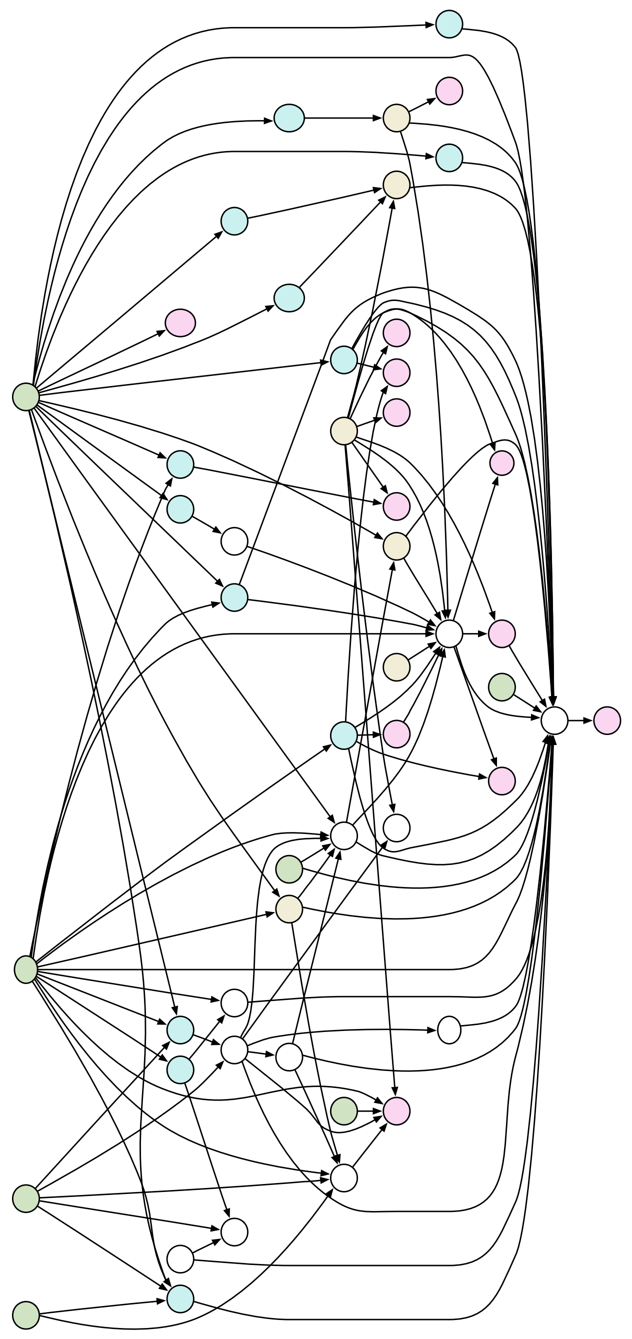
Graph: composable production systems in Clojure

Jason Wolfe (@w01fe)

Strange Loop '12

Motivation

- Interesting software has:
  - many components
  - complex web of dependencies
- Developers want:
  - simple, factored code
  - easy testability
  - tools for monitoring and debugging



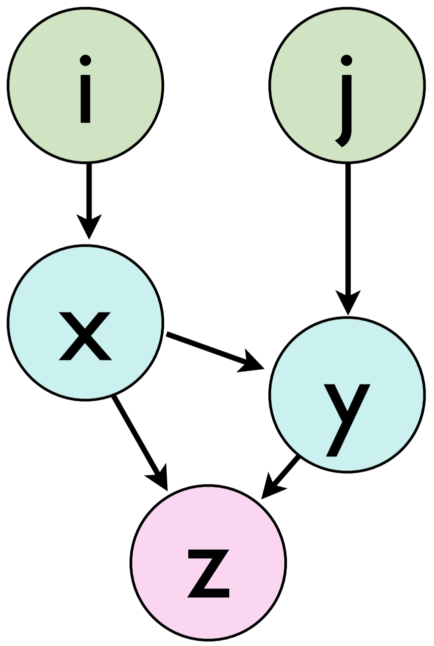
Graph

- Graph is a simple, declarative way to express system composition

- A Graph is just a map of functions that can depend on previous outputs

- Graphs are easy to create, reason about, test, and build upon

```
{:x (fnk [i] ...)
 :y (fnk [j x] ...)
 :z (fnk [x y] ...)}
```



