Atlassian

Functional Architecture:

an Experience Report

JED WESLEY-SMITH • @JEDWS

scala

because jvm

scala/fp

many teams now using Scala

no longer particularly controversial, mostly…

Scala is not very good for product development, for instance no backwards compatibility. but is great for distributed services

those not using/committed to FP failed, pretty hard

level of commitment varies, but pretty much everyone uses scalaz eventually

not a lot of experience, lots of education required

ambiata keep stealing people (Amazon too, bastards!)

problem:
too many containers

hosted products

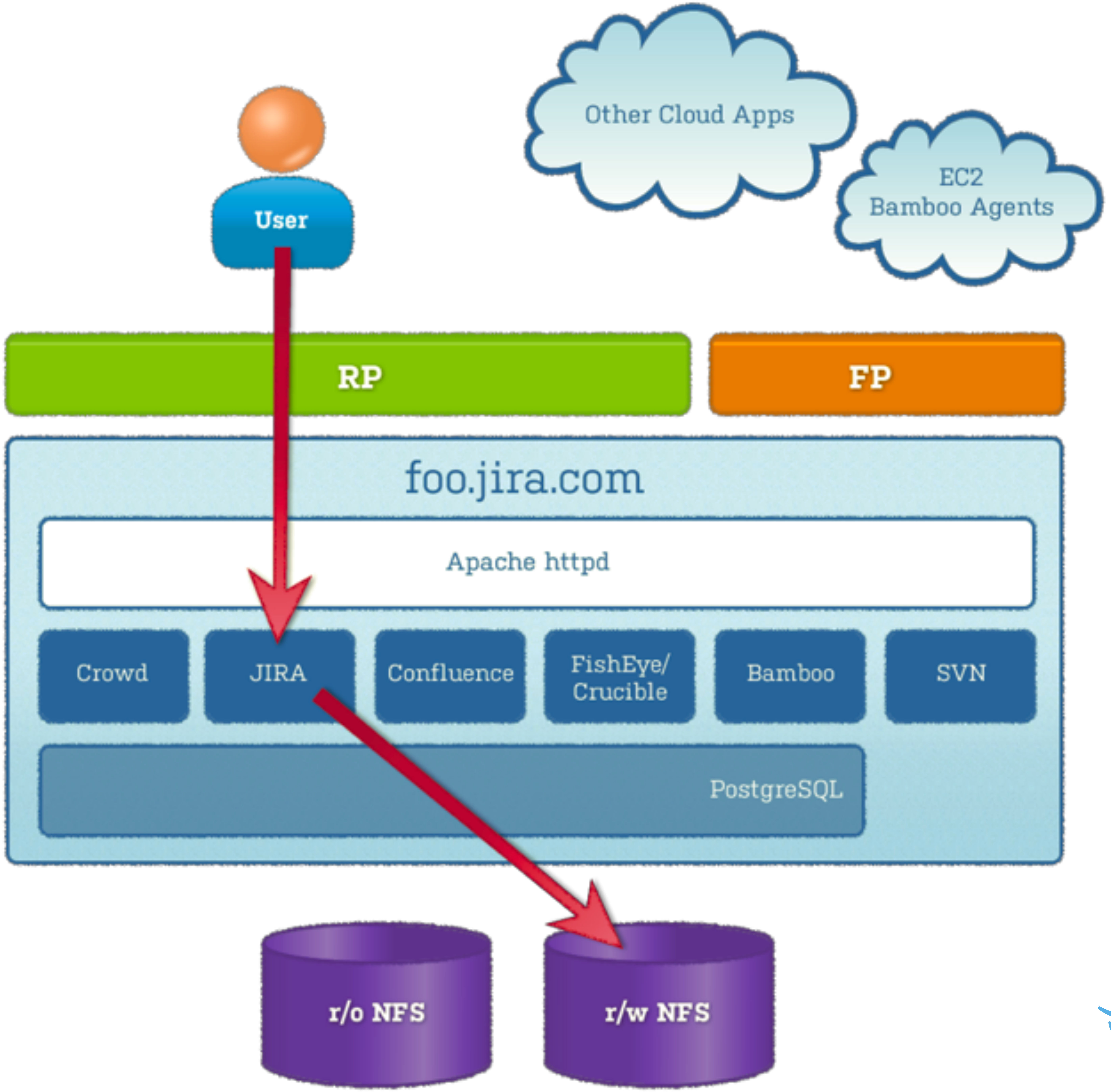
OpenVZ containers

JVM per application

DB per container

R/O + R/W FS

4GB of RAM each (to start)

