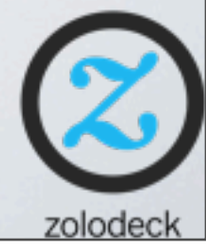
clojure
+ datomic
+ storm

= **zolodeck**



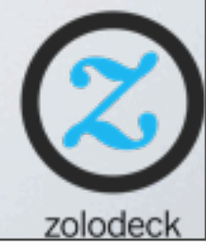
Amit Rathore

Startup Guy

amit@zolodeck.com

zolodeck

software applications today



<insert buzzword>

big data

NoSQL

machine-learning

real-time

performant

availability

cloud

no ops

web-scale

choices

architecture

stack

language

database

productivity

startup world

delivered yesterday

with 2 engineers

(usually)

must provide all the cool features

must provide all the cool features

(that you think you need)

cheaper to operate than what customers pay

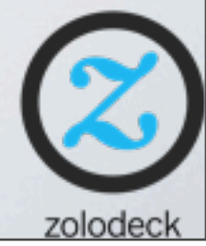
“pivot” on a dime

Right?

design

design to scale

performance vs scalability



typical path

simple beginnings

a web-server,
a database-server

Low complexity, no overhead
Quick development, agility, lots of features
No redundancy, low operational cost

non-trivial number of users

more web-servers

Add some for redundancy, for performance
Optimize the DB,
add redundancy at the DB level
Still fairly simple from the app standpoint

more web-servers
scale up the database

Add some for redundancy, for performance
Optimize the DB,
add redundancy at the DB level
Still fairly simple from the app standpoint

more growth

caching content

reverse proxy, fancy load balancers
may start needing some re-coding of the app

caching content
more web-servers

reverse proxy, fancy load balancers
may start needing some re-coding of the app

caching content
more web-servers
single database-server not enough

reverse proxy, fancy load balancers
may start needing some re-coding of the app

caching content
more web-servers
single database-server not enough

reverse proxy, fancy load balancers
may start needing some re-coding of the app
incidental complexity

caching content
more web-servers
single database-server not enough

may need some re-coding

reverse proxy, fancy load balancers
may start needing some re-coding of the app

database replication

- multiple DB servers can handle high read volumes (load is spread across)
- writes: not so much - all the replicas must write the same data

database replication

reads

- multiple DB servers can handle high read volumes (load is spread across)
- writes: not so much - all the replicas must write the same data

database replication

reads
writes?

- multiple DB servers can handle high read volumes (load is spread across)
- writes: not so much - all the replicas must write the same data

even more growth

memcached

- replication starts taking too long
- certain features get their own DB
- significant re-architecture

memcached
CDN

- replication starts taking too long
- certain features get their own DB
- significant re-architecture

memcached
CDN
database partitioning

- replication starts taking too long
- certain features get their own DB
- significant re-architecture

memcached,
CDN,
database partitioning

significant re-architecture

- replication starts taking too long
- certain features get their own DB
- significant re-architecture

even more growth

- hasn’t someone done this before?
- why wasn’t this designed to scale?
- partition on features - geo, last-names, user-id,
- create user-clusters, all available features on every cluster,
- hashing or master DB to locate cluster

even more growth

Aaaaaa!!

- hasn’t someone done this before?
- why wasn’t this designed to scale?
- partition on features - geo, last-names, user-id,
- create user-clusters, all available features on every cluster,
- hashing or master DB to locate cluster

even more growth

Aaaaaa!!

Aaaaaaaaaa!!

- hasn't someone done this before?
- why wasn't this designed to scale?
- partition on features - geo, last-names, user-id,
- create user-clusters, all available features on every cluster,
- hashing or master DB to locate cluster

even more growth

Aaaaaa!!

Aaaaaaaaaa!!

rethink everything

- hasn't someone done this before?
- why wasn't this designed to scale?
- partition on features - geo, last-names, user-id,
- create user-clusters, all available features on every cluster,
- hashing or master DB to locate cluster

new features?

- everyone only working on infrastructure

finally...

stability through scalability

- OK performance
- new features can begin
- somewhat manageable

ready for more growth

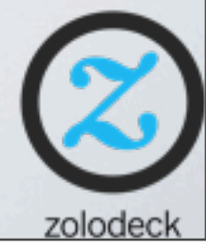
zolodeck

ready for more growth?

zolodeck

here's the point

scalability vs agility

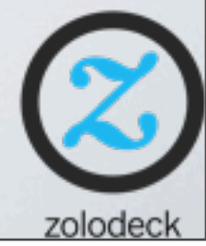


a real tradeoff

typical path for a reason

typical because it is logical

incidental complexity



a proposed solution

data
processing

hadoop anyone?

simple

simple

and easy

zolodeck

improve life through artificial intelligence

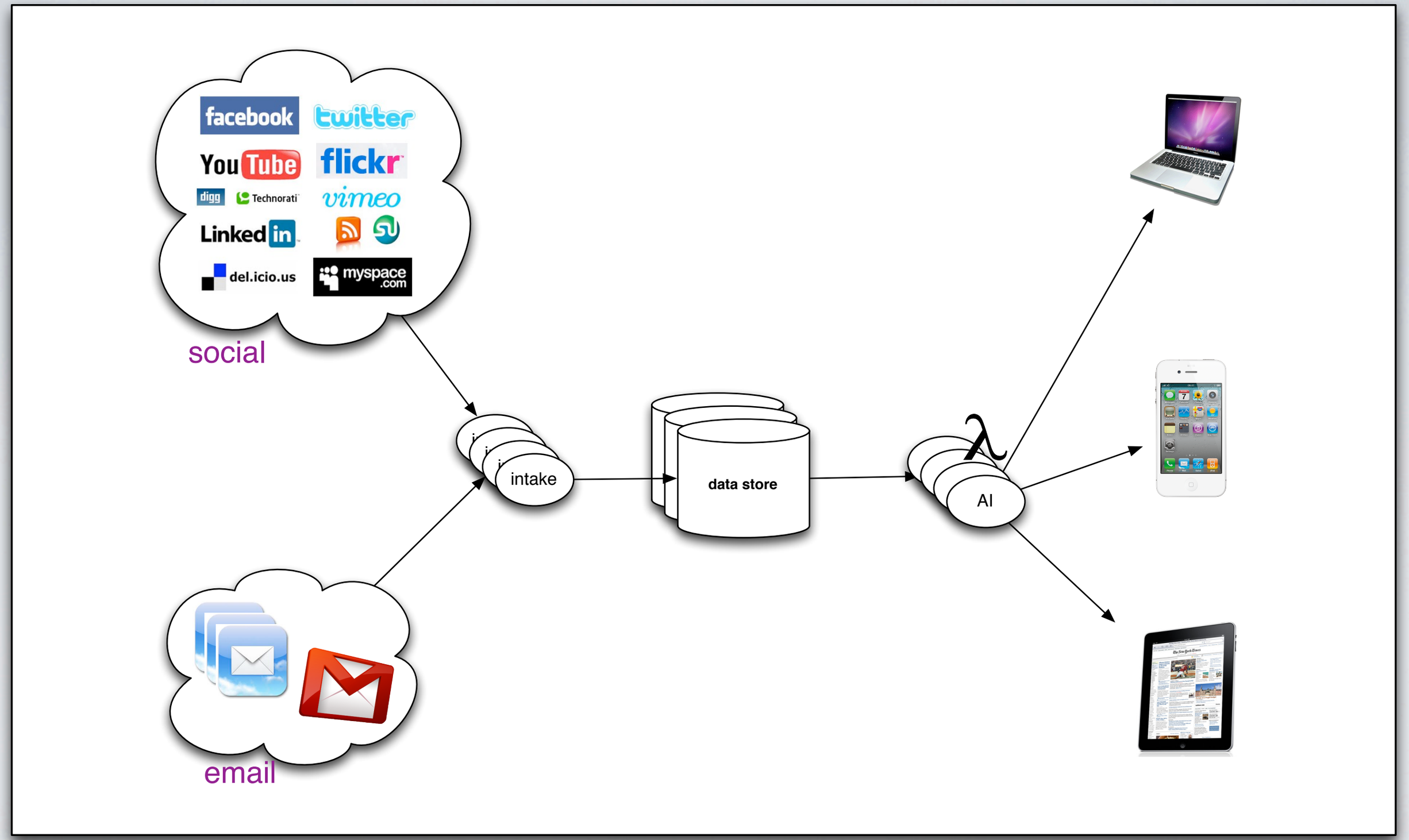
networking is good

(hello Strange Loop)

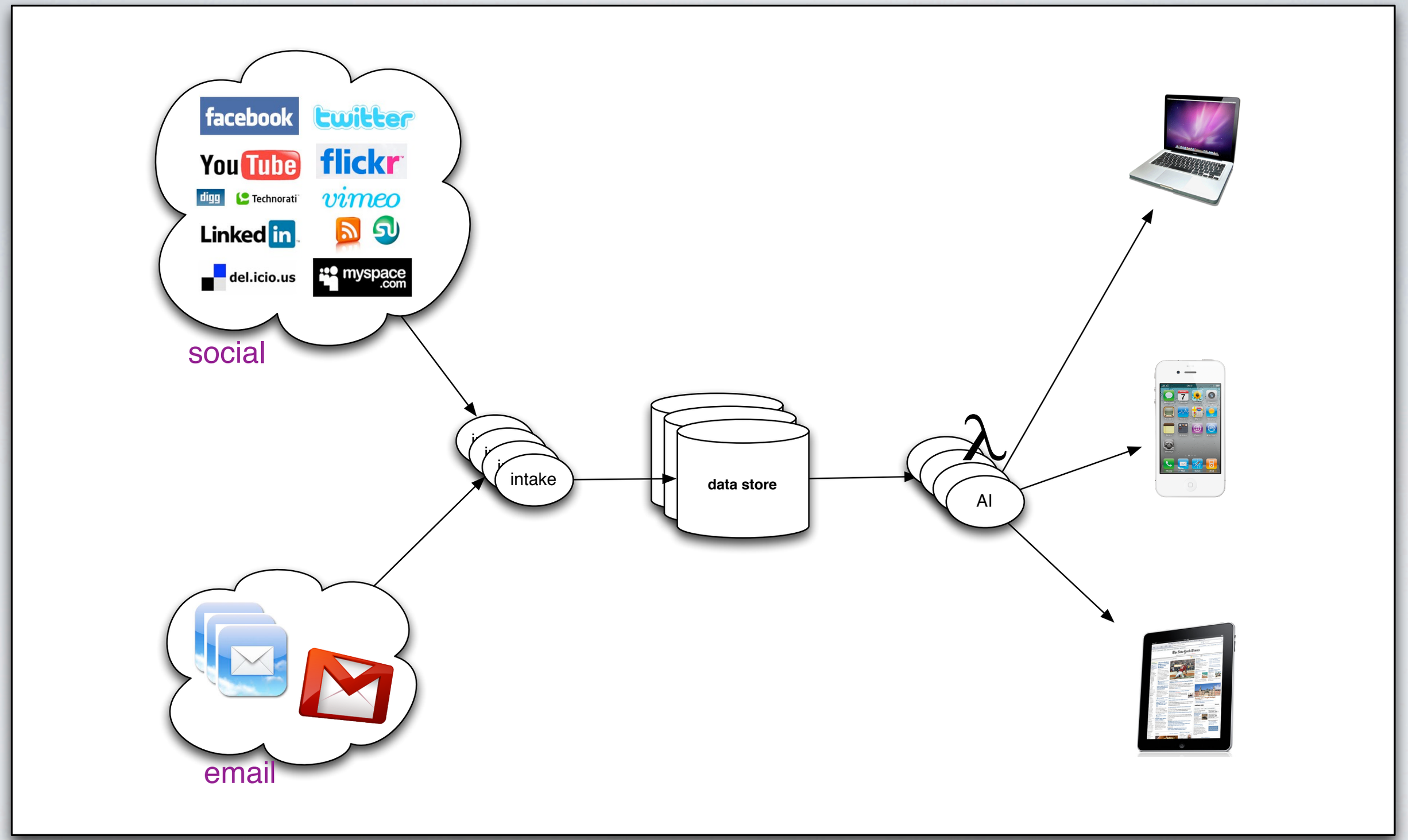
zolodeck - intelligent virtual assistant