input

```
{:i 1
 :j 2}
```

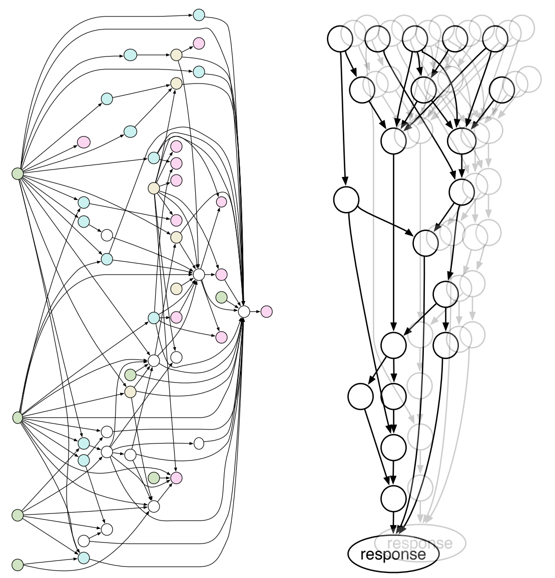
output

```
{:x 2
 :y 5
 :z 12}
```

Outline

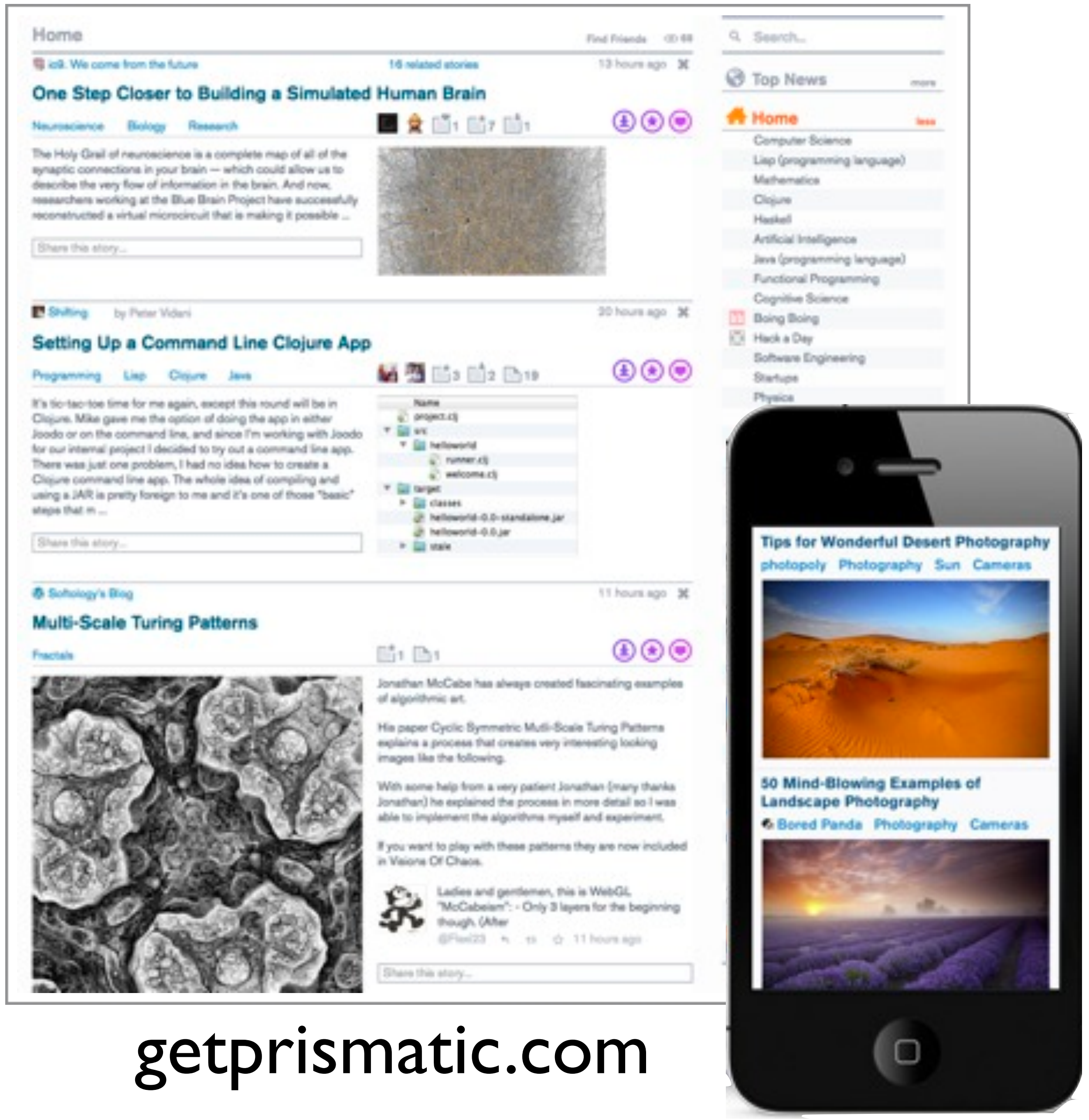
- Prismatic
- Design Goals
- Graph: specs and compilation
- Applications
    - newsfeed generation
    - production services

```
{:x (fnk [i] ...)
 :y (fnk [j x] ...)
 :z (fnk [x y] ...)}
```



Prismatic

- Personalized, interest-based newsfeeds
- Build crawlers, topic models, graph analysis, story clustering, ...
- Backend 99.9% Clojure
- Personalized ranked feeds in real-time (~200ms)



Home

Find Friends

io9. We come from the future

16 related stories

13 hours ago

One Step Closer to Building a Simulated Human Brain

Neuroscience Biology Research

The Holy Grail of neuroscience is a complete map of all of the synaptic connections in your brain — which could allow us to describe the very flow of information in the brain. And now, researchers working at the Blue Brain Project have successfully reconstructed a virtual microcircuit that is making it possible ...

Share this story...

Shifting by Peter Vidani

20 hours ago

Setting Up a Command Line Clojure App

Programming Lisp Clojure Java

It's tic-tac-toe time for me again, except this round will be in Clojure. Mike gave me the option of doing the app in either Joodo or on the command line, and since I'm working with Joodo for our internal project I decided to try out a command line app. There was just one problem, I had no idea how to create a Clojure command line app. The whole idea of compiling and using a JAR is pretty foreign to me and it's one of those "basic" steps that m ...

```
Name
  project.clj
▼ src
  ▼ helloworld
      runner.clj
      welcome.clj
▼ target
  ▶ classes
    helloworld-0.0-standalone.jar
    helloworld-0.0.jar
  ▶ stale
```

Share this story...

Sohology's Blog

11 hours ago

Multi-Scale Turing Patterns

Fractals

Jonathan McCabe has always created fascinating examples of algorithmic art.

His paper Cyclic Symmetric Multi-Scale Turing Patterns explains a process that creates very interesting looking images like the following.

With some help from a very patient Jonathan (many thanks Jonathan) he explained the process in more detail so I was able to implement the algorithms myself and experiment.

If you want to play with these patterns they are now included in Visions Of Chaos.

Ladies and gentlemen, this is WebGL "McCabeism": - Only 3 layers for the beginning though. (After

11 hours ago

Share this story...

Search...