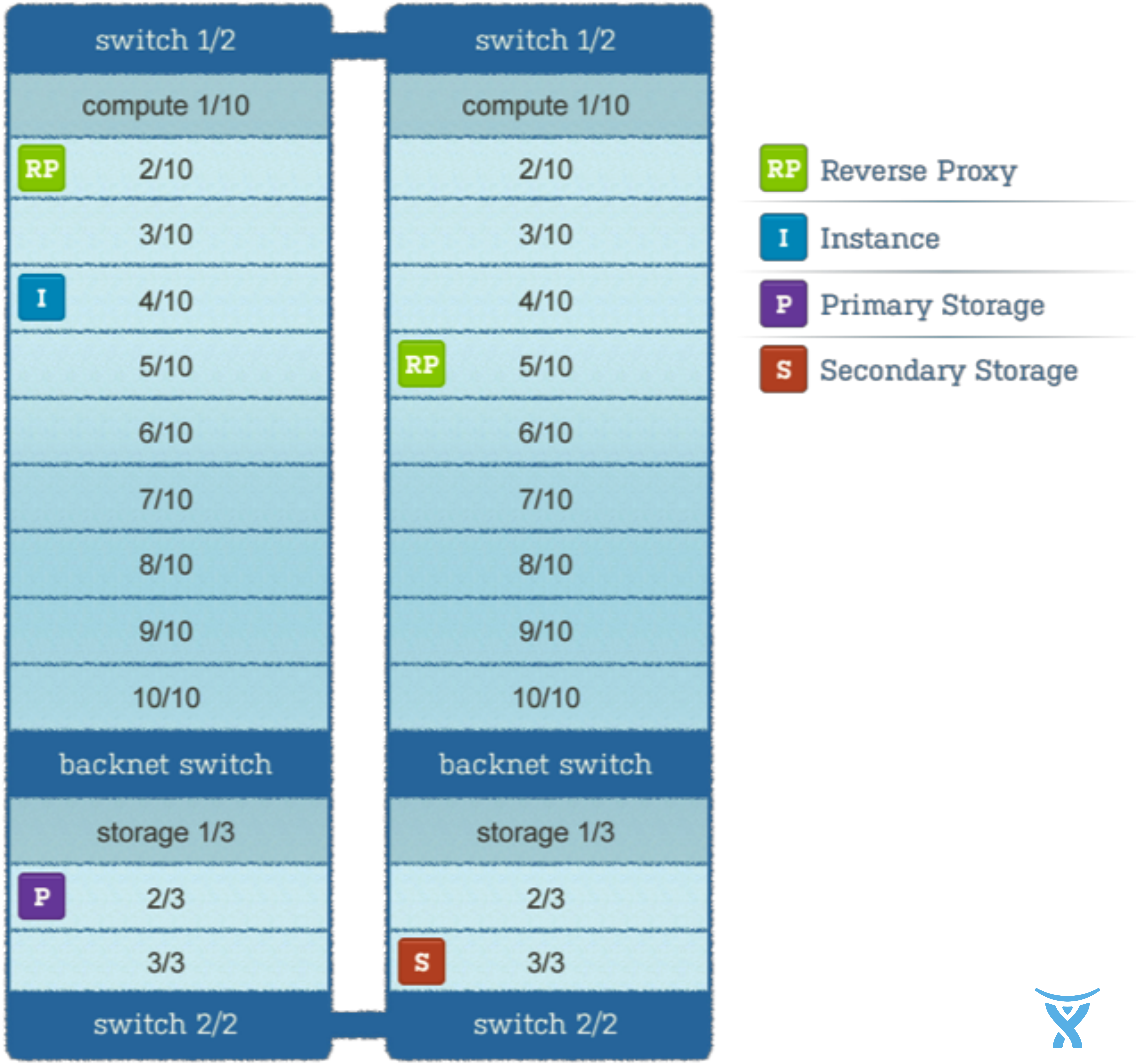


hosted products

>100K container instances

4 data centres

100s of racks



>100K

container instances

most are unused

solution:
services for horizontal decomposition

problem:
disaster recovery

disaster recovery

in case of a disaster, we need to be able to bring up a customer's instance

- within 1 hour
- with no more than 24 hours data loss

data on individual storage nodes physically coupled with a compute node holding the individual customer applications is sub-optimal

usage is asymmetric, some customers may take hours to reprovision

solution:
storage service

blobstore

content addressable storage in S3, client + content hash is address

maps name -> content hash, kept as an event log in DynamoDB

caches use hash as key, can cache forever (modulo secure data removal)

PUT is idempotent, data migration from containers is idempotent, run many times until final cut-over

encrypted storage – decrypted on download

mostly pure code-base, uses **scalaz-stream** for streaming and processing data

heavily tested, mostly ScalaCheck tests

blobstore

high-level operations represented as Reader transformers that take an API implementation as input

```
type StorageOp[F[_], A] = ReaderT[F, StorageAPI[F], A]

object Storage {
  def save[F[_]: Monad](c: Client)(data: UploadData.Stream, pw: Option[Passphrase] = None)
  : StorageOp[F, BlobMetadata] =
    StorageOp { _.save(c)(data, passphrase) }
}

trait StorageAPI[F[_]] {
  def save(client: Client)(data: UploadData.Stream, pw: Option[Passphrase])
          (implicit M: Monad[F]): F[BlobMetadata]

  def get(client: Client)(key: ContentKey, pw: Option[Passphrase], range: Option[ByteRange])
         (implicit M: Monad[F]): F[Option[DownloadResult.Data]]
}
```

blobstore

composition is clunky, much boilerplate and manual labour

```
type ComputationT[F[_], A] = ReaderT[F, Services[F], A]

case class Services[F[_]: Monad](keyStore: KeyStoreAPI[F], storage: StorageAPI[F])

trait ComputationSyntax {