- monitor all social networks, email inboxes
- show where you are with your existing network (lots of metrics)
- watch out for you when a relationship is weakening
- remind you to reach out with the right person, at the right time, with the right content
- help you connect with people you should be connected to, and help you connect people who should be connected

zolodeck

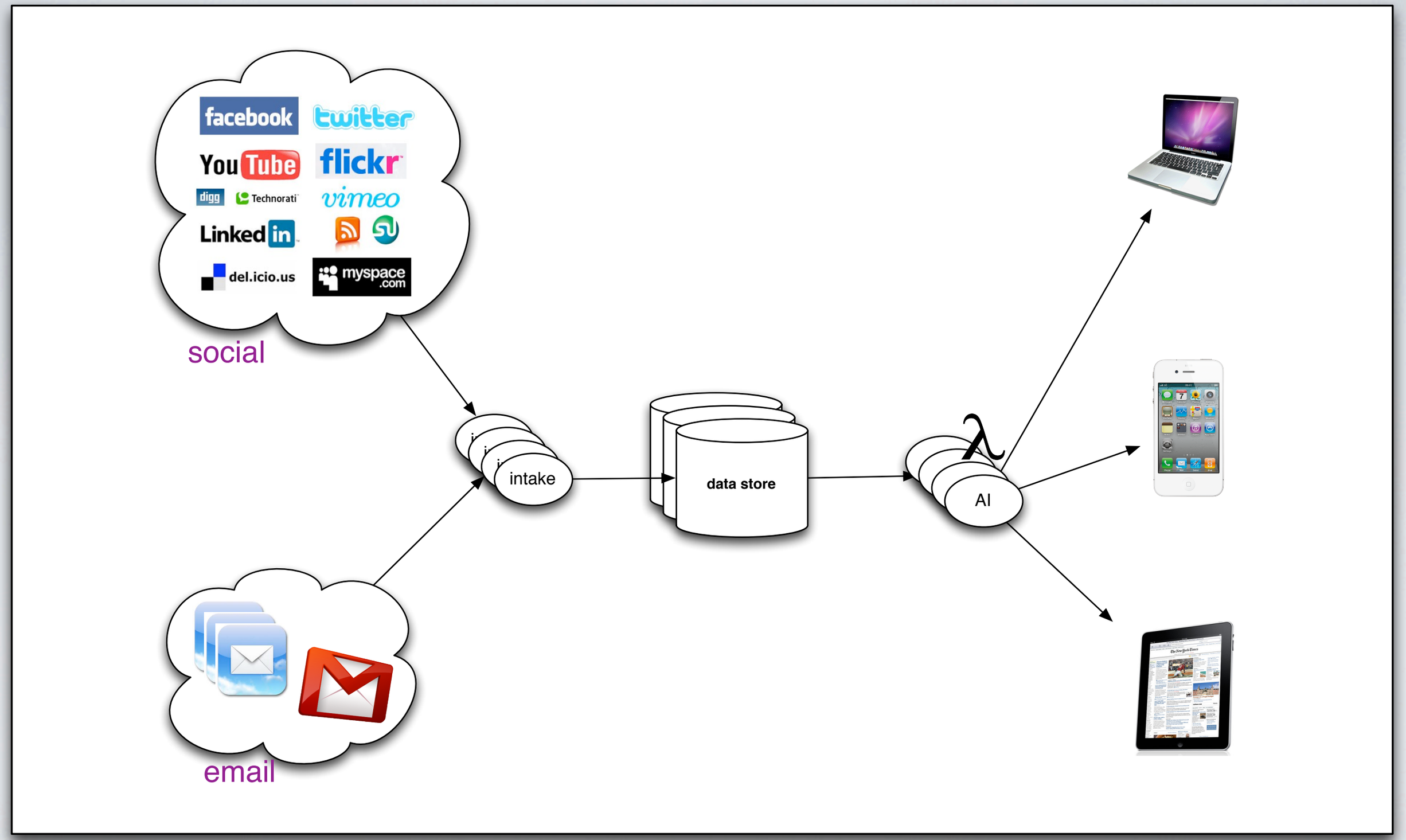

facebook
twitter
YouTube
flickr
digg
Technorati
vimeo
LinkedIn
del.icio.us
myspace.com
social
intake
data store
λ
AI
email

