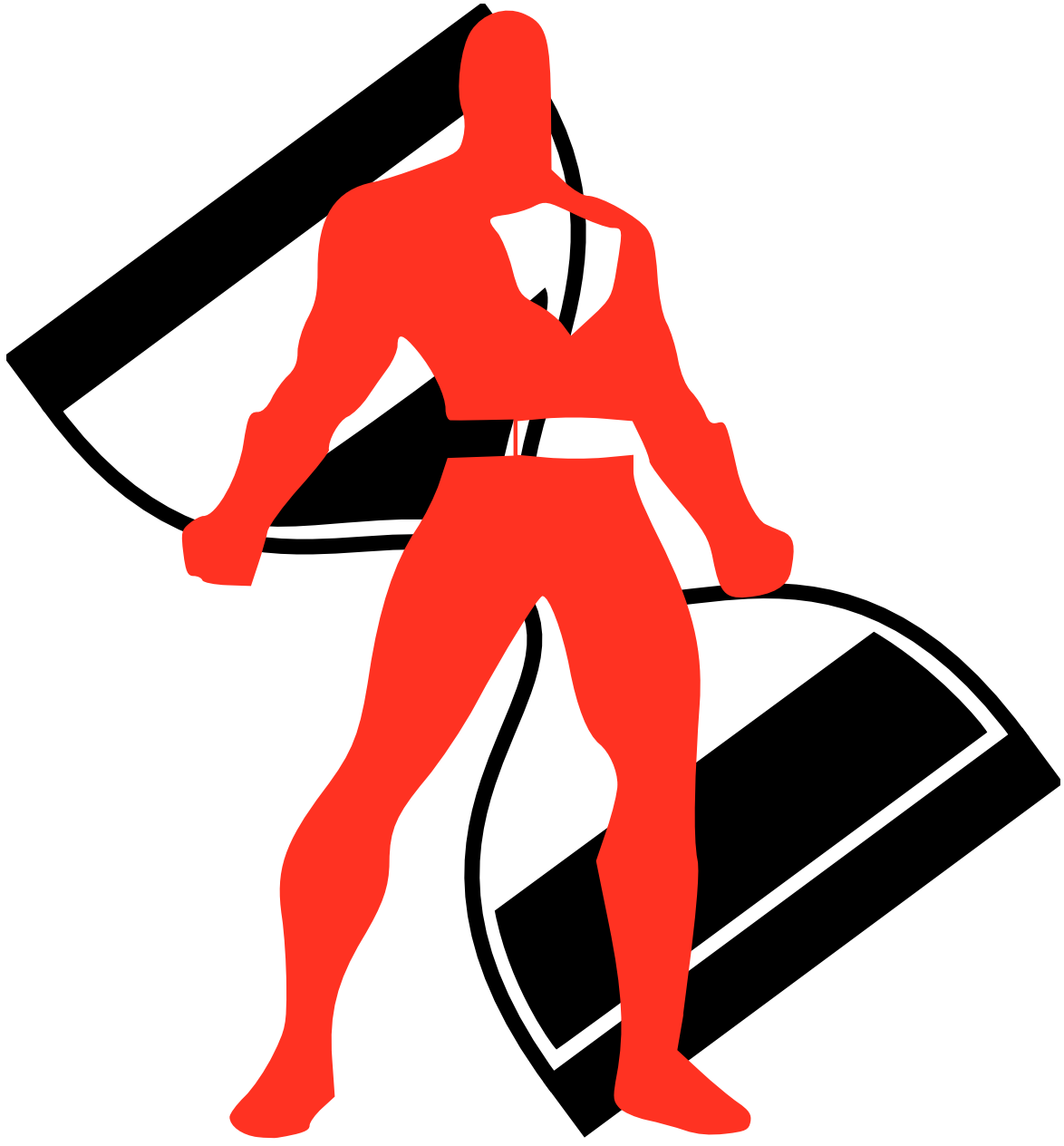


Figure 1:

Architecture

Data Model

- An **entity** is a type of data. Examples might include `customer`, `user`, `invoice`, `domain name`, `hosting account`, or `cloud service`.
- A **data source** is an external system which stores data from some entity. Examples might include CRM, ERP, and accounting systems, control panels, monitoring and alerting systems, etc.

- An **internal key** uniquely identifies a document within Retcon itself.
- A **foreign key** uniquely identifies a particular document in a particular data source.
- A **document** is a particular datum which should be kept up to date across all data sources for an entity. Examples might include the customer details of "Thomas Sutton".