  def lift[F[_], A](f: => F[A]): ComputationT[F, A] =
    ComputationT[F, A] { _ => f }

 implicit class KeyStoreToComputationT[A, F[_]: Monad](op: KeyStore[F, A]) {
    def toComputation: ComputationT[F, A] = op.contramap { _.keyStore }
  }

  implicit class StorageToComputationT[A, F[_]: Monad](op: Storage[F, A]) {
    def toComputation: ComputationT[F, A] = op.contramap { _.storage }
  }
}
```

blobstore

client code is ok, but lots of type plumbing

```
object Upload {
  import ComputationT._

  def apply[F[_]: ErrorM](client: Client, ctx: Context)(condition: BlobMappingValidator)
                        (pw: Option[Passphrase]): UploadData => ComputationT[F, Saved] = {
    case data @ UploadData.Stream(_, length, key) =>
      val k = (client, ctx, key)
      for {
        get <- KeyStore.get[F](k).toComputation
        _ <- condition(get).disjunction.leftMap(Failure.ConditionFailed(client, _)).toF.lift
        meta <- Storage.save(client)(data, passphrase).toComputation
        BlobMetadata(hash, length) = meta
        replaced <- KeyStore.save(k, condition, BlobMapping(hash, length)).toComputation
      } yield replaced
   }
```

blobstore

client code is ok, but lots of type plumbing

```
object Upload {
  import ComputationT._

  def apply[F[_]: ErrorM](client: Client, ctx: Context)(condition: BlobMappingValidator)
                       (pw: Option[Passphrase]): UploadData => ComputationT[F, Saved] = {
    case data @ UploadData.Stream(_, length, key) =>
      val k = (client, ctx, key)
      for {
        get <- KeyStore.get[F](k).toComputation
        _ <- condition(get).disjunction.leftMap(Failure.ConditionFailed(client, _)).toF.lift
        meta <- Storage.save(client)(data, passphrase).toComputation
        BlobMetadata(hash, length) = meta
        replaced <- KeyStore.save(k, condition, BlobMapping(hash, length)).toComputation
      } yield replaced
   }
```

blobstore

service gets redeployed many times per day

never had a regression

99.999% uptime since day 0 (better than Amazon network infrastructure)