zolodeck



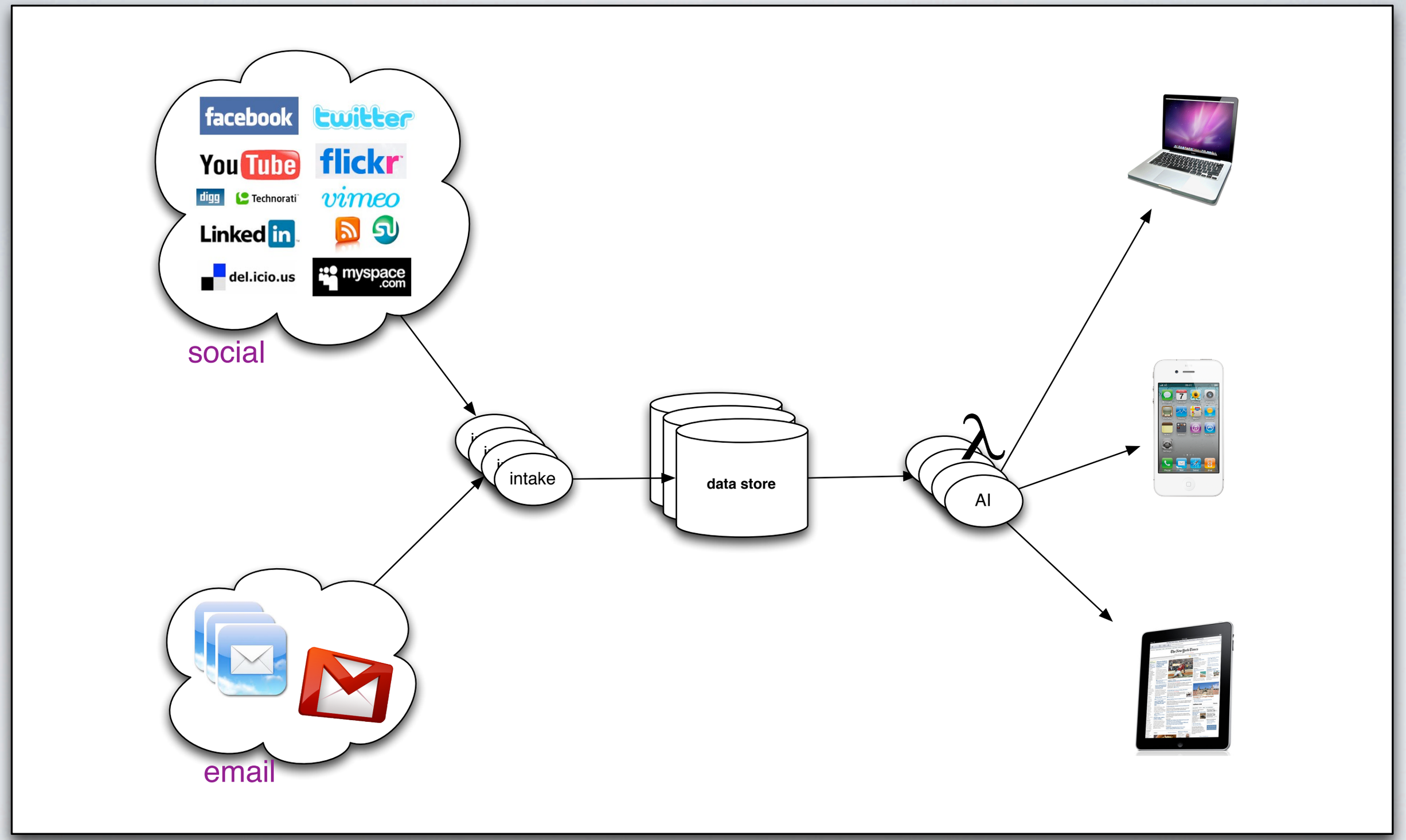
data

zolodeck



data volume

zolodeck

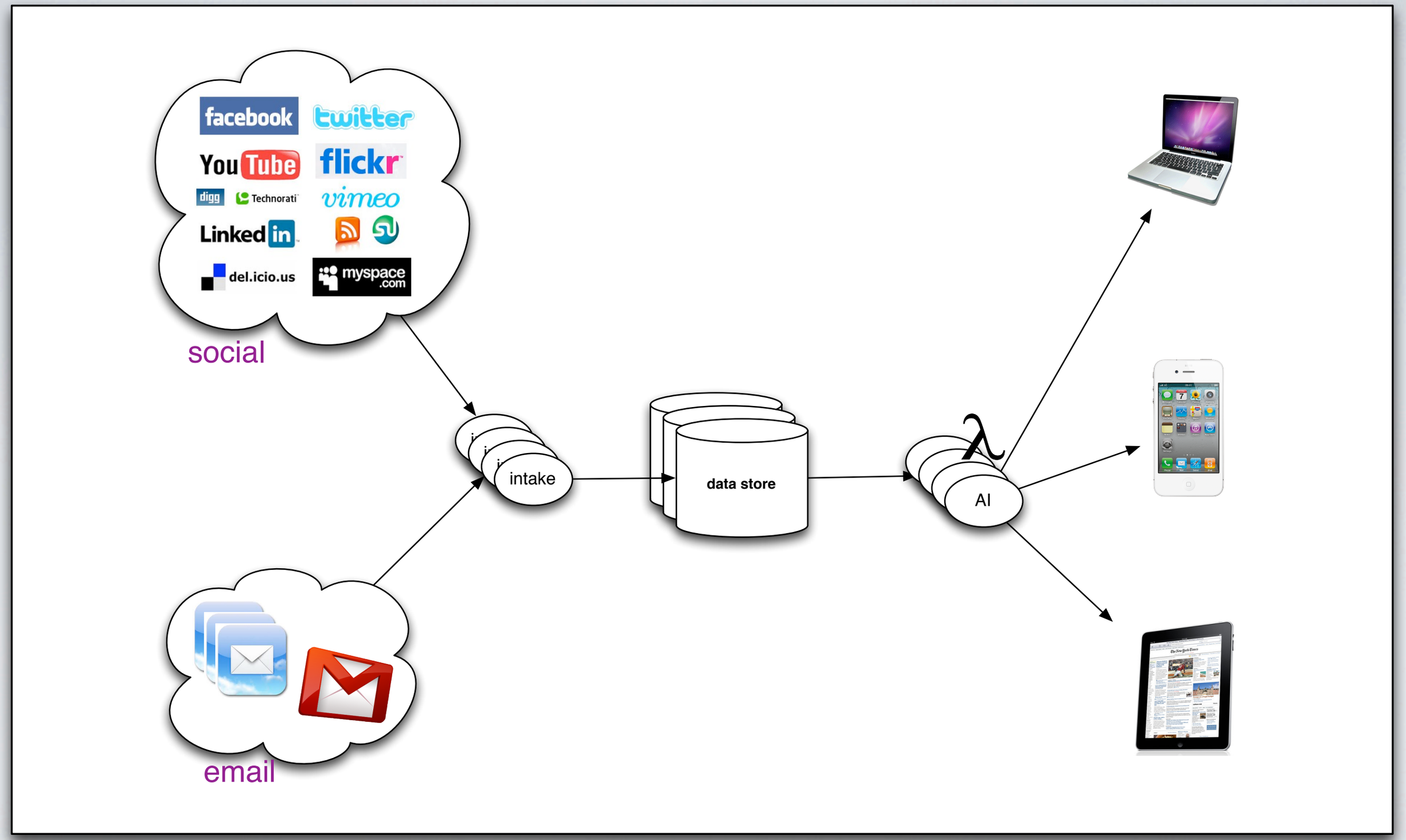


data volume

processing

zolodeck

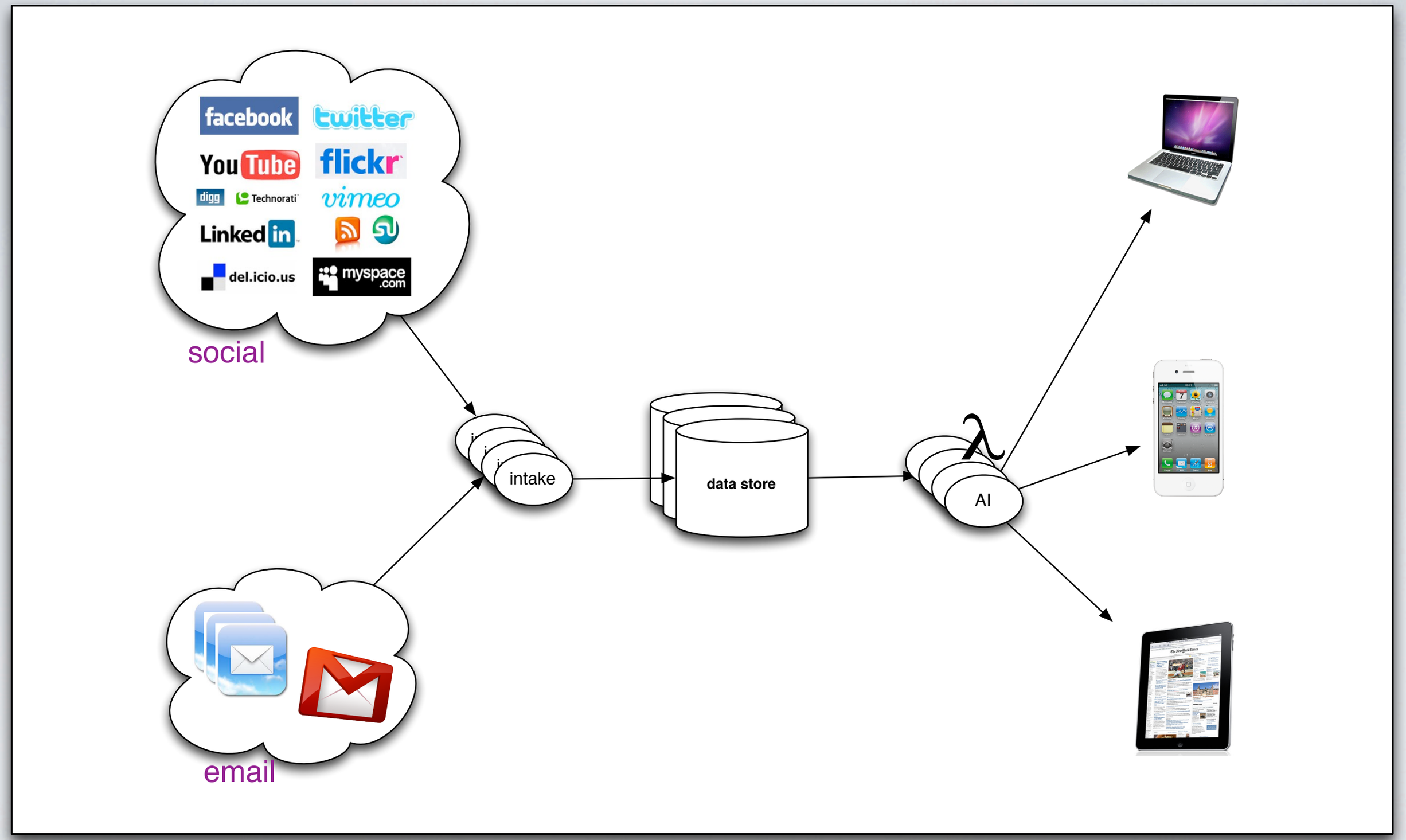
zolodeck



data volume

processing: scale

zolodeck



data volume

processing: scale, real-time

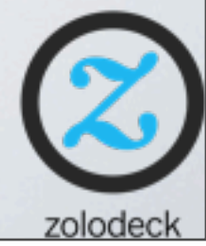
our choices

language: clojure

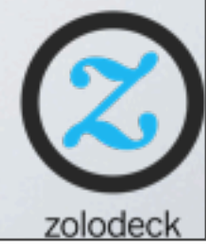
language: clojure
processing: storm

language: clojure
processing: storm
data: datomic

datomic



the inevitable path to sharding



the inevitable pain of sharding

lose the things a database does

lose the things a database does

consistency
transactions
queries

the new NoSQL stuff

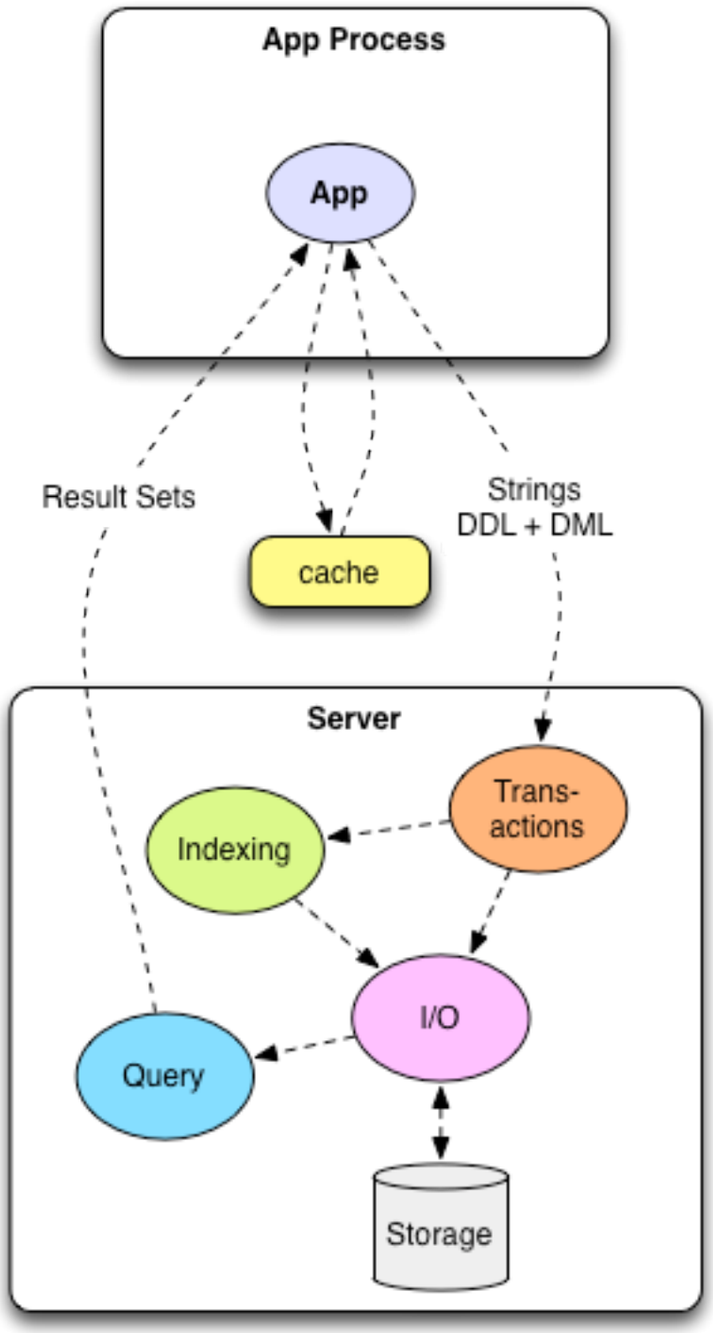
gain vs loss

consistency
transactions
queries

datomic gives you these back

datomic gives you these back

without manual sharding



the good old database

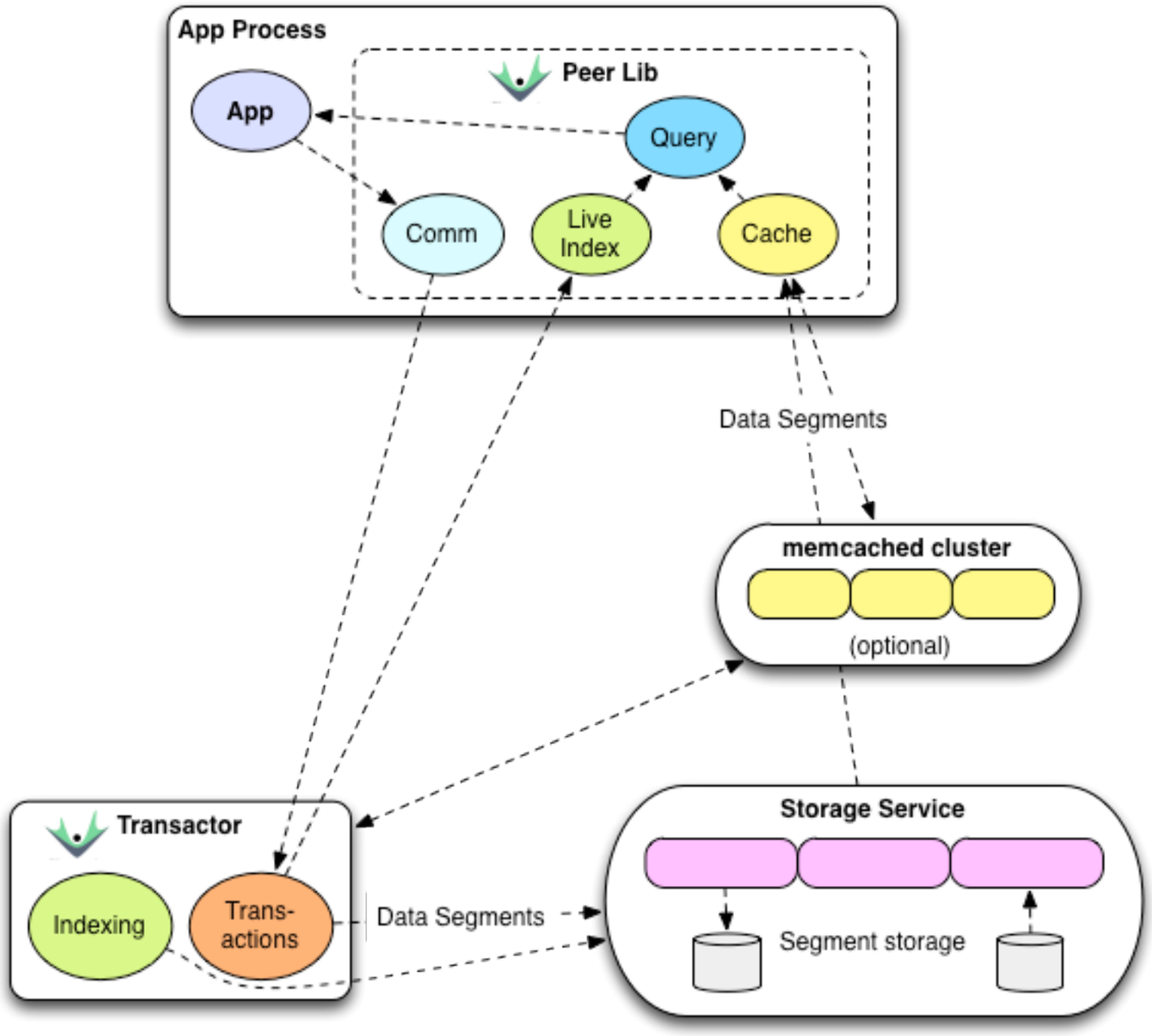
zolodeck

the database deconstructed

the database deconstructed

relocate subsystems
separate services each do one thing
simplification

peers
transactors
storage services



the datomic way

zolodeck

separating reads and writes

applications only read from storage
transactor - handles transactions, provides consistency, reflects changes to peers

integrated data distribution

integrated data distribution

built-in, in-memory caches

integrated data distribution

built-in, in-memory caches
self-tunes to working set

integrated data distribution

built-in, in-memory caches
self-tunes to working set
automatic

zolodeck

peers have the query engine

datalog

simple rules and data patterns
declarative: implicit joins, meaning is evident [:find ?e ?x :where [?e :age 42] [?e :likes ?x]]
locality: data + app

elastic

cloud-ready

elastic

add more peers

cloud-ready

elastic

add more peers
commodity hardware

cloud-ready

elastic

add more peers
commodity hardware
resilient to failures

cloud-ready

immutable

immutable

remember everything

immutable

remember everything
audit everything

immutable

remember everything
audit everything
automatic

Minimal schema

Minimal schema

fact oriented - entity, attribute, value, transaction

Minimal schema

fact oriented - entity, attribute, value, transaction

anya

Minimal schema

fact oriented - entity, attribute, value, transaction

anya age

Minimal schema

fact oriented - entity, attribute, value, transaction

anya age 3

Minimal schema

fact oriented - entity, attribute, value, transaction

anya age 3 t100

Minimal schema

fact oriented - entity, attribute, value, transaction

anya age 3 t100

supports cardinality

Minimal schema

fact oriented - entity, attribute, value, transaction

anya age 3 t1

supports cardinality

structure is up to the app code

transactional

- consistent: without impeding other threads or peers
- consistent: reads/queries without transactions (thanks to immutability)
- peers get a queue of all transactions
- can use transactions in queries
- facilitates event-driven triggers without polling

transactional

ordered

- consistent: without impeding other threads or peers
- consistent: reads/queries without transactions (thanks to immutability)
- peers get a queue of all transactions
- can use transactions in queries
- facilitates event-driven triggers without polling

transactional

ordered
first-class transactions

- consistent: without impeding other threads or peers
- consistent: reads/queries without transactions (thanks to immutability)
- peers get a queue of all transactions
- can use transactions in queries