Top News more

Home less

Computer Science
Lisp (programming language)
Mathematics
Clojure
Haskell
Artificial Intelligence
Java (programming language)
Functional Programming
Cognitive Science
Boing Boing
Hack a Day
Software Engineering
Startups
Physics

Tips for Wonderful Desert Photography

photopoly Photography Sun Cameras

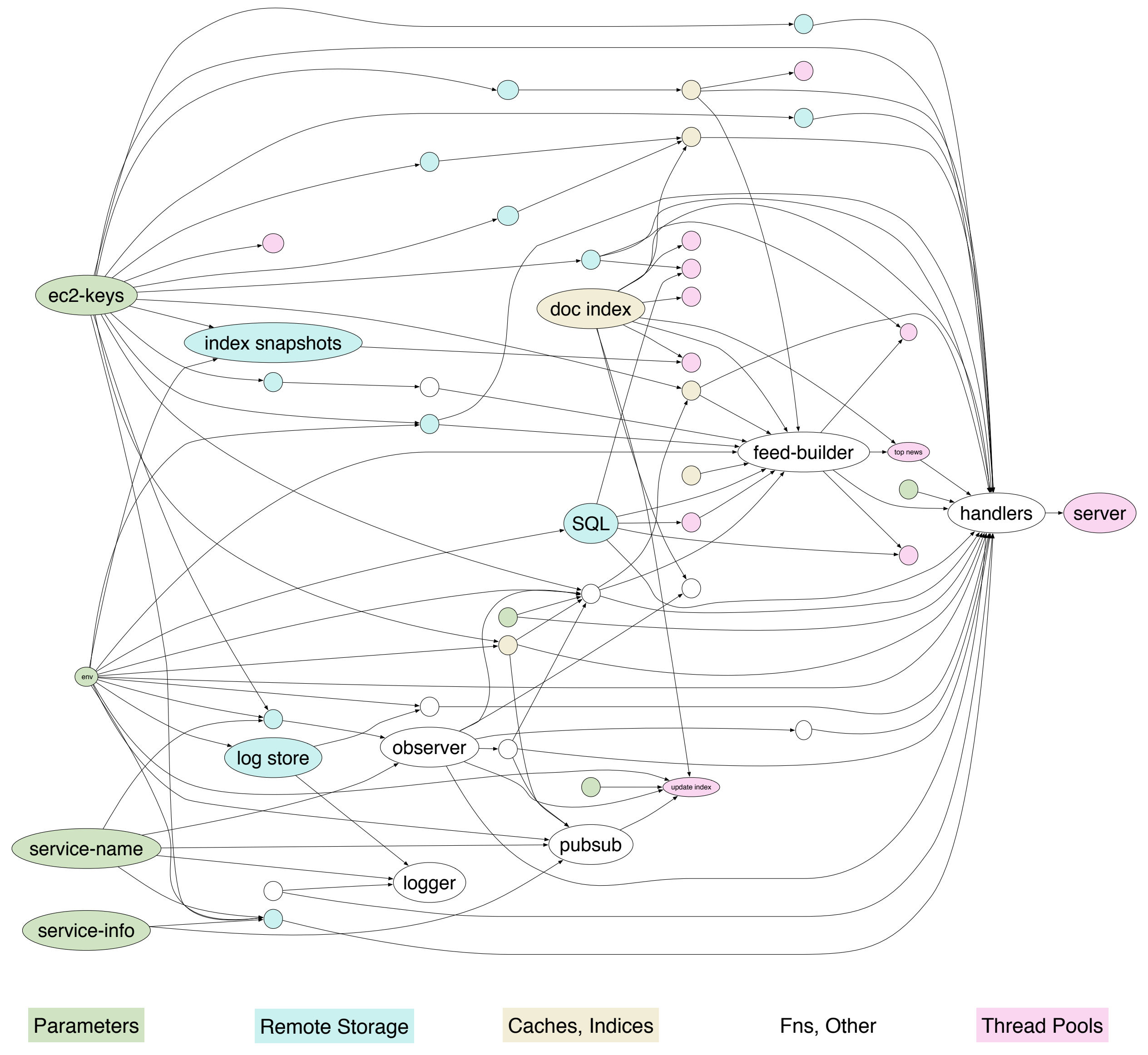
50 Mind-Blowing Examples of Landscape Photography