100s of terabytes of data

centralised backup/restore, append-only so only backup latest writes

one weird occasional error, we blame Twitter/Finagle (but cannot prove it)

quite a few libraries, including nice FP interfaces for speaking to AWS, used by other teams, takes a bit of effort/curation to make generally useful libraries

problem:
sharing identity

throng

systems have their own notions of identity

they also have a notion of Organisation membership

- implied (eg. single Tenant/Org apps)
- modelled directly but poorly (HipChat only allows one membership)

details are poorly shared and synced between systems (which one is source of truth?)

existing apps have been known as Crowd, Horde, new system known as Throng

throng

similar underlying database philosophy as blobstore

everything is kept as an event log, but writes are much higher volume

uses eventually (perhaps causally) consistent secondary snapshot (cache) storage for fast read/query

much richer data model, but all stored using same EventSource abstraction

central operations are represented as simple values, rather than as functions

throng

operations are represented using ADTs

```
object User {
  sealed trait Op[A]
  case class ById(id: UserId) extends Op[Option[UserData]]
  case class AddUser(user: UserData) extends Op[UserData]

  class Users[F[_]](implicit I: Inject[Op, F]) {
    type UserAction[A] = Free.FreeC[F, A]

    def byId(id: UserId): UserAction[Option[UserData]] = lift(ById(id))

    def addUser(user: UserData): UserAction[UserData] = lift(AddUser(user))
  }

  object Users {
    implicit def users[F[_]](implicit ev: Inject[Op, F]): Users[F] = new Users
  }
}
```

throng

execution via NaturalTransforms

```
object EventSourcedUserDao {
  def apply(e: OrganisationUsersStream, es: ExecutorService)
           (usernameToUserId: e.API[Task, OrgUsername, UserId],
            userIdToUser: e.API[Task, UserId, UserData]) =
    new (User.Op ~> DaoResult) {
     def apply[A](a: User.Op[A]) =
        a match {
          case ById(id) => taskToDaoResult(userIdToUser.get(id))
          case AddUser(user) =>
            saveResultToDaoResult[List[UserIdentifierToId], UserData](_ => user, user) {
              for {
                t <- identifierToId.save(user.id.org, OrganisationUsers.InsertUser(user))
                _ <- Task.fork(userIdToUser.refreshSnapshot(user.id, t.id.map { _.sequence }))(es)
              } yield t
            }
        }
```

throng

composition is more general

```scala
sealed abstract class Inject[F[_], G[_]] {
  def inj[A](fa: F[A]): G[A]
  def prj[A](ga: G[A]): Option[F[A]]
}

case class Coproduct[F[_], G[_], A](run: F[A] \/ G[A])

package object api {
  type F0[A] = Coproduct[OrganisationAddress.Op, dao.Application.Op, A]
  type F1[A] = Coproduct[User.Op, F0, A]
  type Throng[A] = Coproduct[Directory.Op, F1, A]

  type Action[F, A] = Free.FreeC[Throng, A]

  type ThrongOp[A] = Action[Throng, A]
}
```

throng

composition is more general

```
object API {

 def userById(id: UserId)(implicit U: Users[Throng]): ThrongOp[Option[UserData]] =
    U.byId(id)

  def addUser(org: OrganisationId, user: NewUser)(implicit U: Users[Throng]): ThrongOp[UserData] =
    U.addUser(UserData(org, user))

  def addApp(app: ApplicationData)(implicit A: Applications[Throng]): ThrongOp[ApplicationData] =
    A.addApplication(app)
}
```

work in progress...

but, this pattern is already being used more widely

thanks!