Data Model

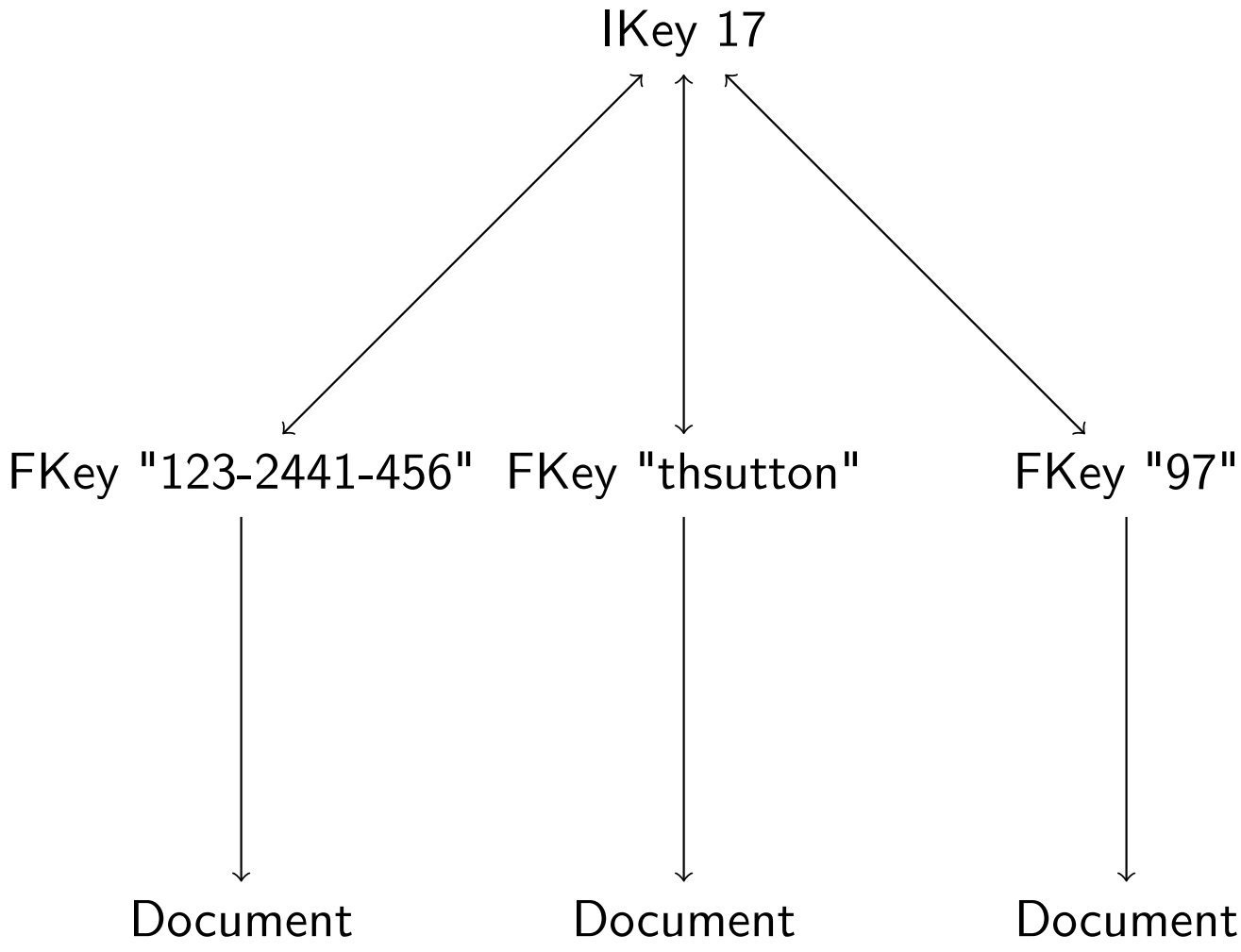


Figure 2: Example data from the User entity with three data sources

Data Model

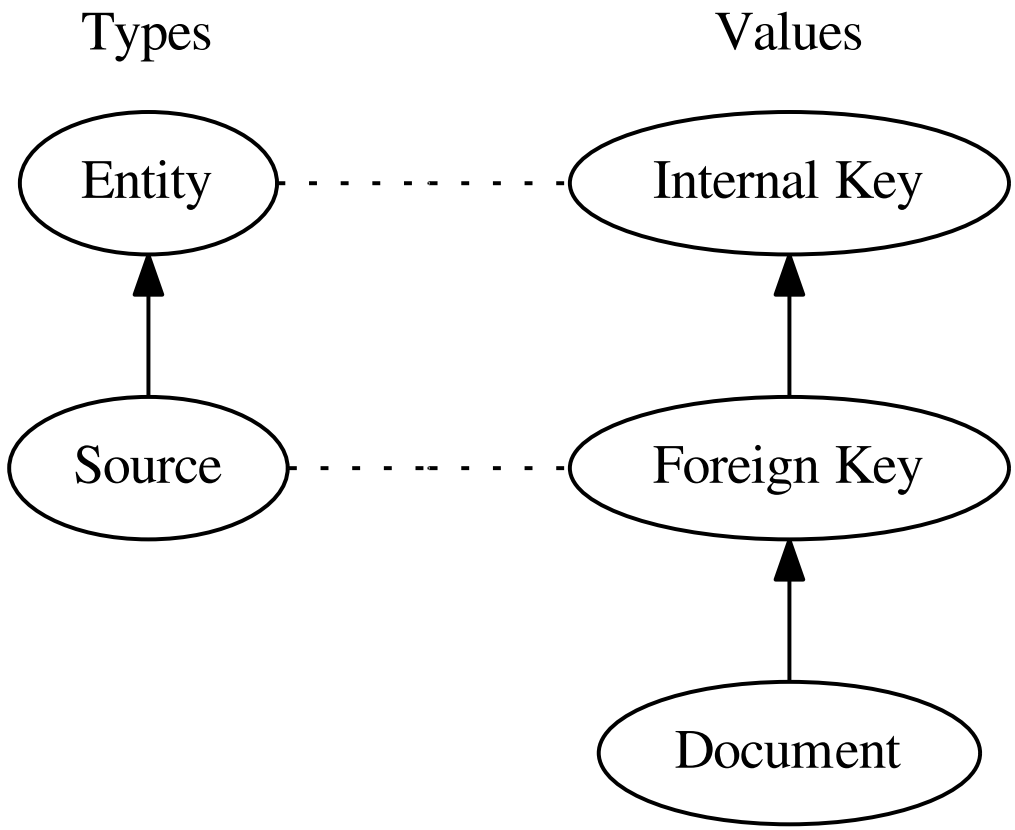
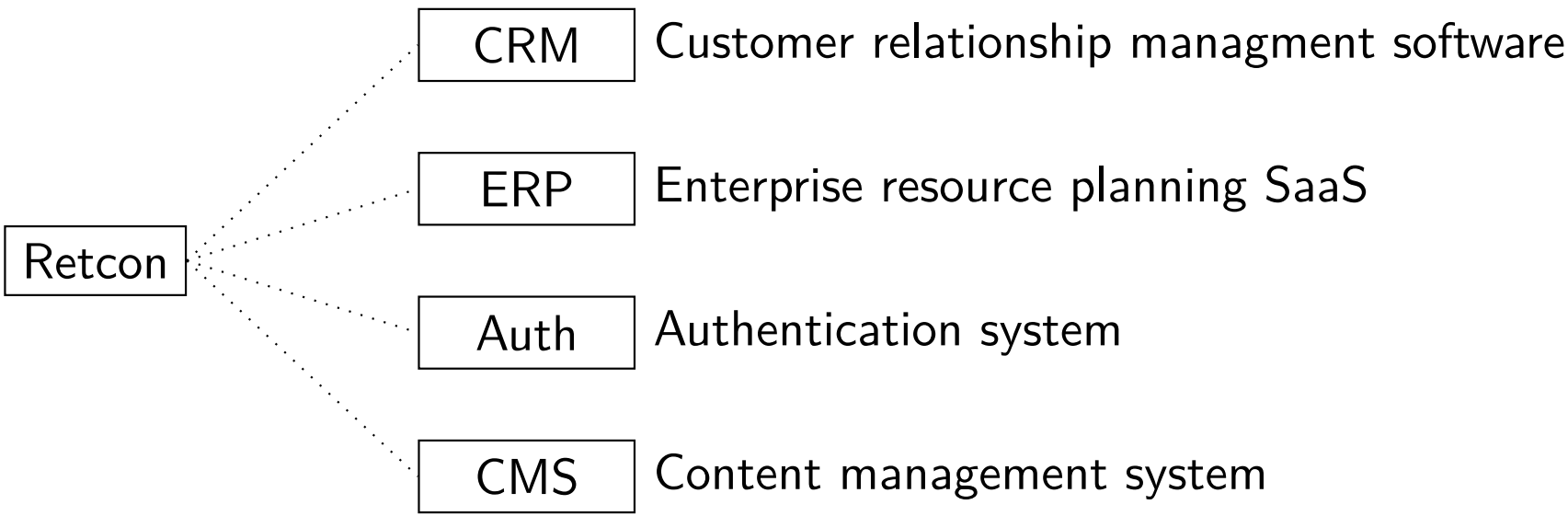


Figure 3: Internal Keys are for Foreign Keys as Entities are to Data Sources

Operation

Here's a simple configuration in which Retcon synchronises data for the User entity between four data sources:



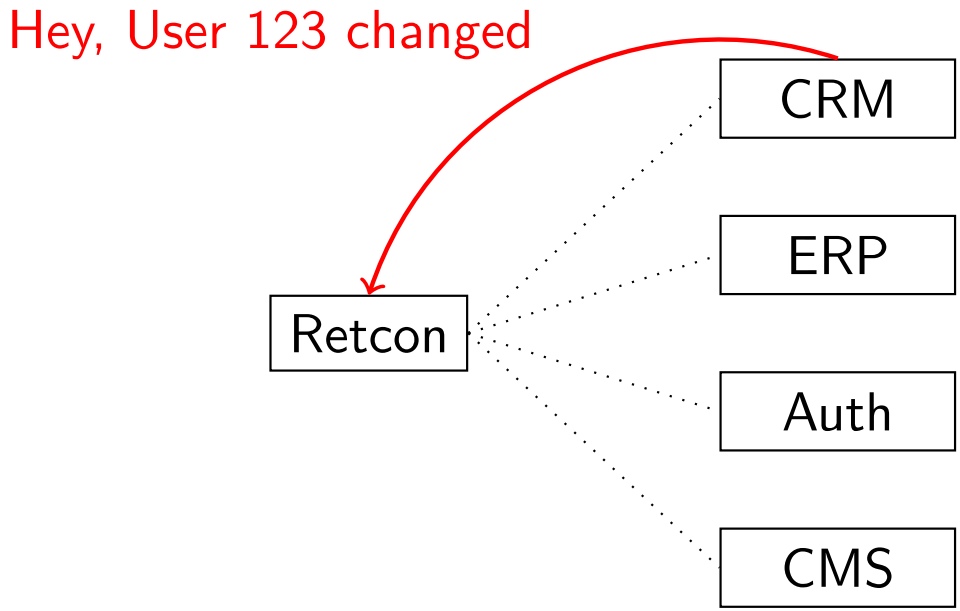
Operation

The Retcon "algorithm":

1. A data source notifies Retcon that some document has change.
2. Retcon determines what sort of even occurred and how to respond:
   - 2.1 A CREATE event allocates a new internal key and is propagated to the other data sources.
   - 2.2 A DELETE event is propagated to the other data sources and the internal key and related resources are deleted.
   - 2.3 An UPDATE event triggers a diff/merge/patch process.

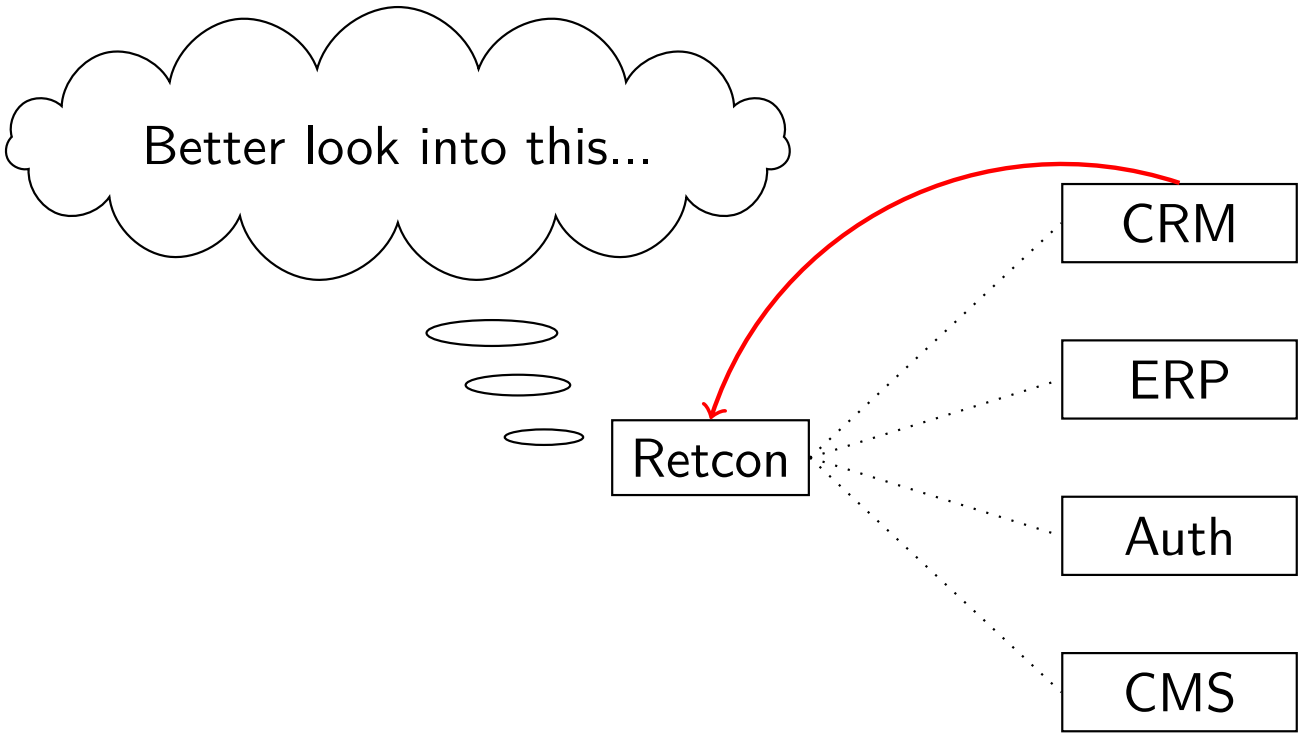
Step 1 - Notification

The process begins when a data source reports an event to Retcon:



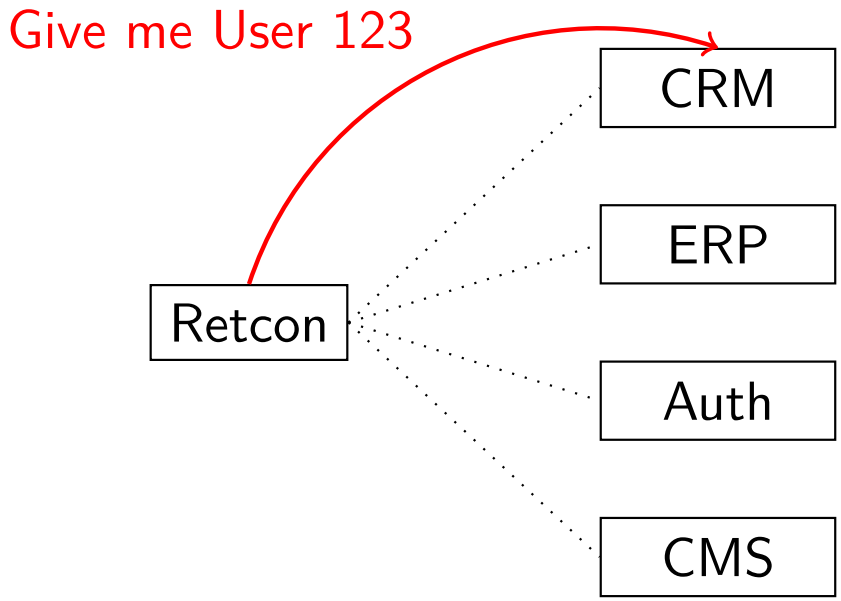
Some of our data sources can do this immediately in response to an event, others use a cron job. This is one place where “eventual” creeps in.

Step 2 - Determination



Step 2 - Determination

Retcon needs two pieces of information to determine how to respond: the *internal key* and the *document*. It can find the internal key which corresponds to the foreign key in its database but has to ask the data source for the document:



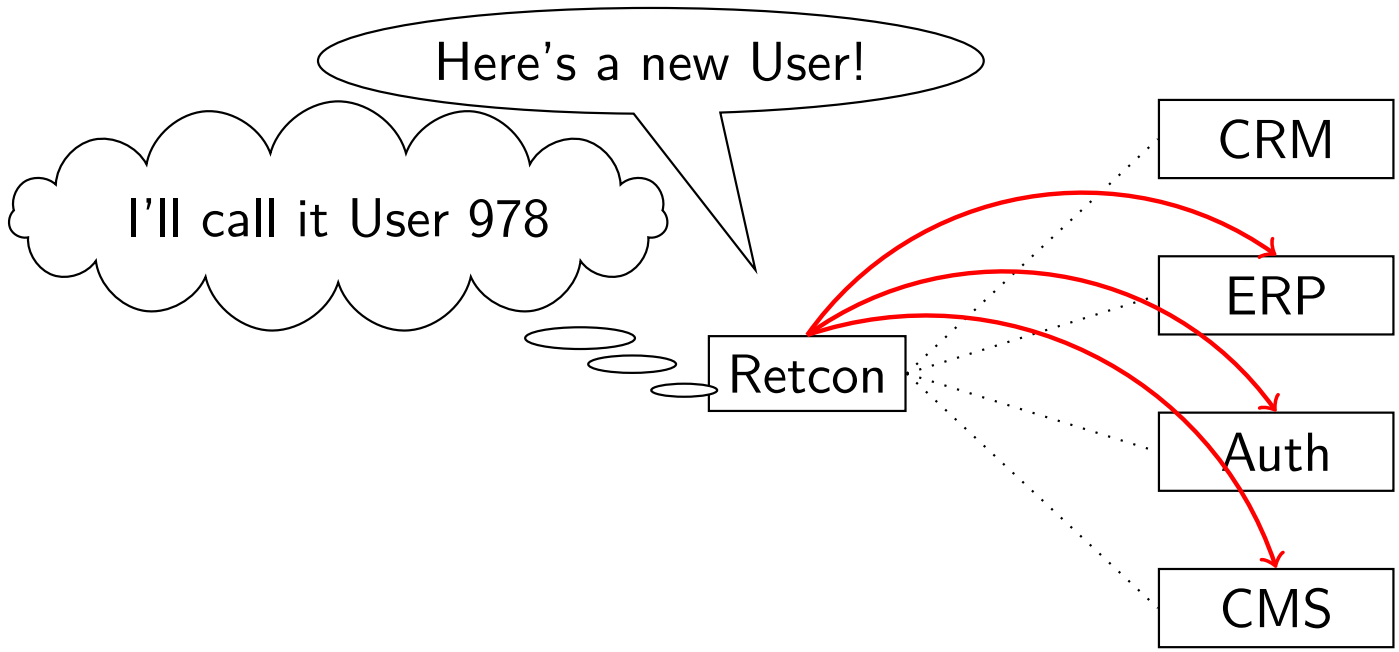
Step 2 - Deciding what to do

Once we have a notification for a *foreign key* and retrieved the *document* from the data source there are ~~three~~ four possible responses Retcon can make:

Document	Key	Operation
Present	Unknown	Create
Present	Known	Update
Missing	Known	Delete
Missing	Unknown	Error

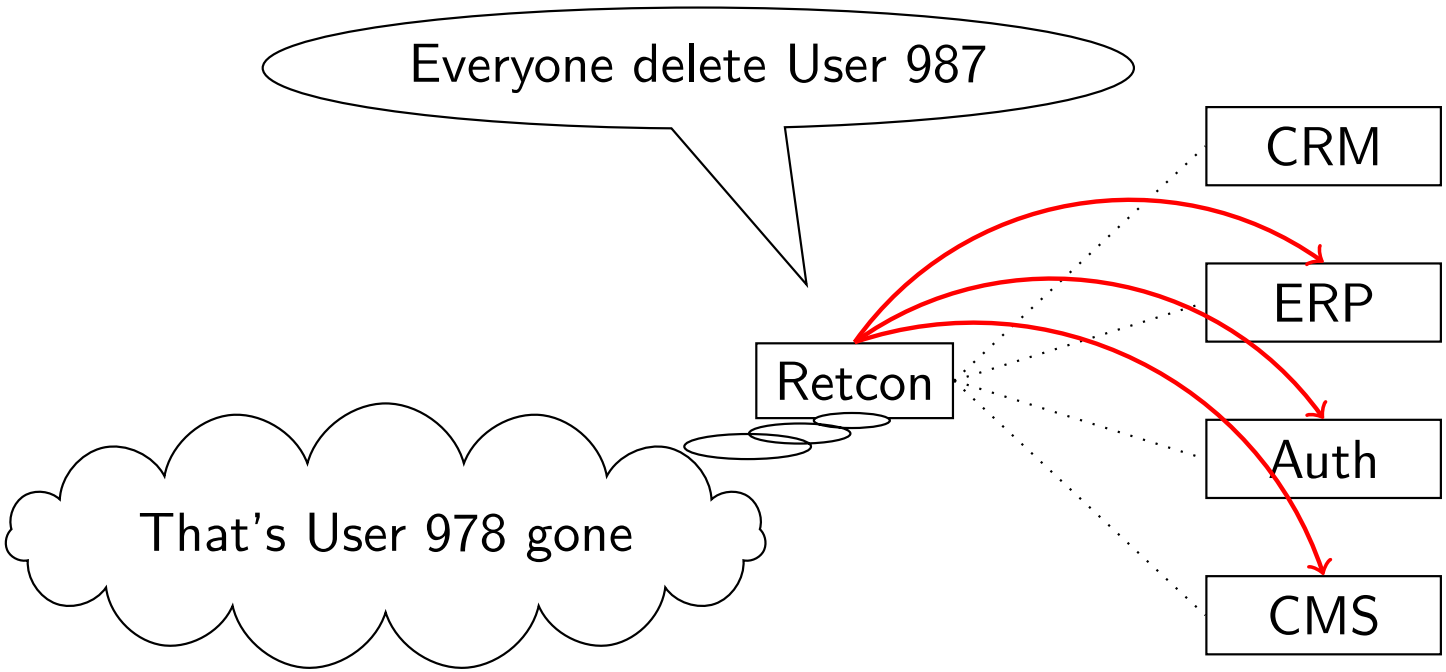
Step 2a - Creation events

If it's a create event, Retcon just allocates a new *internal key* and creates the new document in each data source, recording the *foreign keys* they give back.