Bored Panda Photography Cameras

getprismatic.com

Prismatic's production API service

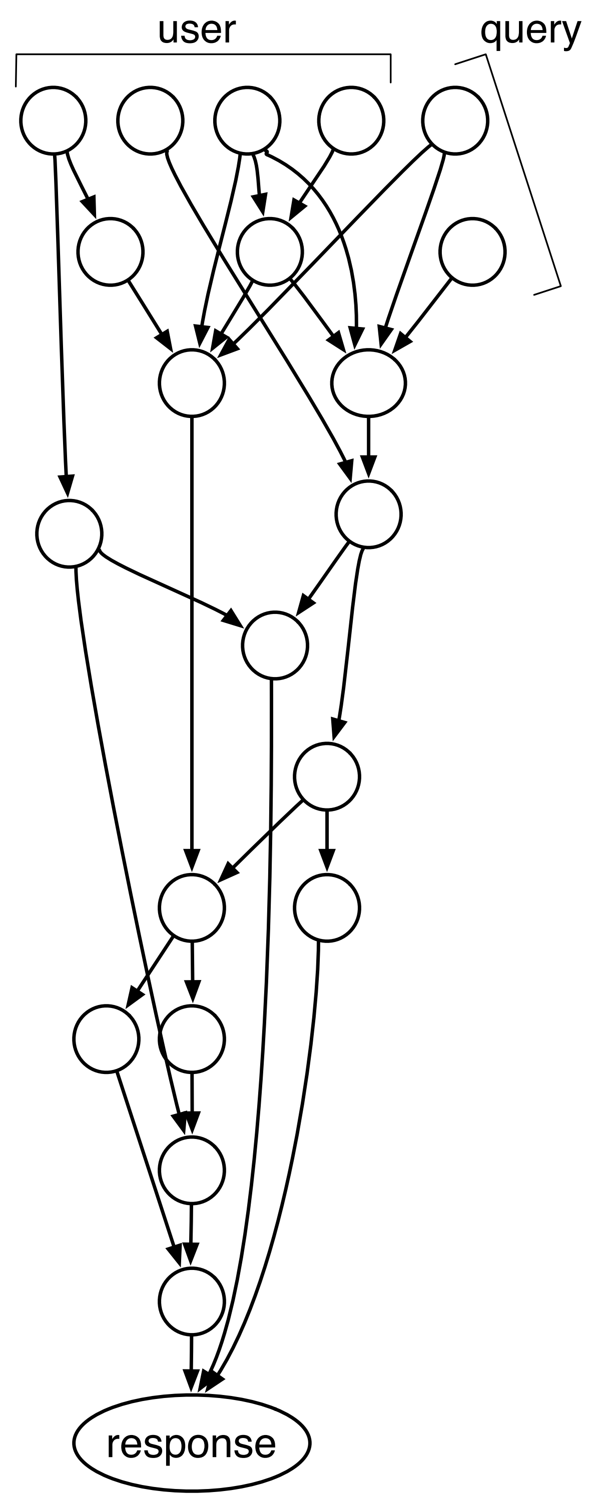
- >100 components
  - storage systems
  - caches & indices
  - ranking algorithms
- Coordinate in intricate dance to serve feeds *fast*
- Relentlessly refactored
- Still dozens of top-level components in complex dependency network



The feed builder

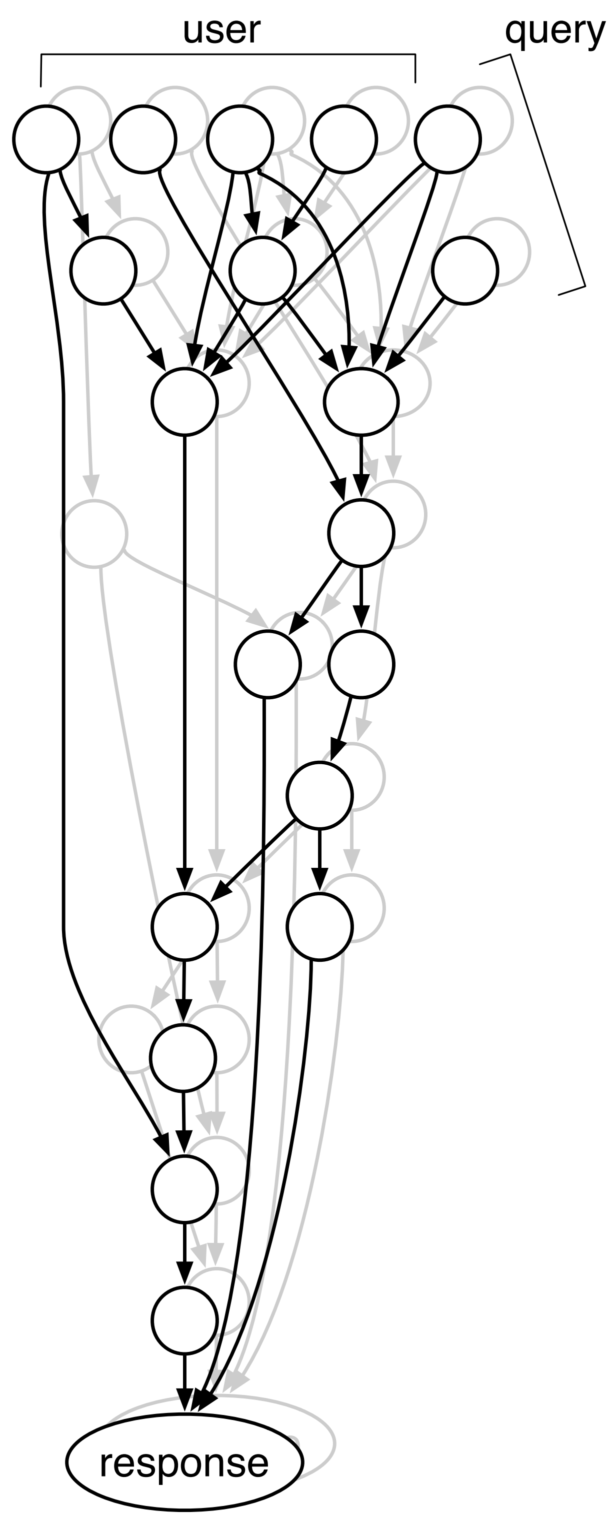
- 20+ steps from query to personalized ranking, 20+ parameters