- facilitates event-driven triggers without polling

transactional

ordered
first-class transactions
annotated transactions

- consistent: without impeding other threads or peers
- consistent: reads/queries without transactions (thanks to immutability)
- peers get a queue of all transactions
- can use transactions in queries
- facilitates event-driven triggers without polling

transactional

ordered
first-class transactions
annotated transactions
as-of, since, or windowed queries

- consistent: without impeding other threads or peers
- consistent: reads/queries without transactions (thanks to immutability)
- peers get a queue of all transactions
- can use transactions in queries
- facilitates event-driven triggers without polling

full-text search

full-text search

built-in lucene

storage services

in-memory, in-process - great for testing

storage services

Amazon Dynamo
RDBMS
memory

in-memory, in-process - great for testing

storage services

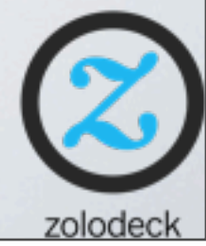
Amazon Dynamo
RDBMS
memory

unit testing

in-memory, in-process - great for testing

so that's the data story

now, the processing story



how to process lots of data?

when machine is not enough?

start with a single box

typical progression:

add machines
assign work (mod hash?)
add more machines

next:

message queue
workers
dispatcher

managing work

managing work

assigning machines

managing work

assigning machines
monitoring workers
restart workers

managing work

assigning machines
monitoring workers
restart workers
re-assign work

managing work

assigning machines
monitoring workers
restart workers
re-assign work
idempotent?

pipelined work

- multiple steps, workers send messages, multiple queues...
- for scale, or failover, or contracting elastic cluster

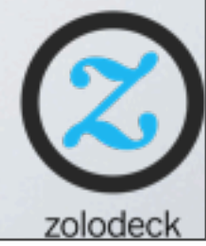
pipelined work

Aaaaaa!!

- multiple steps, workers send messages, multiple queues...
- for scale, or failover, or contracting elastic cluster

painful

painful
not fault-tolerant



painful
not fault-tolerant
tedious

abstract away:

higher-level abstraction than message passing

abstract away:

guaranteed processing
horizontal scalability
fault-tolerance

higher-level abstraction than message passing

abstract away:

guaranteed processing
horizontal scalability
fault-tolerance

focus on business logic

higher-level abstraction than message passing

abstract away:

guaranteed processing
horizontal scalability
fault-tolerance

focus on business logic
not like Hadoop

higher-level abstraction than message passing

abstract away:

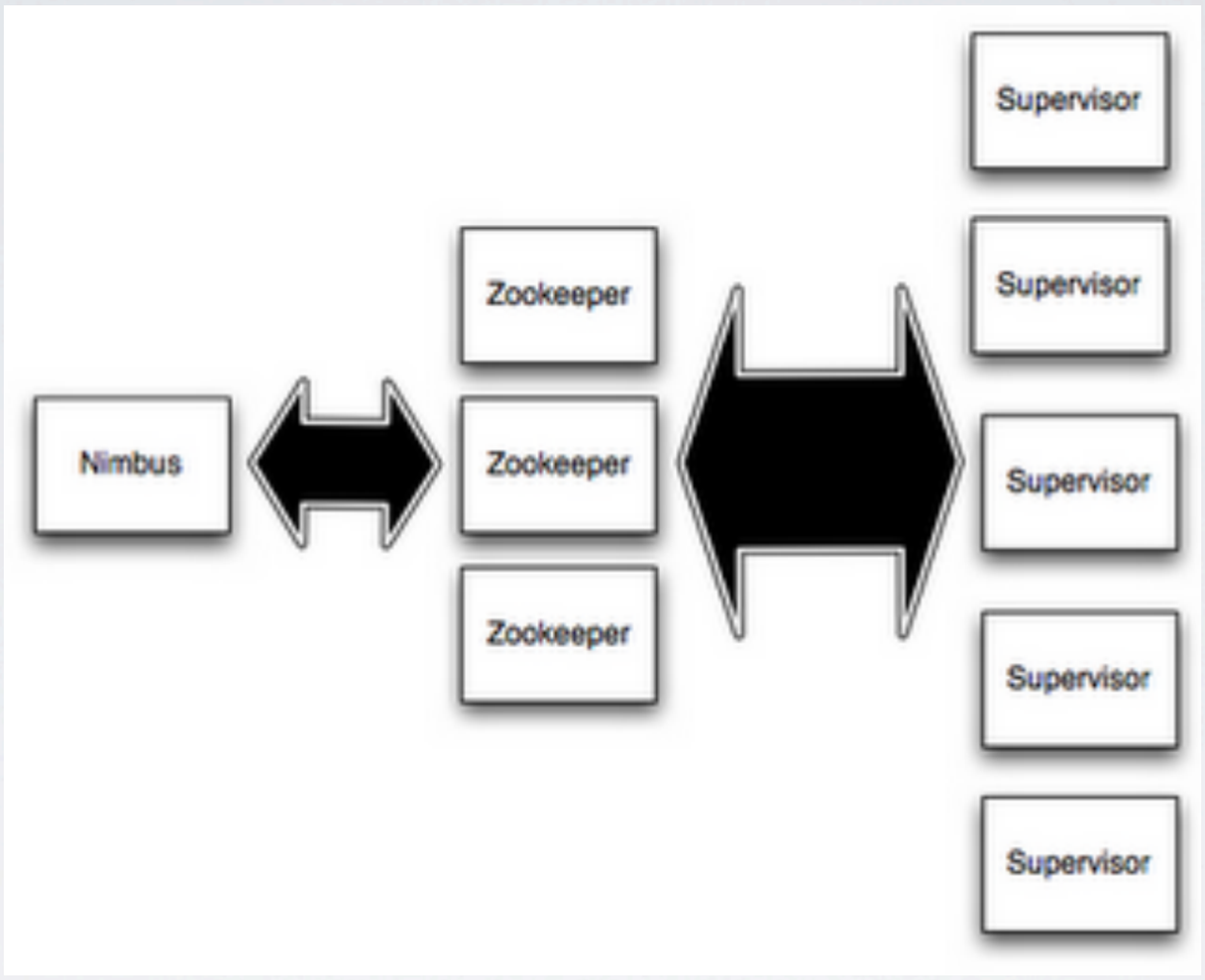
guaranteed processing
horizontal scalability
fault-tolerance

focus on business logic
not like Hadoop

storm

higher-level abstraction than message passing

components:



nimbus
zookeeper
supervisor

topology

```
storm jar something.jar classname arg1 arg2
```

- stream: unbounded sequence of tuples (named list of values)
- tupple: any type of value, with custom serializer
- spout: sources of streams (eg. reads from twitter API)
- bolt: process input streams, and produce output streams
- bolt: does all the work
- topology definitions are thrift structs (so any programming language can be used)

topology

`storm jar something.jar classname arg1 arg2`

stream

- stream: unbounded sequence of tuples (named list of values)
- tupple: any type of value, with custom serializer
- spout: sources of streams (eg. reads from twitter API)
- bolt: process input streams, and produce output streams
- bolt: does all the work
- topology definitions are thrift structs (so any programming language can be used)