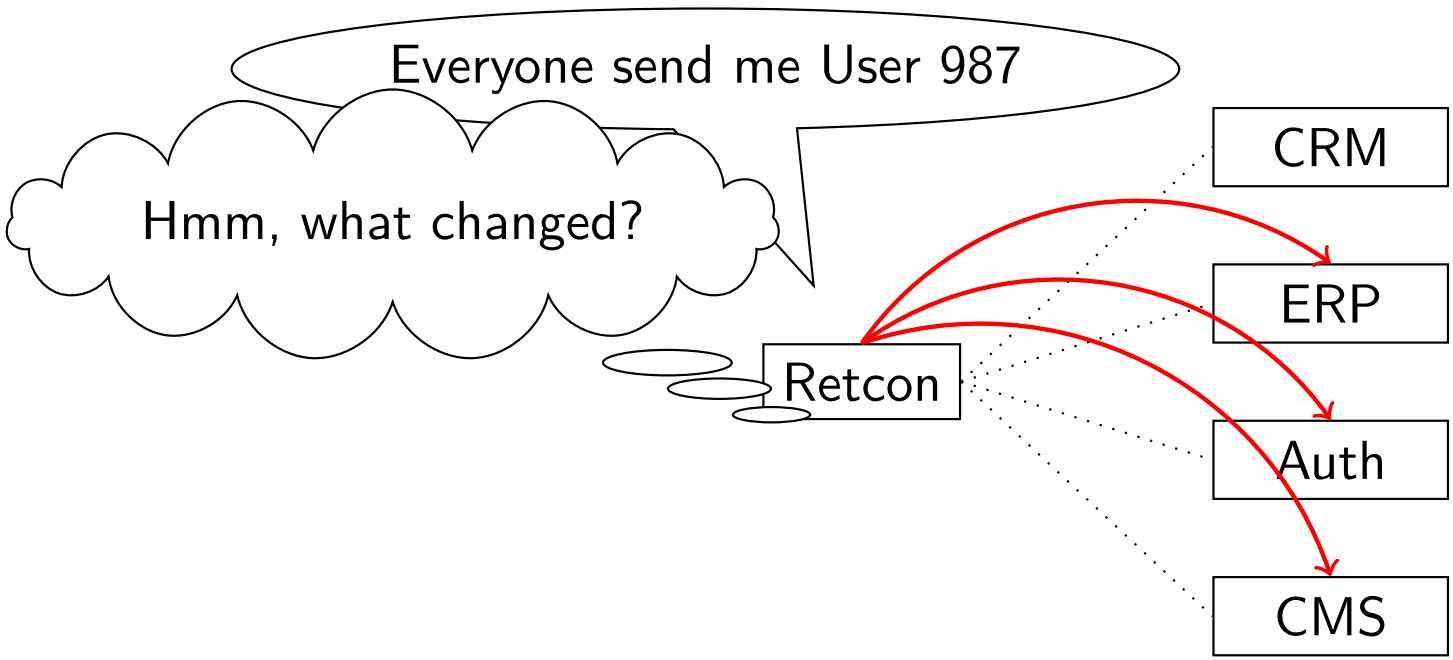
Step 2b - Deletion events

If it's a delete event, Retcon just instructs each data source to delete the corresponding document and then removes the *internal key* and associated records from its database.



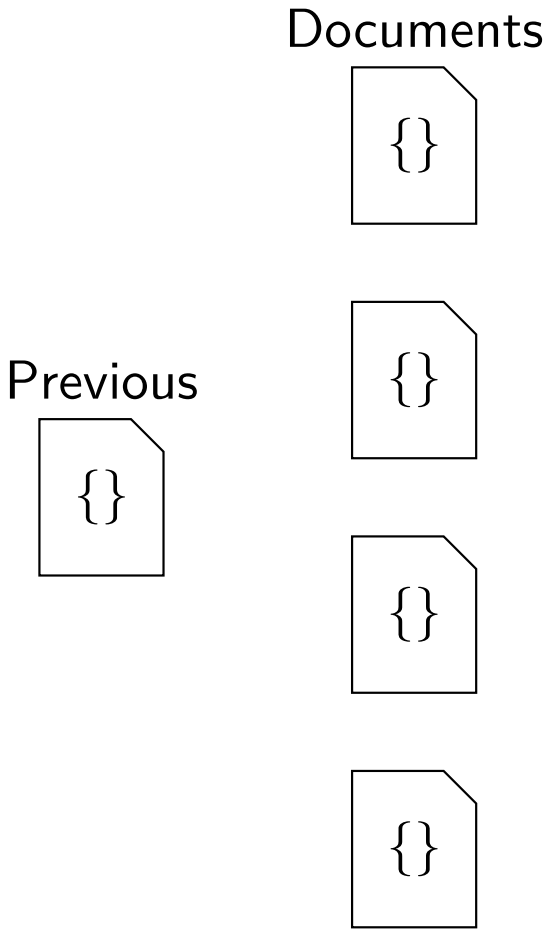
Step 2c - Update events

If it's an update operation, Retcon fetches the corresponding documents from the other data sources and then applies its core algorithm to the set of documents:



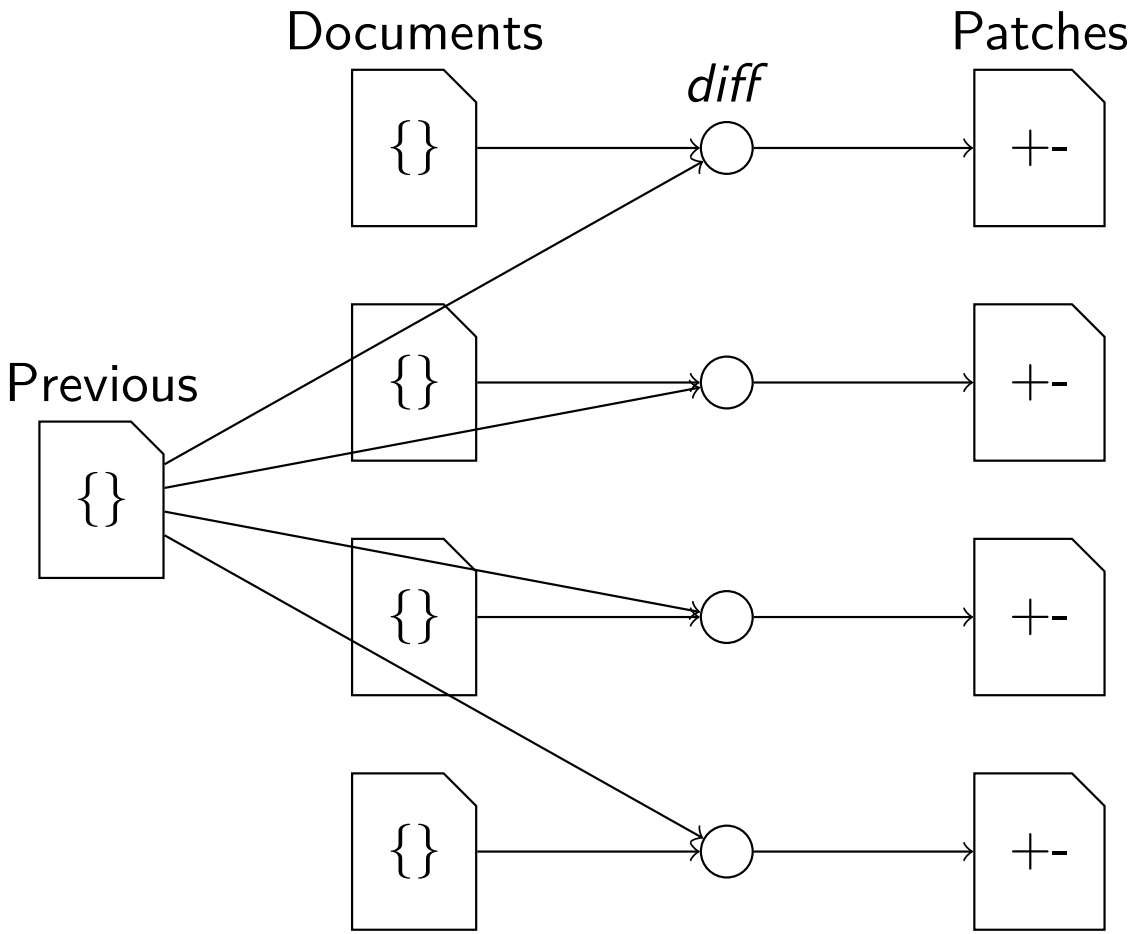
Step 2c - What changed

We start with a "previous" document (which we trust was consistent at some point in the past) and the current documents we fetched from the data sources:



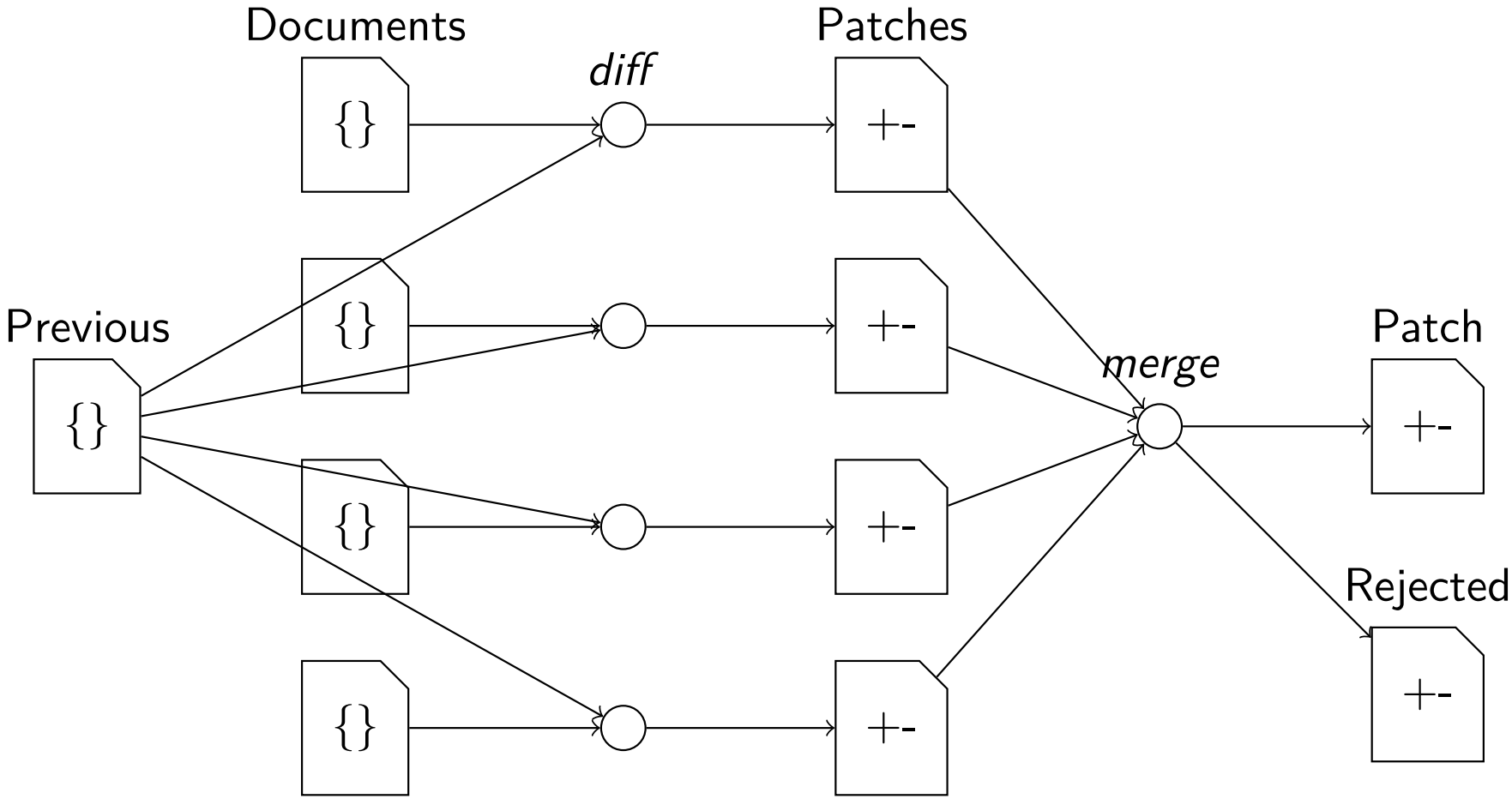
Step 2c - What changed

Then we compare the previous document with each of the data source documents to generate a patch for each.



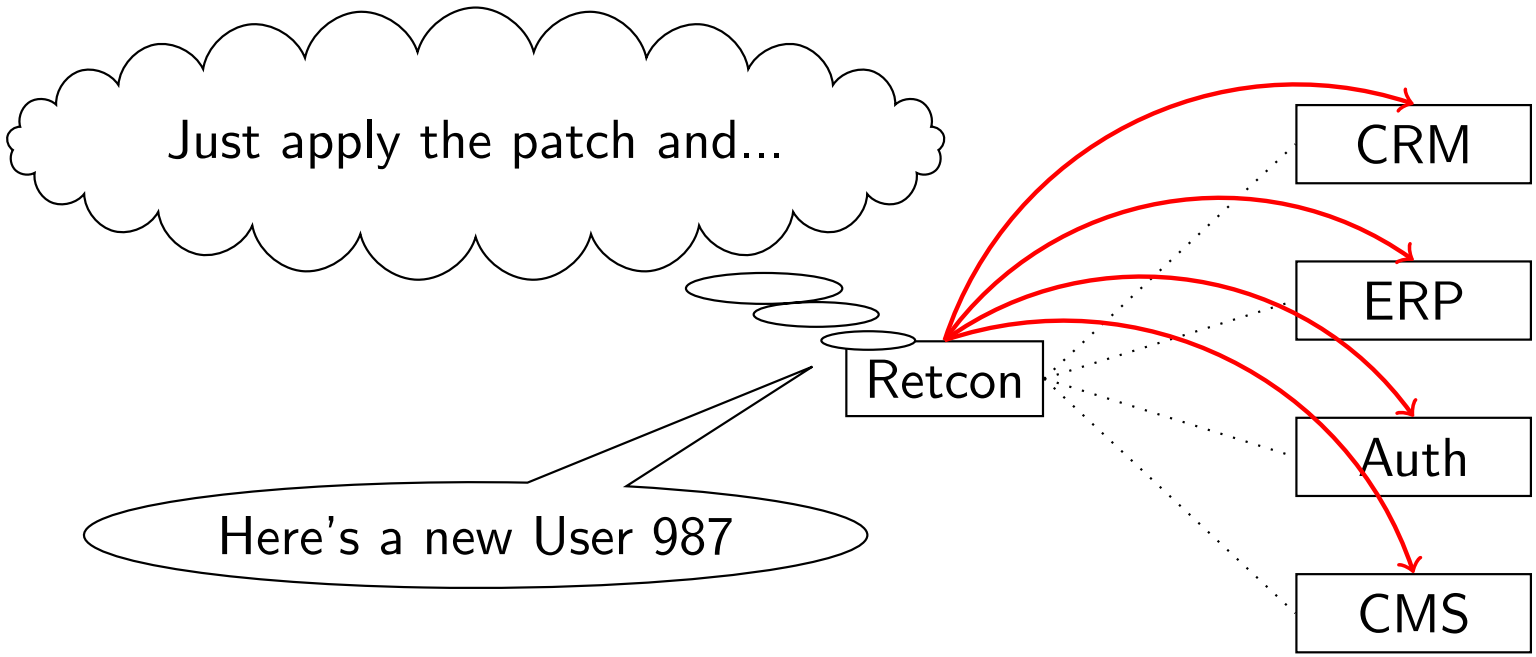
Step 2c - What changed

Finally Retcon merges these patches according to a policy. The result is a patch we can apply to the data sources and some rejected changes. These latter are kept for manual resolution.



Step 2c - Update events

Once it has a patch describing the changes to distribute, Retcon applies it to each document (including the "previous" document, which it'll keep for next time) and sends the updated documents back to the data sources.



Operation - Pseudocode

The table of conditions translates fairly directly into a Haskell case:

```
process :: ForeignKey -> m ()
process fkey = do
    ikey <- lookupInternalKey fkey
    doc  <- getDocument fkey
    case (ikey, doc) of
        (Just ikey', Just doc') -> processUpdate ikey'
        (Just ikey', Nothing  ) -> processDelete ikey'
        (Nothing   , Just doc') -> processCreate fkey doc'
        (Nothing   , Nothing  ) -> reportError   fkey
```

Operation - Pseudocode

The create and delete operations too:

```
processCreate :: ForeignKey -> Documents -> m ()
processCreate fkey doc = do
  ikey <- allocateInternalKey fkey
  forM_ datasources $ \src -> do
    fkey <- createDocument src doc
    recordForeignKey ikey fkey

processDelete :: InternalKey -> m ()
processDelete ikey = do
  forM_ datasources $ \src -> do
    fkey <- lookupForeignKey src ikey
    deleteDocument src fkey
  deleteInternalKey ikey
```

Operation - Pseudocode

```
processUpdate :: InternalKey -> m ()
processUpdate ikey = do
  -- Fetch the documents.
  initial <- lookupInitialDocument ikey
  datas <- forM datasources $ \src -> do
    fkey <- lookupForeignKey src ikey
    doc <- getDocument src fkey
    return ((src, fkey), doc)

  -- Extract and merge patches.
  let patches = map (diff initial) $ catMaybes (map snd datas)
      (accepted, rejected) = merge patches

  -- Update state.
  saveConflicts ikey rejected
  recordInitialDocument ikey (patch initial accepted)

  -- Update data sources.
  forM_ datas $ \((src, fkey), doc) -> do
    updateDocument fkey (patch doc accepted)
```

Version One

Overview

- Implemented as a library; just plug-in implementations of your entities and data sources, compile, and deploy!
- Data sources (i.e. code to interface with external systems) executed within Retcon.
- Entity and data source code implemented as type class instances.
- Phantom types to make sure keys and documents only passed to correct back-ends, etc.
- The phantom type arguments just identify an entity or data source. So use GHC's type-level literals. One less type for the client to define!