- Not a simple pipeline



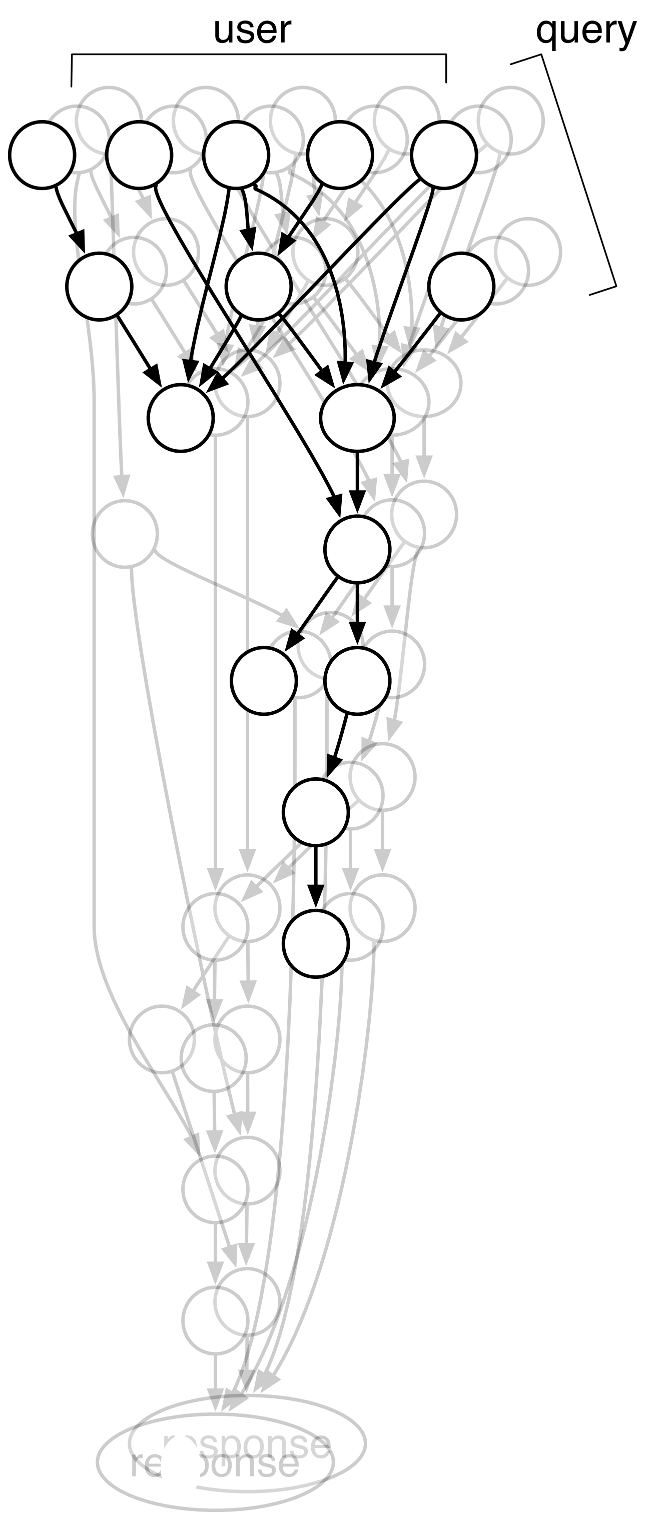
The feed builder

- 20+ steps from query to personalized ranking, 20+ parameters
- Not a simple pipeline
- > 10 feed types w/ slightly different steps, configurations