topology

```
storm jar something.jar classname arg1 arg2
```

stream

spout

- stream: unbounded sequence of tuples (named list of values)
- tupple: any type of value, with custom serializer
- spout: sources of streams (eg. reads from twitter API)
- bolt: process input streams, and produce output streams
- bolt: does all the work
- topology definitions are thrift structs (so any programming language can be used)

topology

```
storm jar something.jar classname arg1 arg2
```

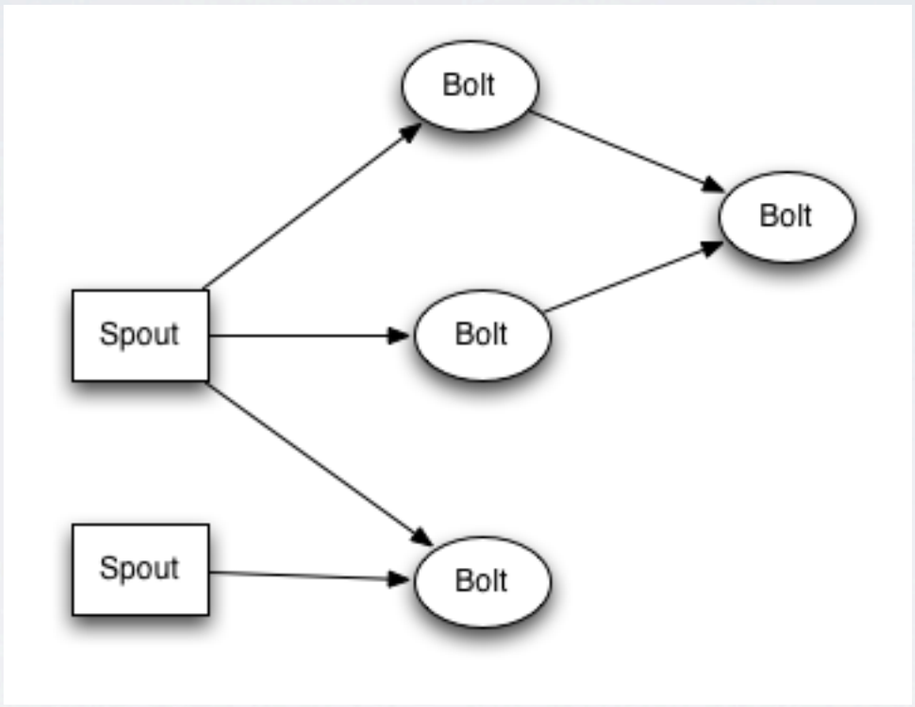
stream

spout

bolt

- stream: unbounded sequence of tuples (named list of values)
- tupple: any type of value, with custom serializer
- spout: sources of streams (eg. reads from twitter API)
- bolt: process input streams, and produce output streams
- bolt: does all the work
- topology definitions are thrift structs (so any programming language can be used)

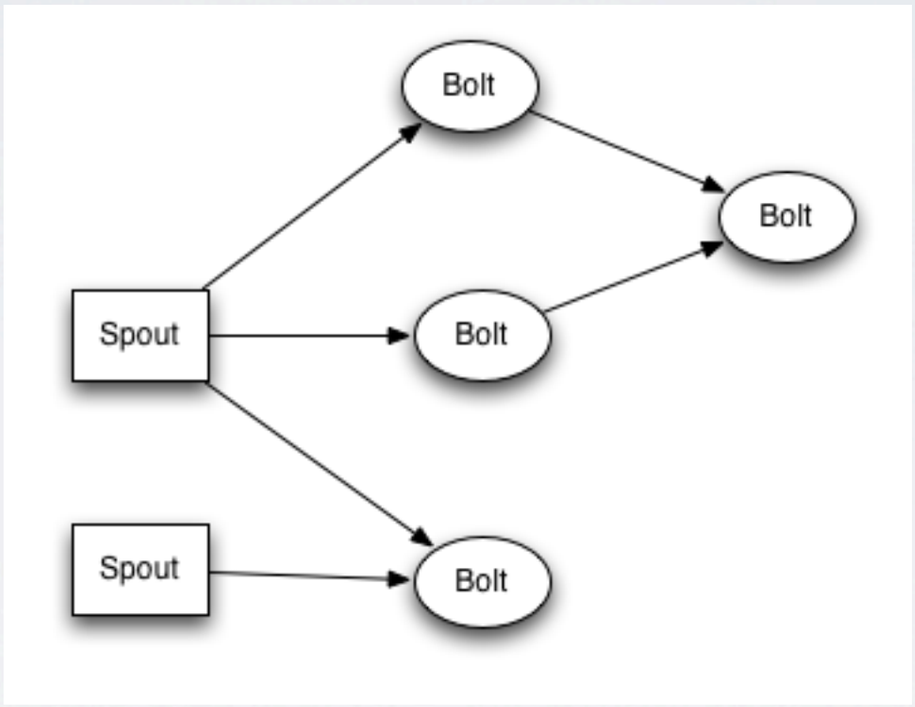
a topology



zolodeck

- multiple nodes
- multiple workers on each node
- multiple tasks will run on each worker, in parallel
- exchanging data as needed
- stream grouping: how to partition stream of messages (e.g. shuffle, fields (mod hash), all, global)

a topology

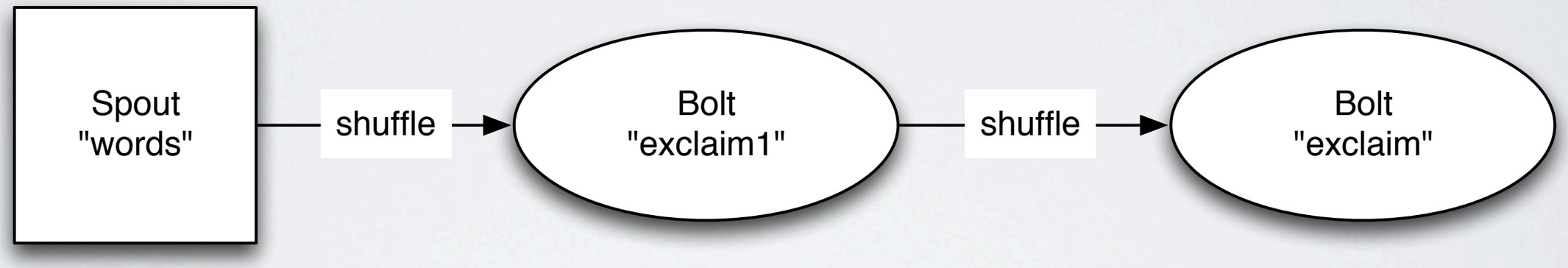


grouping

zolodeck

- multiple nodes
- multiple workers on each node
- multiple tasks will run on each worker, in parallel
- exchanging data as needed
- stream grouping: how to partition stream of messages (e.g. shuffle, fields (mod hash), all, global)