Structure

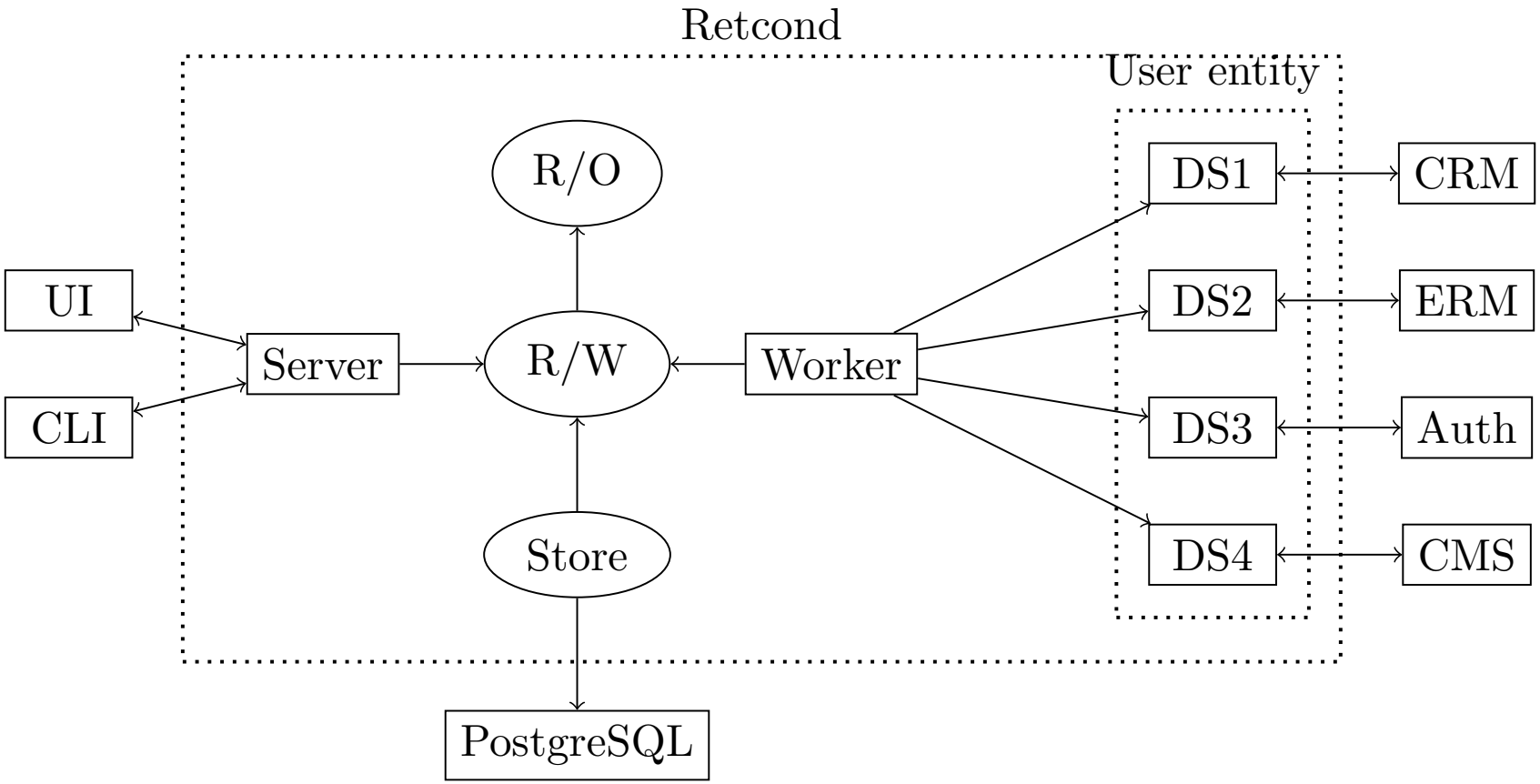


Figure 4: Structure of Retcon 1.x

Structure

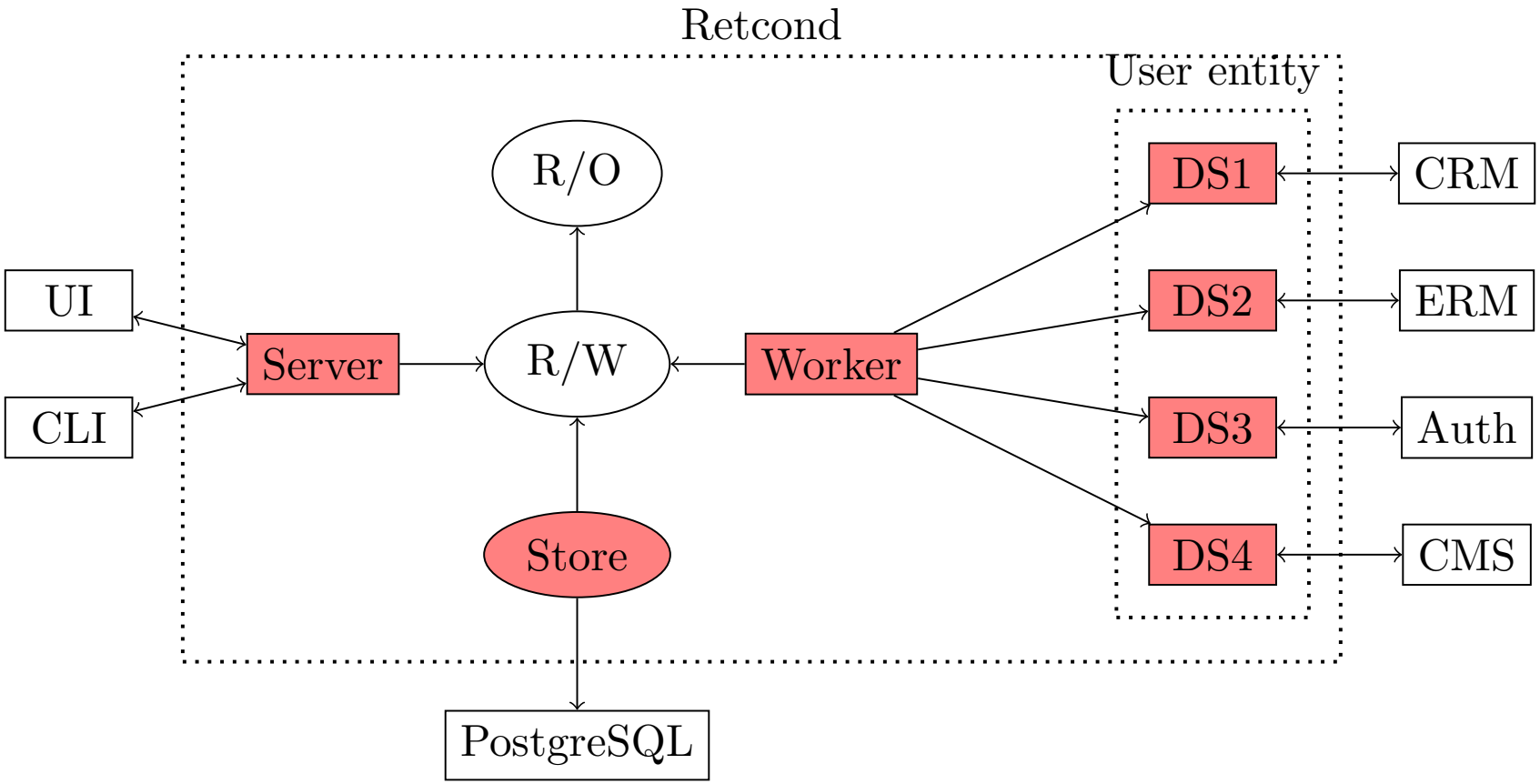


Figure 5: Structure of Retcon 1.x - Stateful components highlighted

Type Literals

Our phantom types will be used with an open set of arguments (the names of the entities and data sources the library is used with) and we'll need to operate on these types (to dispatch operations to the right data source). So we'll use GHC's type-level symbols.

```
{-# LANGUAGE DataKinds #-}
import GHC.TypeLits

ik1 :: IKey "user"
fk2 :: FKey "user" "erp"
fk1 :: FKey "user" "crm"
ik1 = IKey 42
fk1 = FKey "thsutton"
fk2 = FKey "5f8bd327-c8b3-4efb-92cd-6da993112ed1"
```

Phantom types

A phantom type is a parameterised type which does not use one (or more) of the parameters:

```
newtype IKey ent      = IKey { unIKey :: Int }
newtype FKey ent src = FKey { unFKey :: ByteString }
```

It's a type error to use an `IKey` or `FKey` for one entity or data source where another is expected.

We don't have *values* of these phantom types (that's the point!) but still need to pass them around some times. So `ScopedTypeVariables` and `Data.Proxy`:

```
data Proxy t = Proxy
```

Type classes

Each entity and data source is represented by a unique type, so we implemented them as typeclasses:

```
class RetconEntity ent where
    sources :: Proxy ent -> [SomeSource ent]

class (RetconEntity ent) => RetconSource ent src where
    createDoc ::                    Document -> IO (FKey ent src)
    readDoc   :: FKey ent src                -> IO Document
    updateDoc :: FKey ent src -> Document -> IO (FKey ent src)
    deleteDoc :: FKey ent src                -> IO ()
```

Existential types

But now we need to put *types* (representing, e.g., the data sources for a particular entity) into a data structure. Existential types let us do this:

```
data SomeEntity =
    forall e. (RetconEntity e) => SomeEntity (Proxy e)

data SomeSource e =
    forall s. (RetconSource e s) => SomeSource (Proxy s)

sources :: [SomeDataSource "user"]
sources =  [ SomeDataSource (Proxy :: Proxy "crm")
           , SomeDataSource (Proxy :: Proxy "erp")
           ]
```

ScopedTypeVariables and InstanceSigs

By this time we are in quite a strange place and it seemed sensible to write code like this:

```
{-# LANGUAGE ScopedTypeVariables, InstanceSigs #-}
instance RetconStore DB where
  lookupFKey :: forall ent src. (RetconSource ent src)
      => DB -> IKey ent -> m (Maybe (FKey ent src))
  lookupFKey (DB conn) ik = do
      let (ent, ik') = iKeyValue ik
          src = symbolVal (Proxy :: Proxy src)
      res <- query conn sql (ent, src, ik')
      return (FKey <$> res)
    where
      sql = "SELECT fk FROM fks WHERE ent = ? AND src = ? "
          <> "AND id = ?"
```

We're using the *return type* of the function to decide which strings to pass into the database?!?