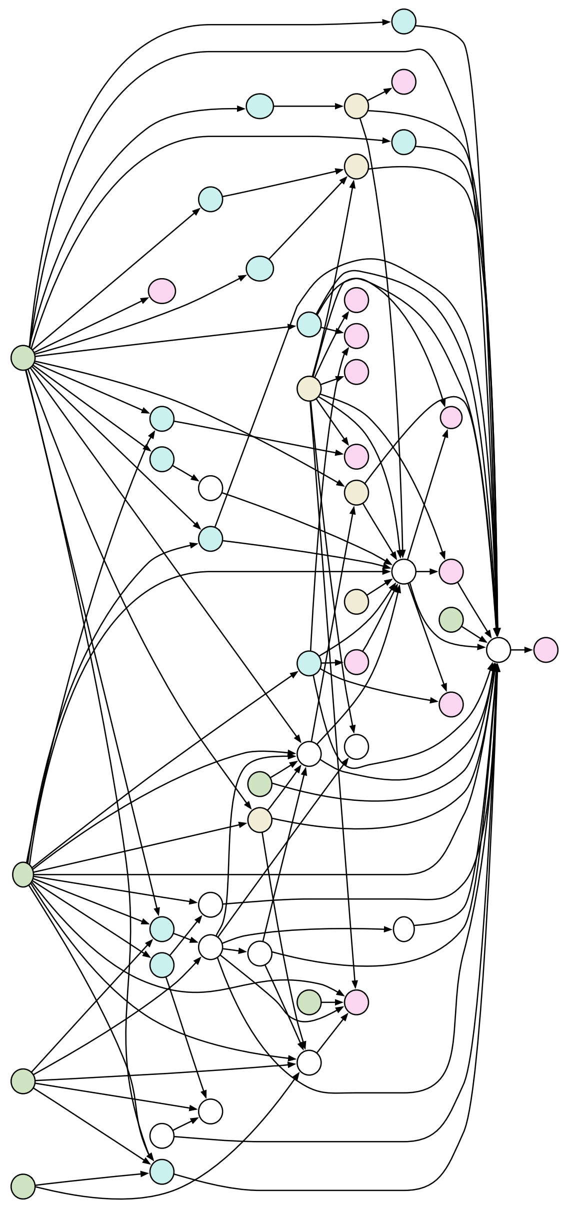
The feed builder

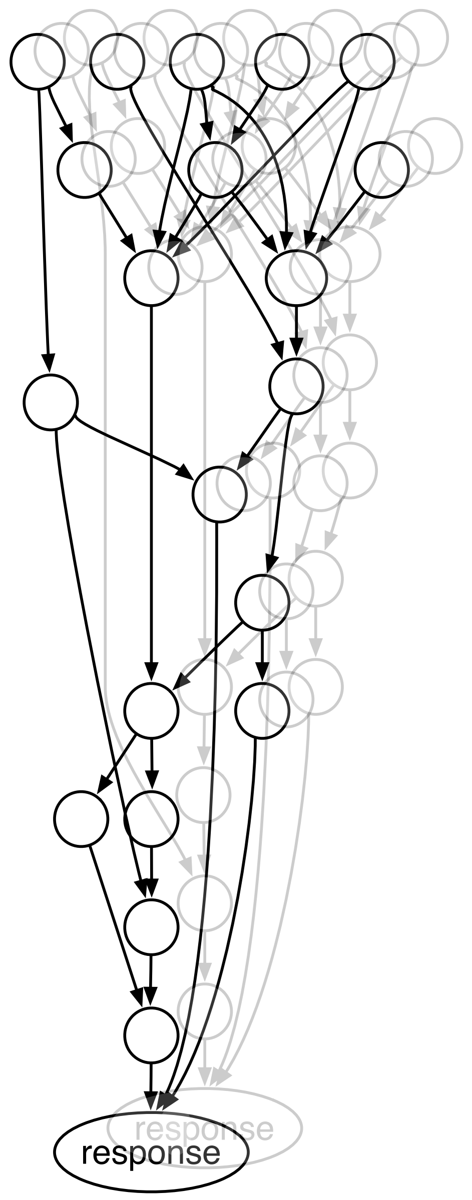
- 20+ steps from query to personalized ranking, 20+ parameters
- Not a simple pipeline
- > 10 feed types w/ slightly different steps, configurations
- Support for early stopping



Theme: complexity of composition

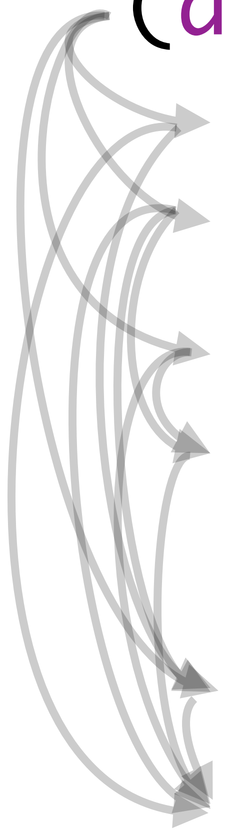
- Previous implementations: `defn`s with huge `let`s
- Unwieldy for large systems with complex or polymorphic dependencies
- Hard to test, debug, and monitor





The ‘monster `let`’

- Tens of parameters, not compositional
- Mocks/polymorphic flow difficult
- Ad hoc monitoring & shutdown logic per item
- Core issue: structure of (de)composition is locked up in an opaque function



```
(defn start [{:keys [a,z]}]
  (let [s1 (store a ...)
        s2 (store b ...)
        db (sql-db c)
        t2 (cron s2 db...)
         ...
        srv (server ...)]
    (fn shutdown []
      (.stop srv)
      ...
      (.flush s1))))
```

Prismatic software engineering philosophy

- Fine-grained, composable abstractions (FCA)

Libraries >> Frameworks

- Strive for simplicity, work with the language
- Graph is a FCA for composition

Goal: declarative

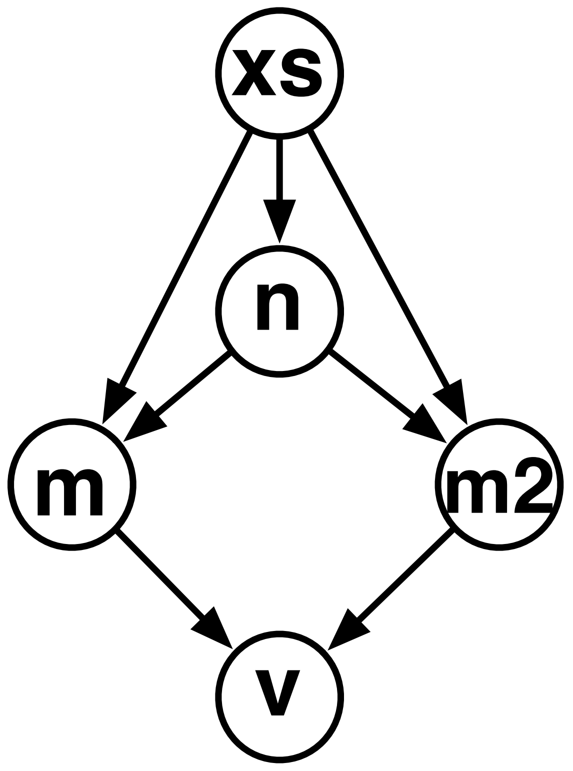
- Declarative specifications fix ‘monster `let`’
  - Explicitly list components, dependencies
  - Enable abstractions over components, reasoning about composition
- Not new: Pregel, Dryad, Storm, ...

Goal: simple

- Distill this idea to its simplest, most idiomatic expression
  - a Graph spec is just a (Clojure) map
  - no XML files or interface hell
- Graphs are ordinary data
  - manipulate them ‘for free’
  - --> unexpected applications

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures. - Alan Perlis

From ‘let’ to Graph


$$\begin{tikzcd}
 & xs \arrow[d] \arrow[dl] \arrow[dr] & \\
 & n \arrow[dl] \arrow[dr] & \\
m \arrow[dr] & & m2 \arrow[dl] \\
 & v &
\end{tikzcd}$$

```
(defn stats [{:keys [xs]}]
  (let [n  (count xs)
        m  (/ (sum xs) n)
        m2 (/ (sum sq xs) n)
        v  (- m2 (* m m))]
    {:n n :m m :m2 m2 :v v}))

{:n  (fnk [xs]    (count xs))
 :m  (fnk [xs n] (/ (sum xs) n))
 :m2 (fnk [xs n] (/ (sum sq xs) n))
 :v  (fnk [m m2] (- m2 (* m m)))}
```