topology builder



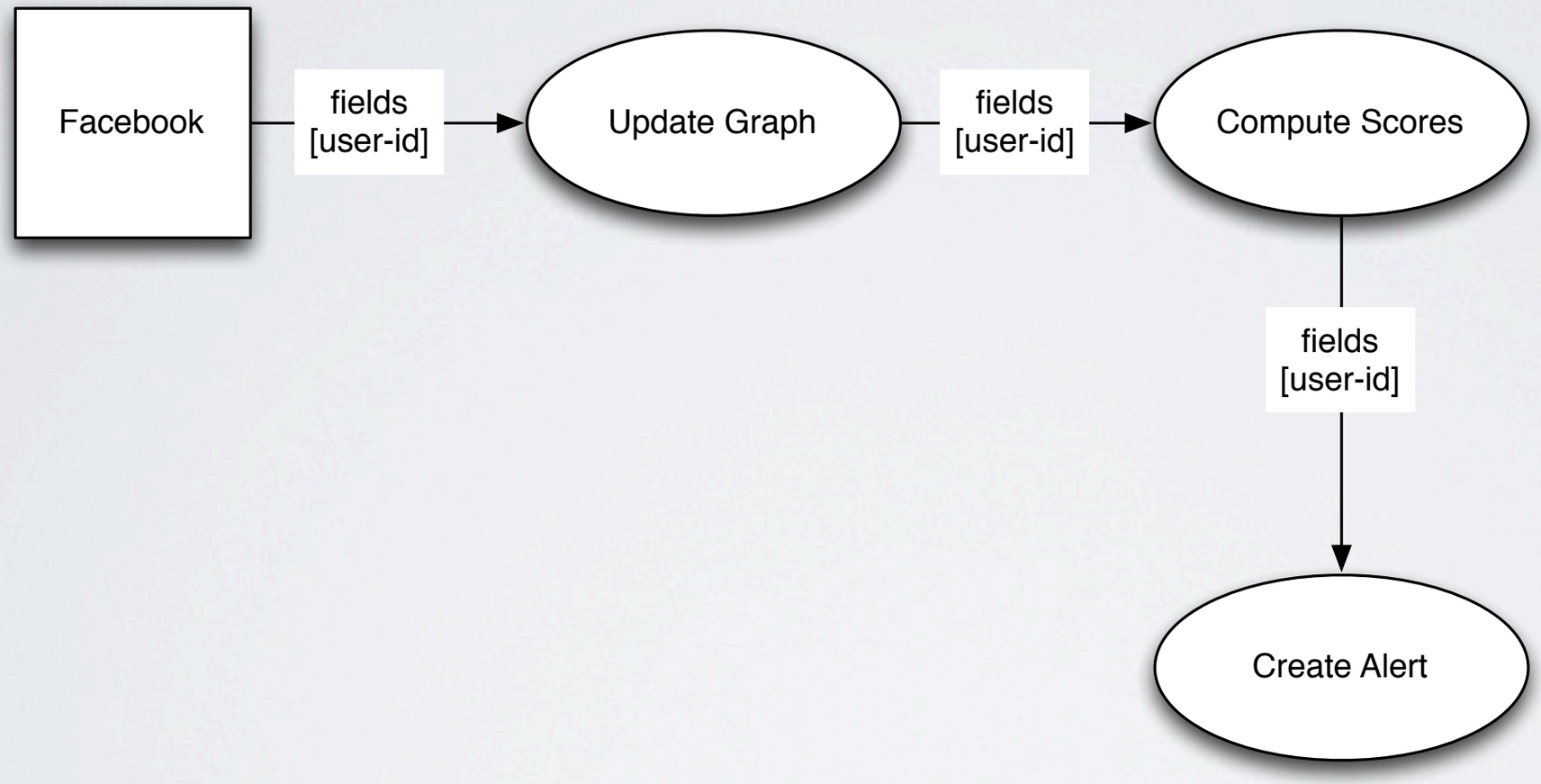
topology builder

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
        .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .shuffleGrouping("exclaim1");
```

- “words” is the name, 10 is parallelism, # of tasks across the cluster
- “exclaim1” is name of bolt, 3 is parallelism, # of tasks, reads from “words” via shuffle grouping
- consumer decides what to accept

zolodeck snippet

zolodeck snippet



submit the topology

config: specify the number of JVM process on the cluster

submit the topology

nimbus manages the cluster and the job

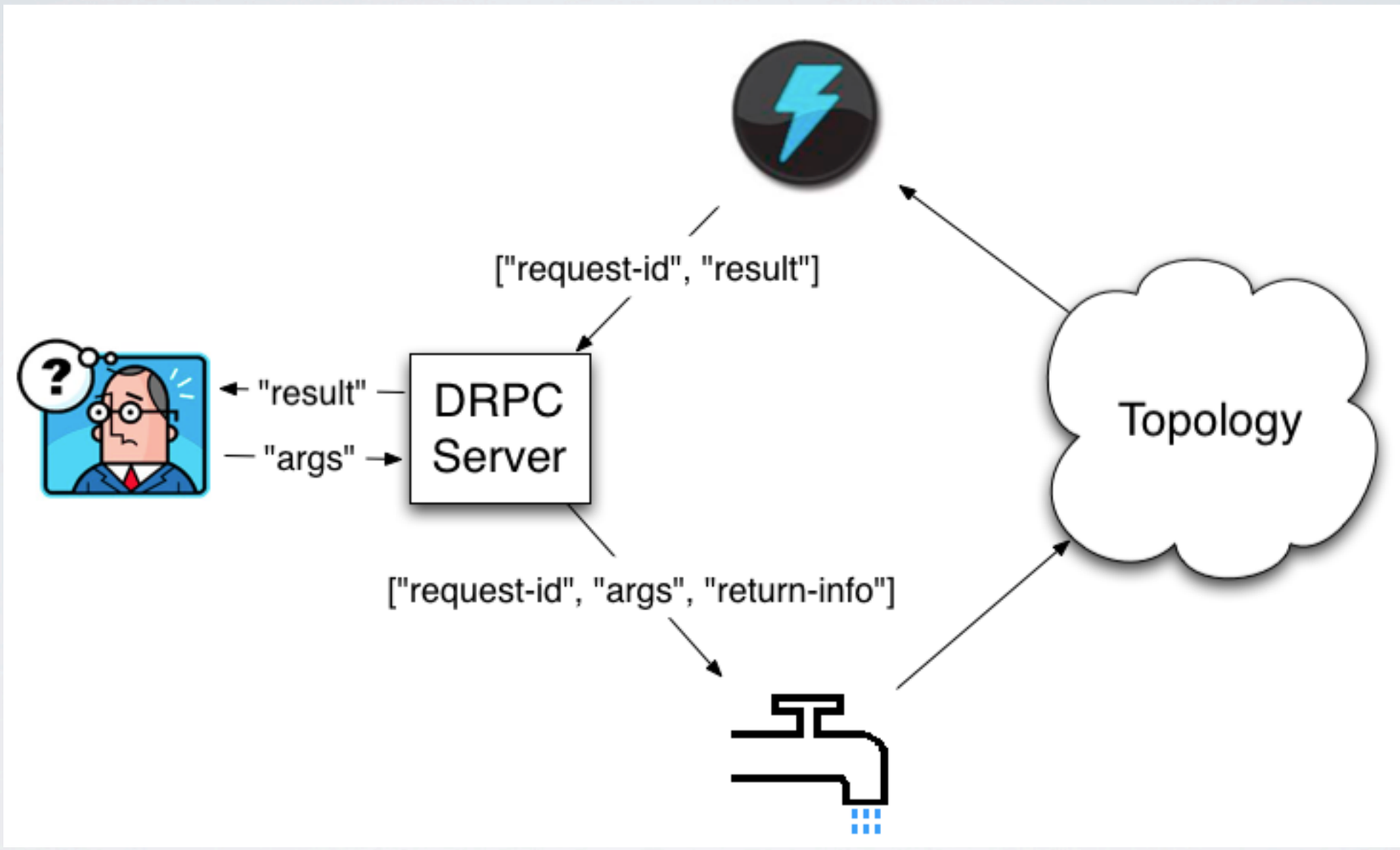
config: specify the number of JVM process on the cluster

local mode

local mode

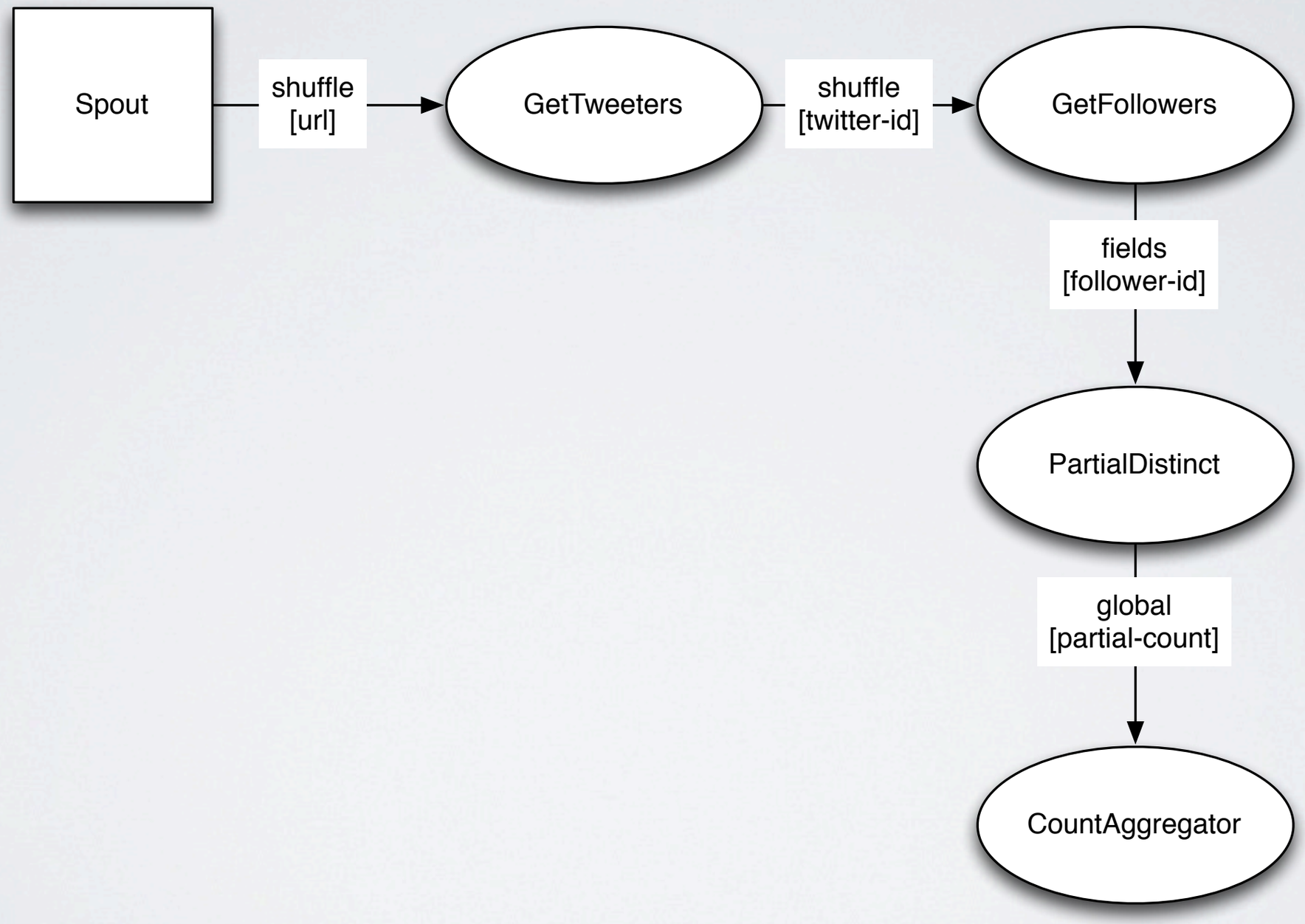
unit testing

distributed RPC



- server acts as spout
- creates a stream of function invocations (tuple - id of req, args, return info (host, port))
-

get-reach(url)



get-reach(url)