Structure

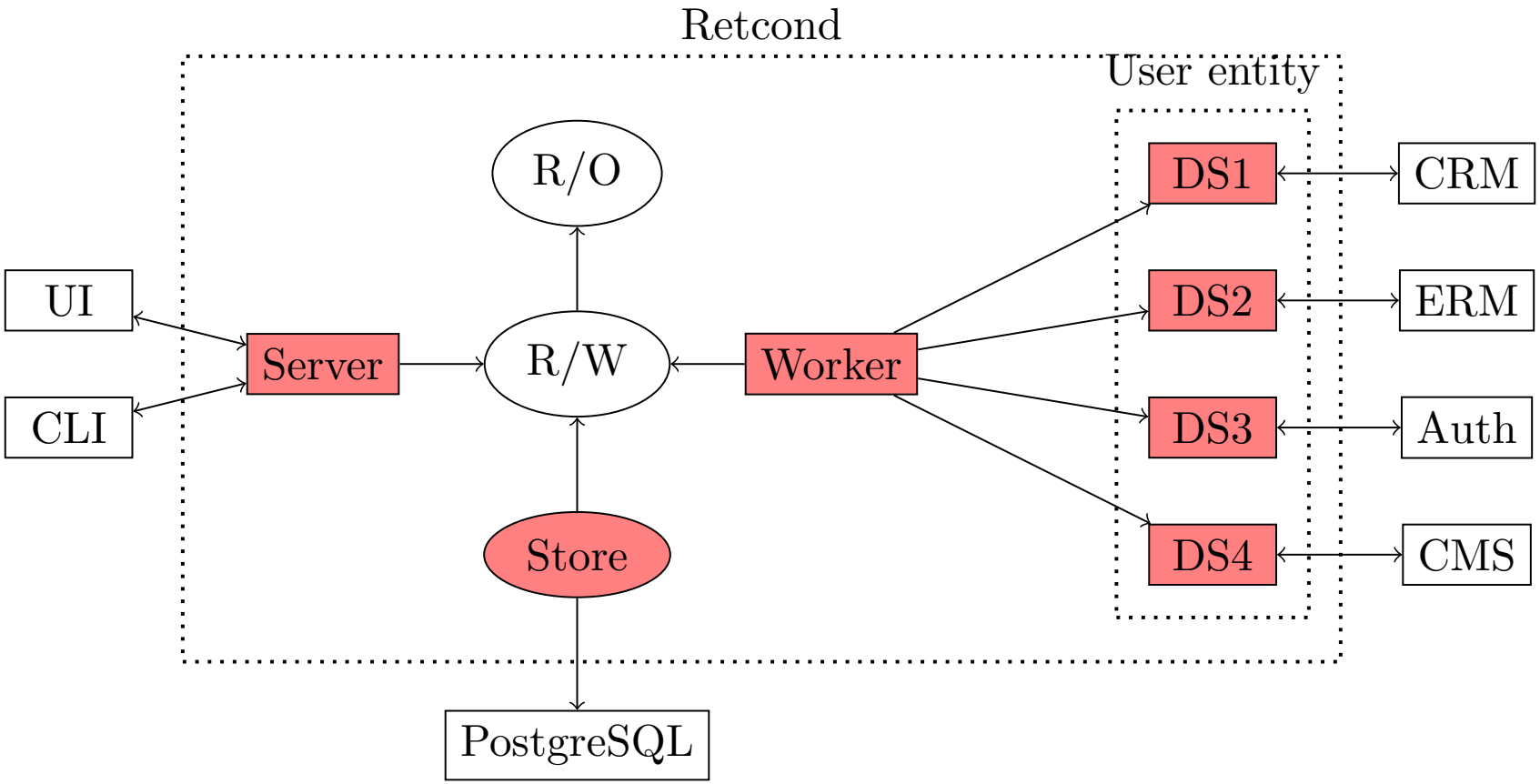


Figure 6: Structure of Retcon 1.x - Stateful components highlighted

What?

Shortcomings

By the time Retcon 1.x was done we had a big pile of:

- Scope creep.
- Inappropriate compromises.
- High incidental complexity.

While endlessly entertaining to work on (especially the "adding complexity" bit), this is clearly not a good place to be.

Data representation

The biggest, meanest, most pressing problem was the conversion of JSON data to a trie of `Text` strings before processing.

```
type DocKey =  Text
type DocValue = Text

newtype Document = Doc { unwrapDoc :: Trie DocKey DocValue }
```

This made the diff, merge, and patch operations very easy to implement but meant throwing away what little "type" information a JSON document contains.

Some data sources ended up `Showing` and `Reading` data coming into and out of the system.

Internal data sources

The next biggest mistake was compiling the data sources into the daemon:

- We found ourselves responsible for much more mutable state (database connections, etc.) which meant associated types and another set of existential wrappers.
- It allowed us to give data sources access to Retcon's operational data store, which led us to build several layers of abstraction to prevent them from *modifying* that data.
- It made it very difficult for anyone without a good handle on Haskell to use the system (MPTCs, type literals, phantom types, existential types).
- It made it difficult for someone *with* a good handle on Haskell to reuse their code (reuse a typeclass instance? At best you get to wrap a lot of helpers in an instance - boiler plate).

Type literals

We types to identify entities and data sources. Because these types were just “names” we picked type literals.

We should have used `DataKinds` (already enabled for type literals) and required client code to define a type to represent an entity and its data sources:

```
{-# LANGUAGE DataKinds #-}
data Customer = CustomerCRM | CustomerERP | CustomerCMS

instance RetconSource Customer CustomerCRM where
  -- ...
```

If nothing else this would have saved us the trouble of disabling the orphan instance warnings in every data source.

Version Two

Version Two

Using Retcon 1.x for a few entities quickly convinced us that it needed significant work.

Version Two

While the high-level structure of the system is unchanged, the implementation is much simpler:

1. We restructured Retcon 2.x as a **daemon**, not a library.
2. Each data source is an **external program** with a simple interface.
3. We standardised on an existing library for **configuration file handling** (rather than two different custom formats with fairly rubbish parsers).
4. We **removed** a number of **abstractions**, some **state**, and some of the over-the-top **type gymnastics** which added no value (but plenty of complexity).
5. We resolved the largest outstanding `@TODO(thsutton)` and now manipulate **JSON** directly without converting to other structures.