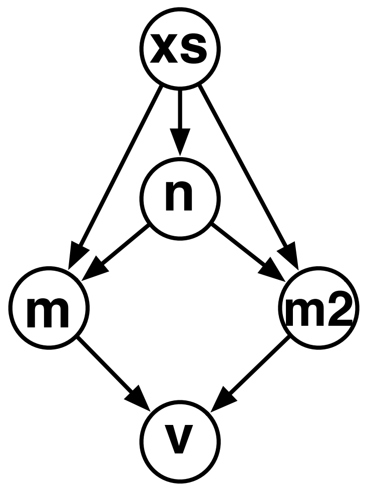
Bring on the fnk

- fnk = keyword function
- Similar to {:keys []} destructuring
  - nicer opt. arg. support
  - asserts that keys exist
  - metadata about args
- Quite useful in itself
- Only macros in Graph

```
(defnk foo [x y [s 1]]
  (+ x (* y s)))

(= 8 (foo {:x 2 :y 3 :s 2}))

(= 5 (foo {:x 2 :y 3}))

(thrown? Ex. (foo {:x 2}))

(= (meta foo)
   {:req-ks #{:x :y}}
    :opt-ks #{:s})
```

A Graph Specification

- A Graph is just a map from keywords to fnks
- Required keys of each fnk specify graph relationships
  - Entire graph specifies a fnk to map of results

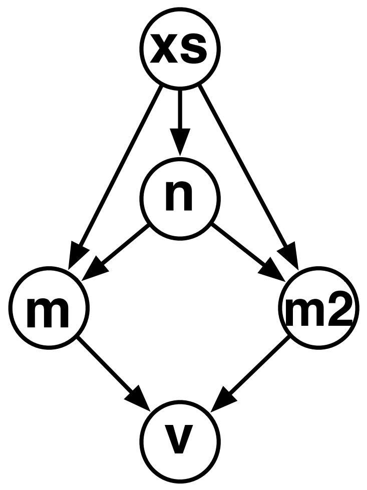


```
{:n  (fnk [xs]
       (count xs))
 :m  (fnk [xs n]
       (/ (sum xs) n))
 :m2 (fnk [xs n]
       (/ (sum sq xs) n))
 :v  (fnk [m m2]
       (- m2 (* m m)))}
```

A Graph Specification

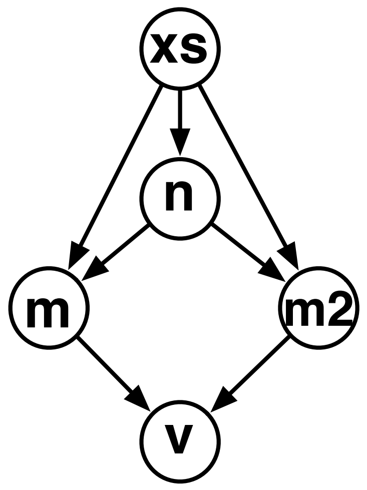
- A Graph is just a map from keywords to fnks
- Required keys of each fnk specify graph relationships
  - Entire graph specifies a fnk to map of results

```
{:xs [1 2 3 6]}

{:n  (fnk [xs]
       (count xs))
 :m  (fnk [xs n]
       (/ (sum xs) n))
 :m2 (fnk [xs n]
       (/ (sum sq xs) n))
 :v  (fnk [m m2]
       (- m2 (* m m)))}
```


$$\begin{tikzcd} & xs \arrow[d] \arrow[dl] \arrow[dr] & \\ & n \arrow[dl] \arrow[dr] & \\ m \arrow[dr] & & m2 \arrow[dl] \\ & v & \end{tikzcd}$$

A Graph Specification

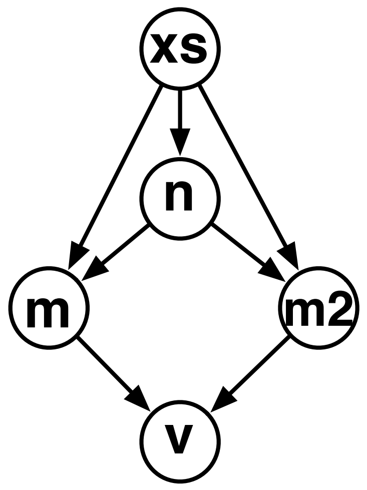
- A Graph is just a map from keywords to fnks
- Required keys of each fnk specify graph relationships
  - Entire graph specifies a fnk to map of results



```
{:xs [1 2 3 6]}

{:n  4

  :m  (fnk [xs n]
        (/ (sum xs) n))
  :m2 (fnk [xs n]
        (/ (sum sq xs) n))
  :v  (fnk [m m2]
        (- m2 (* m m)))}
```

A Graph Specification

- A Graph is just a map from keywords to fnks
- Required keys of each fnk specify graph relationships
  - Entire graph specifies a fnk to map of results

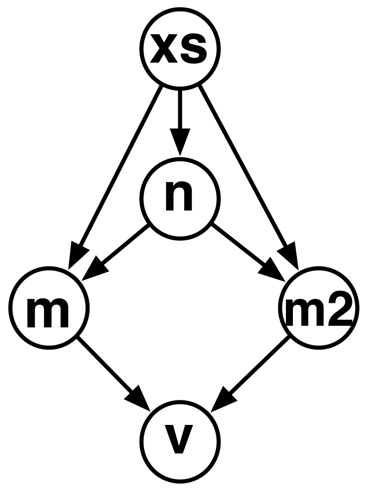


```
{:xs [1 2 3 6]}

{:n  4

  :m  3

  :m2 (fnk [xs n]
         (/ (sum sq xs) n))
  :v  (fnk [m m2]
         (- m2 (* m m)))}
```

A Graph Specification

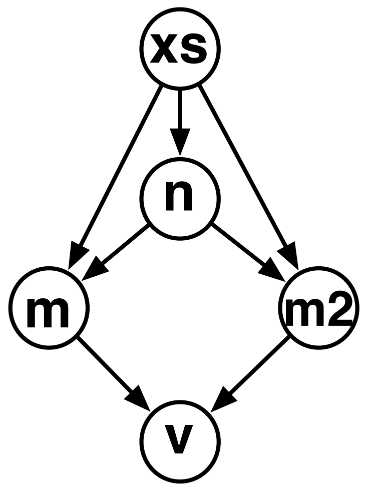
- A Graph is just a map from keywords to fnks
- Required keys of each fnk specify graph relationships
  - Entire graph specifies a fnk to map of results

```
{:xs [1 2 3 6]}

{:n  4

 :m  3

 :m2 12.5

 :v  (fnk [m m2]
       (- m2 (* m m)))}
```