```
LinearDRPCTopologyBuilder builder = new
LinearDRPCTopologyBuilder("reach");
builder.addBolt(new GetTweeters(), 3);
builder.addBolt(new GetFollowers(), 12)
        .shuffleGrouping();
builder.addBolt(new PartialUniquer(), 6)
        .fieldsGrouping(new Fields("id", "follower"));
builder.addBolt(new CountAggregator(), 2)
        .fieldsGrouping(new Fields("id"));
```

- request ID is the parameter based on which an initial request is broken up, and sub tasks are run
- linearDRPCTopologyBuilder knows when to call “finish” on each batch

storm-deploy

single script deployment of storm cluster on AWS

storm

at-least once guarantee
tuple tree
tell storm - adding a new edge to tuple tree
tell storm - acknowledging a tuple is processed
tuple tree - 20 bytes for each, regardless of size of tuple tree

storm

guaranteed processing
easy horizontal scalability
fault-tolerance

at-least once guarantee
tuple tree
tell storm - adding a new edge to tuple tree
tell storm - acknowledging a tuple is processed
tuple tree - 20 bytes for each, regardless of size of tuple tree

storm

guaranteed processing
easy horizontal scalability
fault-tolerance

focus on business logic

at-least once guarantee
tuple tree
tell storm - adding a new edge to tuple tree
tell storm - acknowledging a tuple is processed
tuple tree - 20 bytes for each, regardless of size of tuple tree

storm

guaranteed processing
easy horizontal scalability
fault-tolerance

focus on business logic

stream processing
distributed RPC
continuous computation

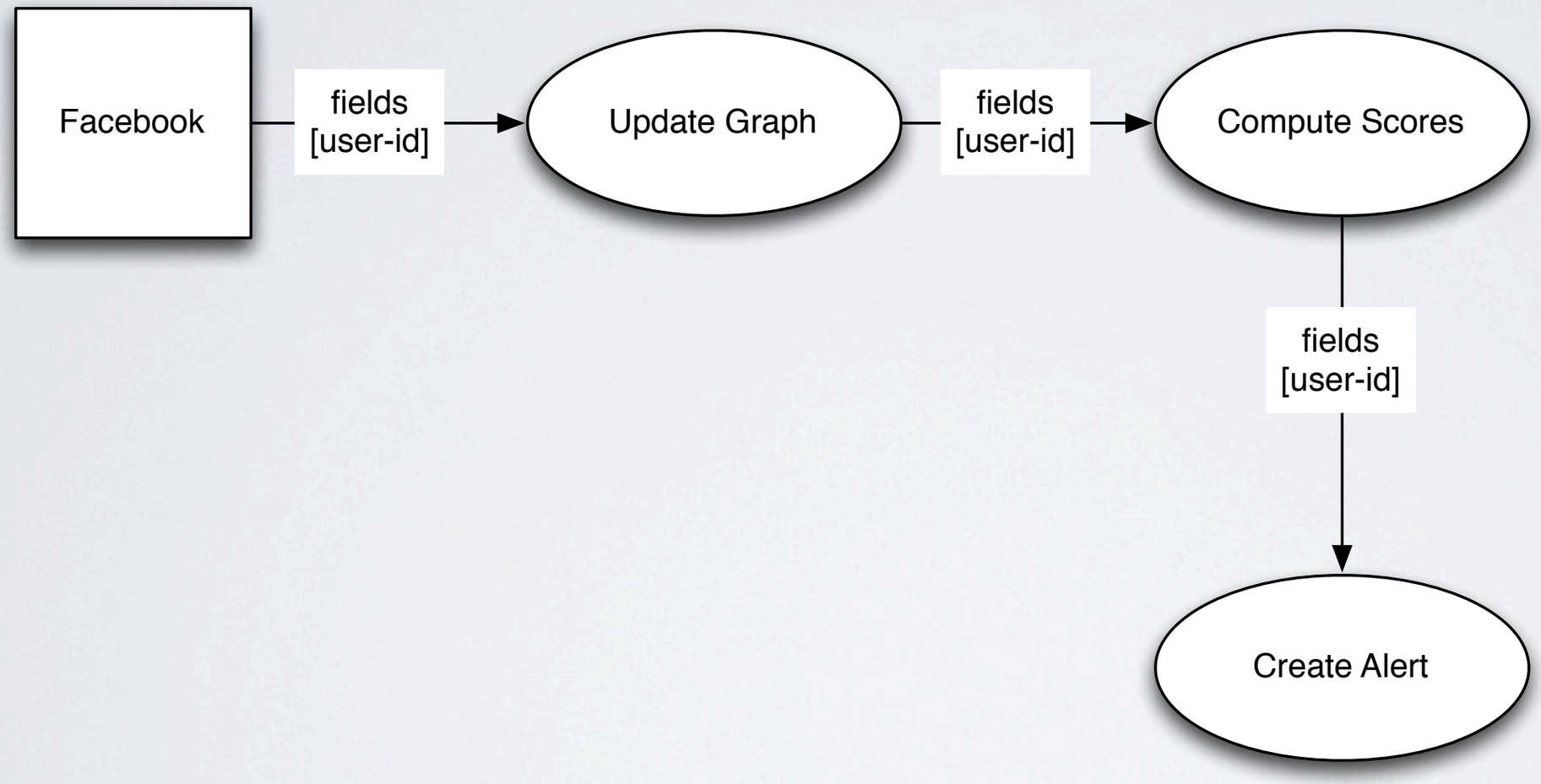
at-least once guarantee
tuple tree
tell storm - adding a new edge to tuple tree
tell storm - acknowledging a tuple is processed
tuple tree - 20 bytes for each, regardless of size of tuple tree

so that's the processing story

used by Twitter, and a whole bunch of other companies
1M messages/sec

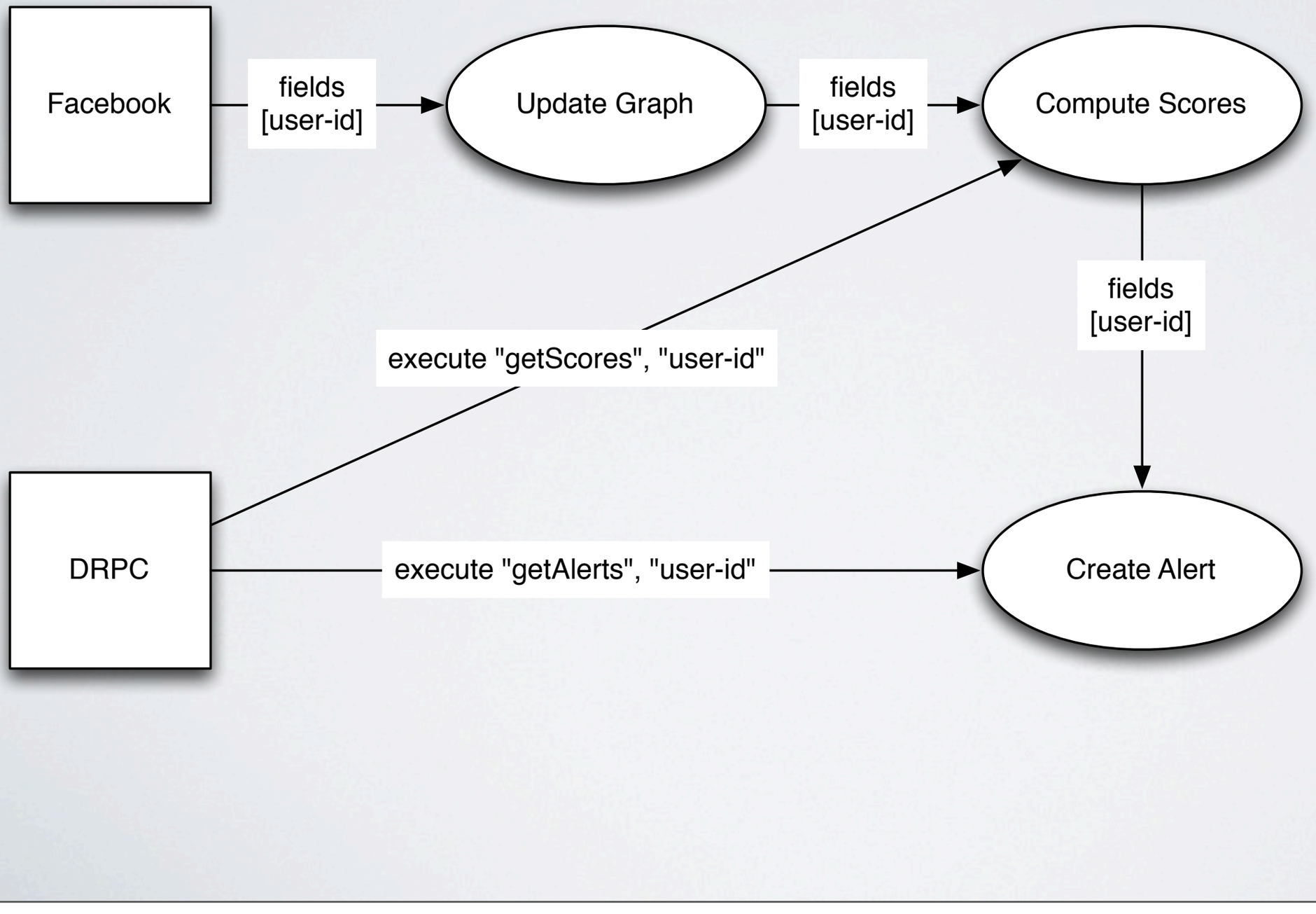
zolodeck example