Retcon 1.x structure

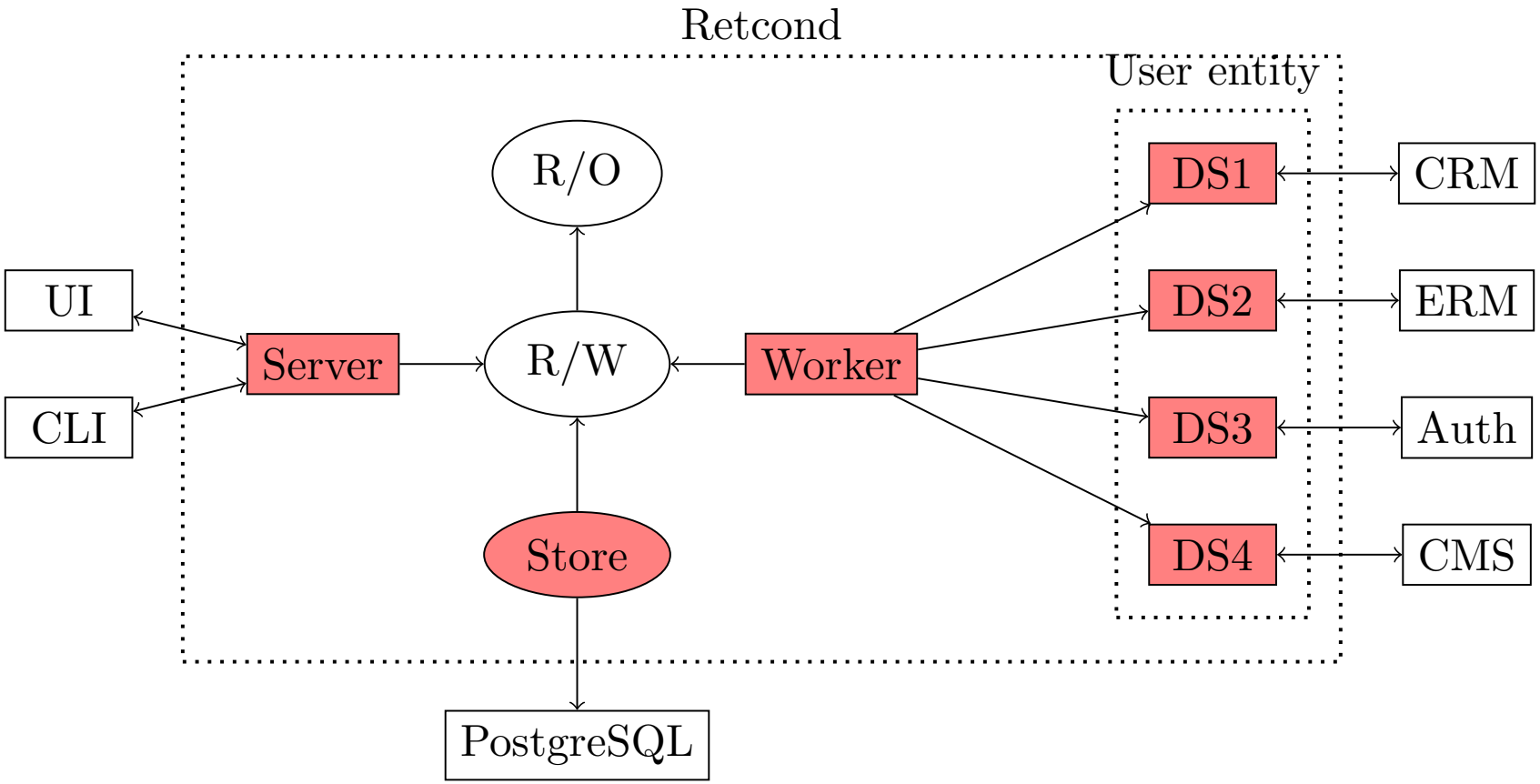


Figure 7: **Retcon 1.x** - Stateful components highlighted

Retcon 2.x structure

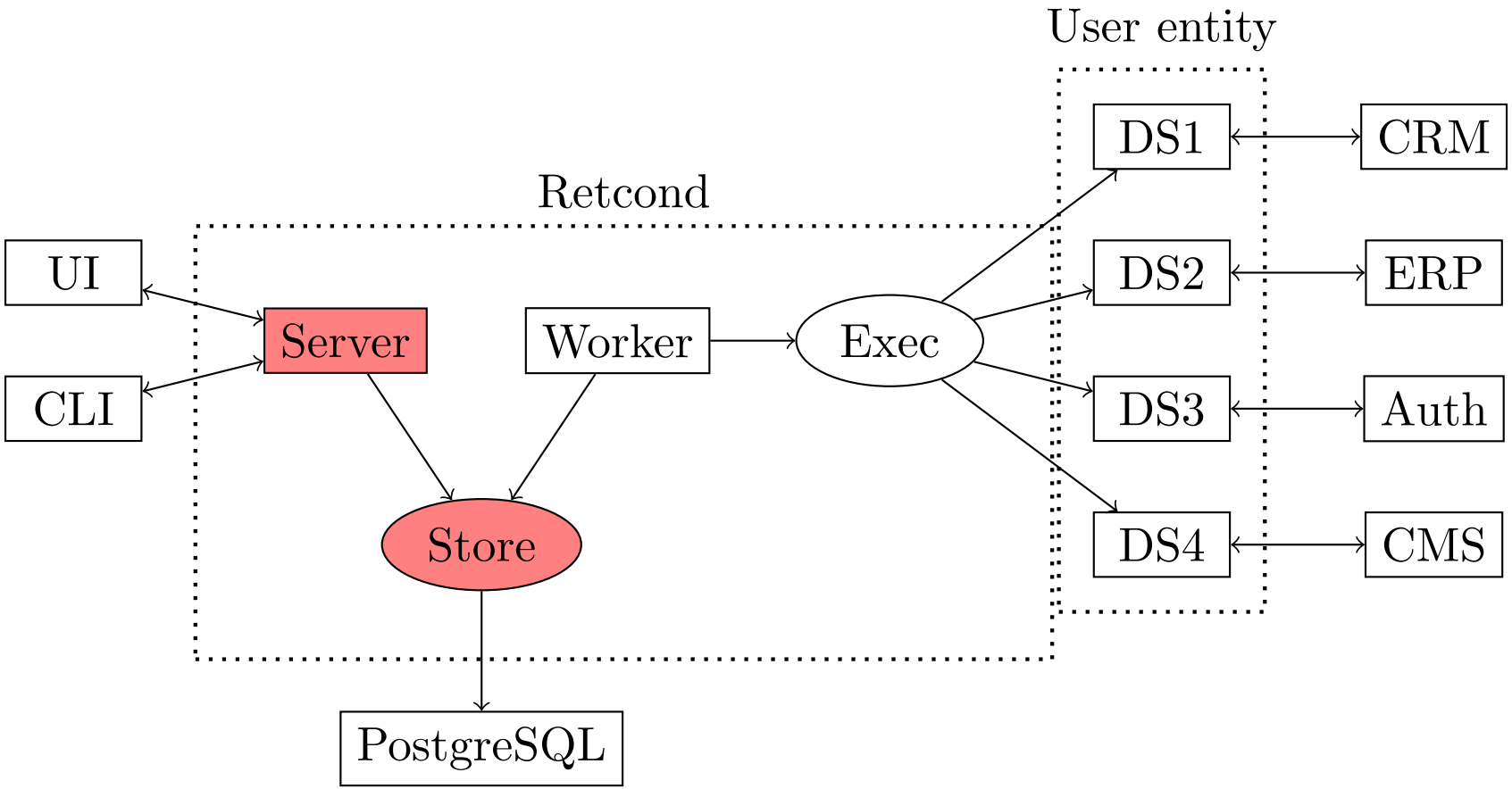


Figure 8: **Retcon 2.x** - Stateful components highlighted

Data sources

A Retcon 2.x data source is a program which follows a very simple interface:

Command	Arguments	Input	Output
`create`		Document	Foreign Key
`read`	Foreign Key		Document
`update`	Foreign Key	Document	Foreign Key
`delete`	Foreign Key		

If a data source wants or needs configuration files, persistent connections, etc. then it needs to implement those things for itself.

Daemon

Now that it's a standalone daemon, and not a library, deploying Retcon is now a relatively simple procedure:

1. Install it on the target system (we build and install OS packages).
2. Implement data source programs in your favourite programming language.
3. Write a simple configuration file describing:
   - the database, network, and logging settings; and
   - your entities and their data sources.
4. Configure your system to run the daemon.

Configuration

We used the `configurator` package so the format is pretty straightforward.

```
server {
  database  = "dbname=retcon user=retcon password=secr3t1"
  listen    = "tcp://127.0.0.1:9999"
  log-level = "DEBUG"
}
entities {
  customers {
    crm {
      command = "/usr/libexec/retcon/customers-crm"
    }
    erp {
      command = "/usr/libexec/retcon/customers-erp"
    }
  }
}
```

Type gymnastics

We replaced the complex, difficult to understand code using type literals, existentials, phantoms, scoped type variables, instance signatures, and other crazy type extensions with much simpler mechanisms:

- `IKey`, `FKey`, `Document` are now normal records with private fields.
- Honest to goodness `newtype` wrapped strings - not type literals - identify entities and data sources. They all come from outside the system:
  - The configuration file - which makes them correct by structure.
  - The network clients - which means someone *else* has a problem we need to report.
- Looking things up in a `Map` and pattern matching.

The resulting code is shorter, simpler, and easier to understand. Most of it now also benefits from compiler checks for incomplete pattern matches, etc.

Use JSON natively

Retcon 2.x operates on JSON documents directly. To support this we implemented the **aeson-diff** library (on Hackage) to extract and apply patches to aeson's JSON `Value` type.

We now maintain all distinctions between values (string, number, boolean, null) and collections (array, object) in JSON documents, and the newly implemented diff, patch, and merge algorithms are less likely to break a document.

Using JSON also leaves open some options for future functionality. I plan to extend Retcon with support for JSON Schema (just as soon as I see a schema used in the wild) to check the validity of documents entering and leaving the system.

What stayed the same?

Database Access

Retcon 2.x accesses its store of operational data through a typeclass which is largely unchanged from Retcon 1.x. What has changed is that the two further layers of abstraction (wrappers providing read/write or read/only interfaces) are gone.

We have implementations for PostgreSQL (for production) and an `IORef` containing a bunch of `Maps` (for testing purposes).

Messaging

Retcon is a client-server system. We used a very simple request-response protocol:

```
data Header request response where
  HeaderConflicted :: Header ReqConflicted RespConflicted
  HeaderChange     :: Header ReqChange     RespChange
  HeaderResolve    :: Header ReqResolve    RespResolve
  HeaderInvalid    :: Header ReqInvalid    RespInvalid
```

The networking is implemented using the ZeroMQ library. This, at least, is one thing that worked well in Retcon 1.x and we kept it more or less unchanged for Retcon 2.x.

Learnings

Learnings

- Keep it simple, stupid! If you can't justify a decision from the *user's requirements* it's probably wrong. Even in Haskell.
- Exotic type system and languages features are fun but probably won't reduce your defect rate. Even in Haskell.
- If you have a requirement, just meet it. "Short-term simplification" is often just code for "technical debt". Even in Haskell.

Conclusion

Conclusion

You've heard me rabbit on about:

1. A business problem we've had, which seems likely to be common;
2. The design of a fairly generic solution to the problem;
3. The initial implementation of the problem;
4. A number of problems in that implementation;
5. A second, better, implementation; and
6. The lessons I drew from the experience.

The team

The design and development of Retcon 1.x and, especially, 2.x saw contributions from:

- Oswyn Brent
- Andrew Cowie
- Barney Desmond
- Timo von Holtz
- Tran Ma
- Katie McLaughlin
- Christian Marie
- Geoffrey Roberts
- Thomas Sutton

But most of the really crazy stuff was down to me.

It's open source

Retcon is open source and available on GitHub and, Real Soon Now, Hackage.

- anchor/retcon
- thsutton/aeson-diff

Questions?