zolodeck snippet

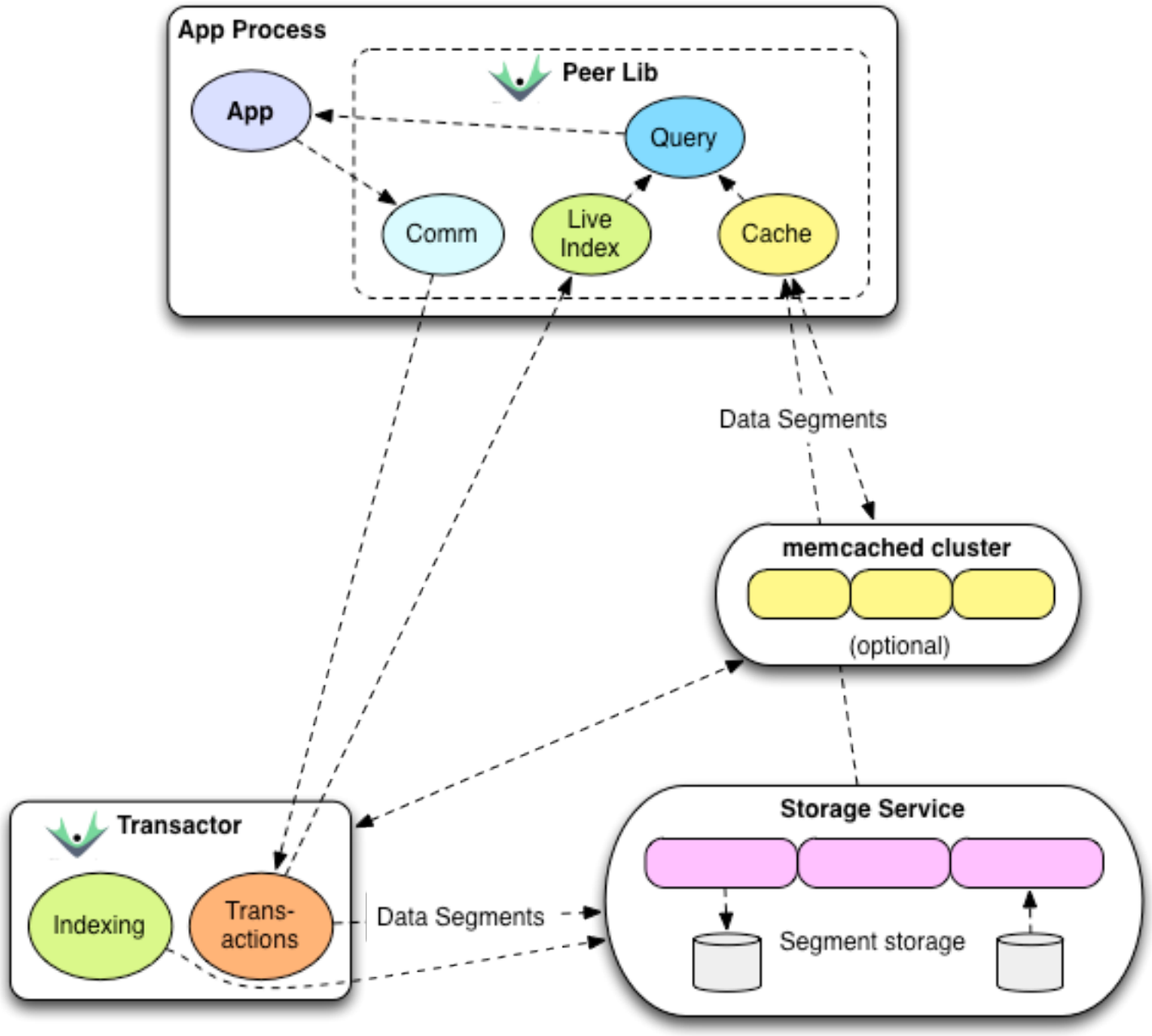


zolodeck

zolodeck snippet

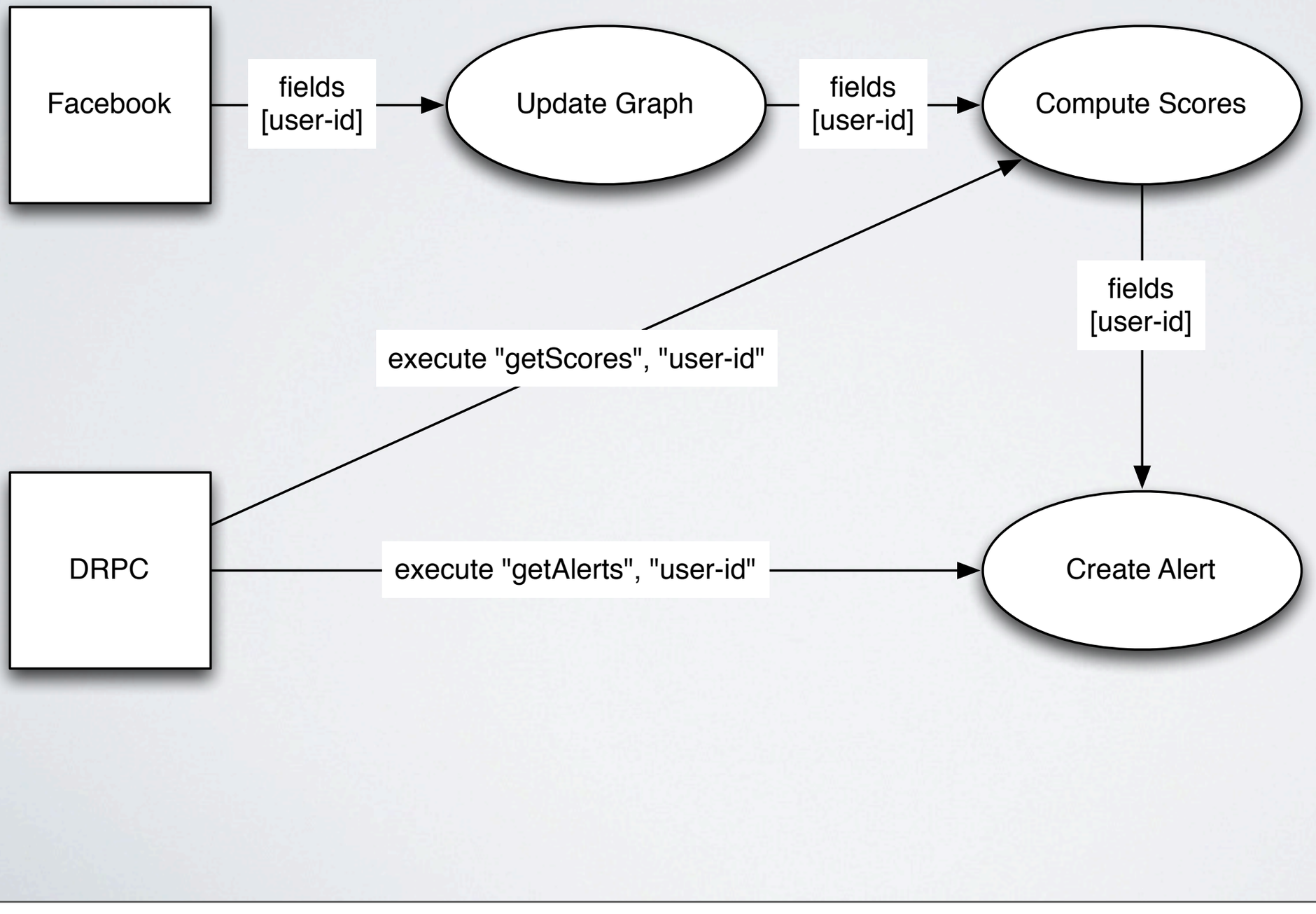


zolodeck



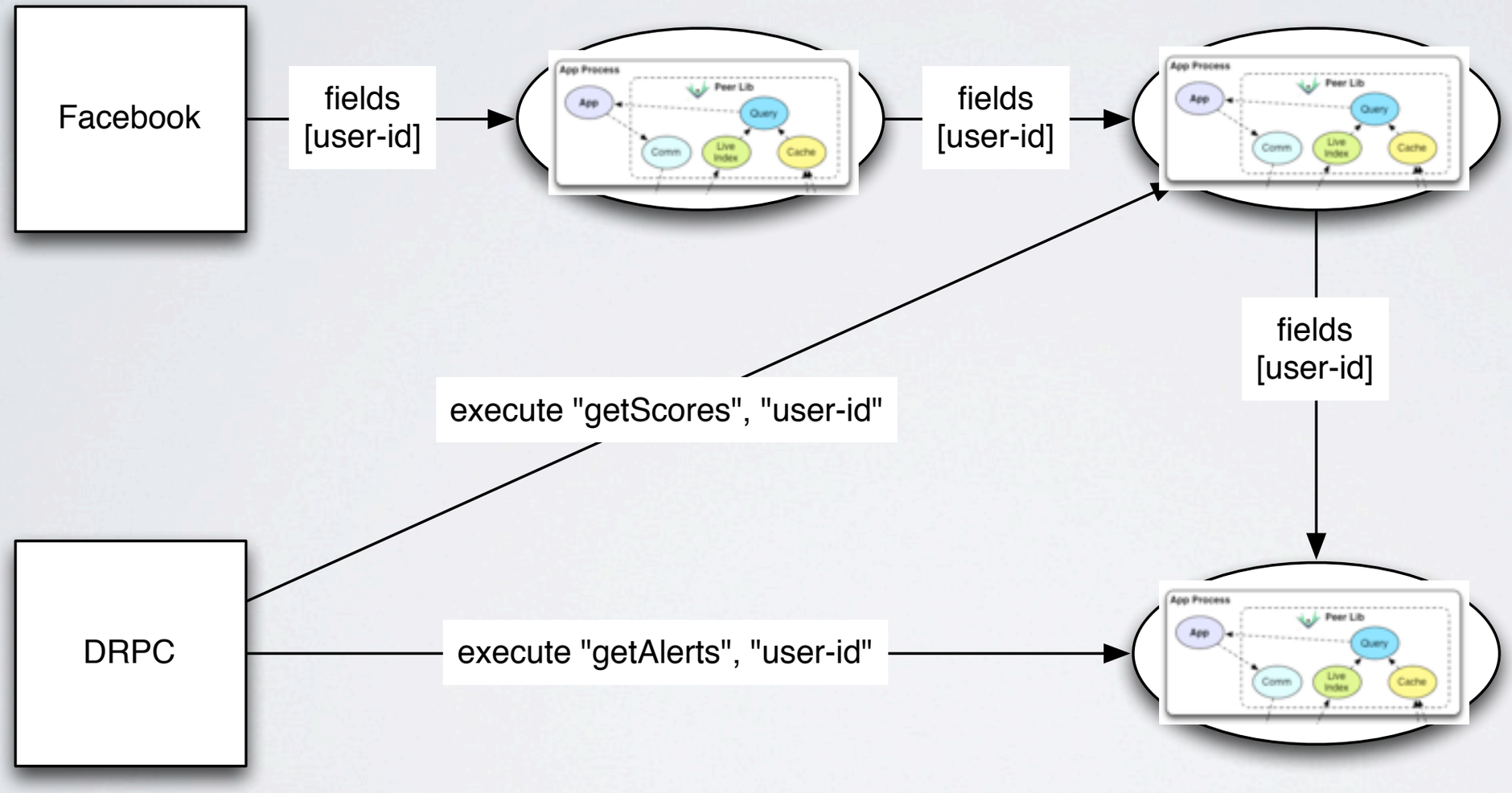
recap: the datomic way

zolodeck snippet



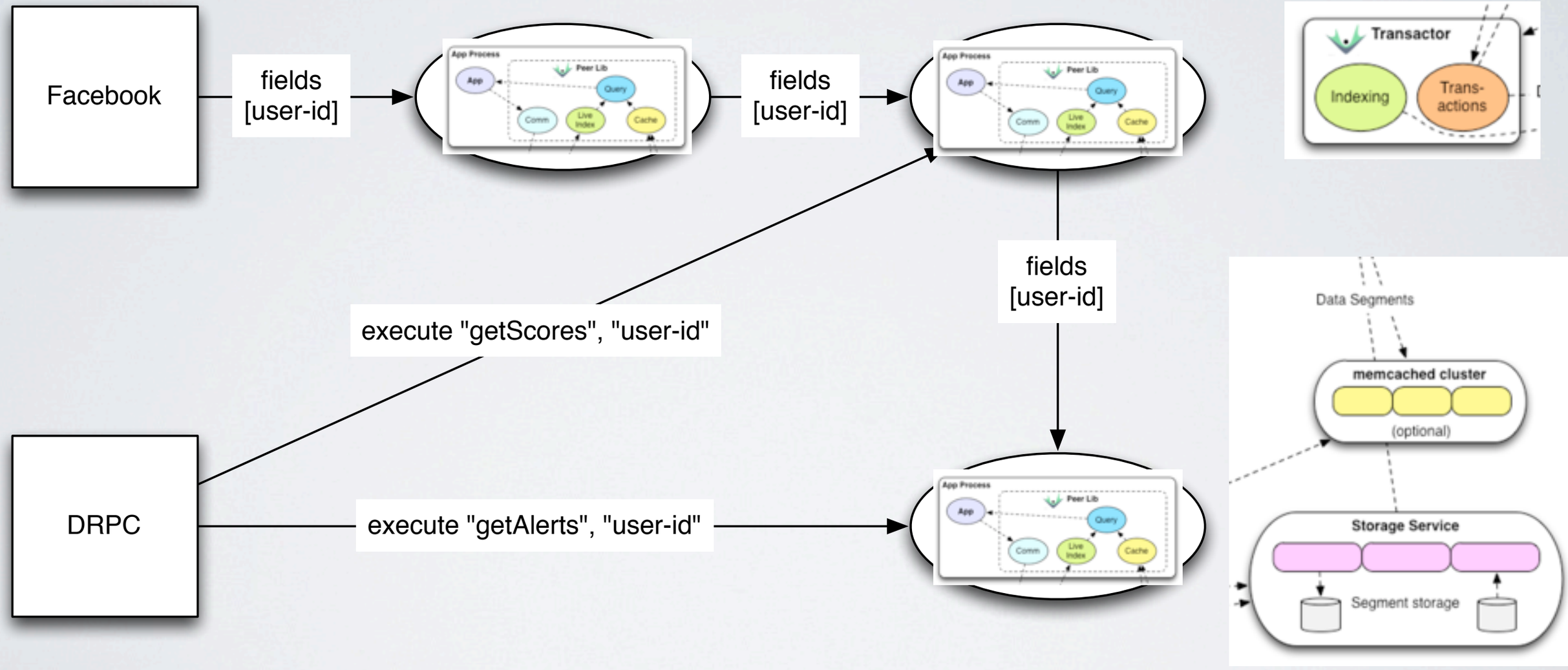
zolodeck

zolodeck snippet



zolodeck

zolodeck snippet



zolodeck

clojure

functional language
modern Lisp
concurrent programming
JVM

simplicity

simplicity

functional abstractions

simplicity

functional abstractions
easy to test

simplicity

functional abstractions
easy to test
immutability

simplicity

functional abstractions
easy to test
immutability
macro system

datomic speaks clojure

written in Clojure
schema, transactions, datalog are data-structures

storm speaks Clojure

storm speaks Clojure

50% written in Clojure
beautiful DSL

zolodeck

datomic

consistent, scalable
no sharding
built-in caching

datalog power
db is local
peer isolation
schema flexibility
time travel

configuration-free
cloud-ready

storm

guaranteed processing
easy horizontal scalability
fault-tolerance

focus on business logic

stream processing
distributed RPC
continuous computation

- analytics peers won't bog down transaction processing
- datalog is truly powerful, whole another topic

zolodeck = datomic + storm + clojure

zolodeck = datomic + storm + clojure

work locally

zolodeck = datomic + storm + clojure

work locally
start small

zolodeck = datomic + storm + clojure

work locally
start small
scale effortlessly

zolodeck = datomic + storm + clojure

work locally
start small
scale effortlessly
~ no ops

zolodeck = datomic + storm + clojure

work locally
start small
scale effortlessly
~ no ops
focus on business logic

zolodeck = datomic + storm + clojure

work locally
start small
scale effortlessly
~ no ops
focus on business logic

become an networking extraordinaire

thanks

zolodeck.com

Your personal virtual assistant