A Graph Specification

- A Graph is just a map from keywords to fnks
- Required keys of each fnk specify graph relationships
  - Entire graph specifies a fnk to map of results



```
{:xs [1 2 3 6]}

{:n  4

 :m  3

 :m2 12.5

 :v  3.5
                         }
```

Compiling Graphs

- Compile graph to fnk that returns map of outputs

```
(def g
  {:n  (fnk [xs]    ...)
   :m  (fnk [xs n] ...)
   :m2 (fnk [xs n] ...)
   :v  (fnk [m m2] ...)})

(def stats
  (compile g))

(= (stats {:xs [1 2 3 6]})
   {:n 4       :m 3
    :m2 12.5 :v 3.5})
```

Compiling Graphs

- Compile graph to fnk that returns map of outputs
    - error checked

```
(def g
  {:n  (fnk [xs]   ...)
   :m  (fnk [xs n] ...)
   :m2 (fnk [xs n] ...)
   :v  (fnk [m m2] ...)})

(def stats
  (compile g))

(thrown?
   (Ex. “missing :xs”)
   (stats {:x 1}))
```

Compiling Graphs

- Compile graph to fnk that returns map of outputs
  - error checked
  - can return lazy map

```
(def g
  {:n  (fnk [xs]   ...)
   :m  (fnk [xs n] ...)
   ~~:m2 (fnk [xs n] ...)~~
   ~~:v  (fnk [m m2] ...)~~}})

(def stats
  (lazy-compile g))

(= (:m (stats {:xs [1 5]}))
   3)
```

Compiling Graphs

- Compile graph to fnk that returns map of outputs
  - error checked
  - can return lazy map
  - can auto-parallelize

```
(def g
  {:n  (fnk [xs]   ...)
   :m  (fnk [xs n] ...)
   :m2 (fnk [xs n] ...)
   :v  (fnk [m m2] ...)})

(def stats
  (par-compile g))

(= (:v (stats {:xs [1 5]}))
   3.5)
```

Compiling Graphs

- Compile graph to fnk that returns map of outputs
  - error checked
  - can return lazy map
  - can auto-parallelize

```
(def g
  {:n  2
   :m  (fnk [xs n] ...)
   :m2 (fnk [xs n] ...)
   :v  (fnk [m m2] ...)})

(def stats
  (par-compile g))

(= (:v (stats {:xs [1 5]}))
   3.5)
```

Compiling Graphs

- Compile graph to fnk that returns map of outputs
  - error checked
  - can return lazy map
  - can auto-parallelize

```
(def g
  {:n  2
   :m  3
   :m2 13
   :v  (fnk [m m2] ...)})

(def stats
  (par-compile g))

(= (:v (stats {:xs [1 5]}))
   3.5)
```

Compiling Graphs

- Compile graph to fnk that returns map of outputs
    - error checked
    - can return lazy map
    - can auto-parallelize

```
(def g
  {:n  2
   :m  3
   :m2 13
   :v  4                    })

(def stats
  (par-compile g))

(= (:v (stats {:xs [1 5]}))
   3.5)
```

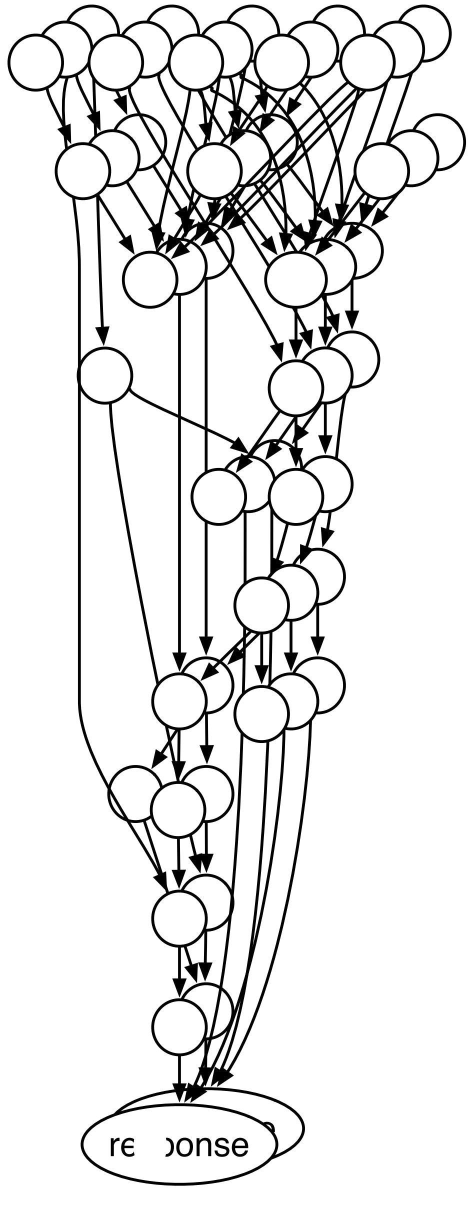
Compiling Graphs

- Compile graph to fnk that returns map of outputs
  - error checked
  - can return lazy map
  - can auto-parallelize
- With more tooling, also compile graphs to production services
- Could compile to cross-machine topologies, ...

```
(def g
  {:n  2
   :m  3
   :m2 13
   :v  4                   })

(def stats
  (par-compile g))

(= (:v (stats {:xs [1 5]}))
   3.5)
```

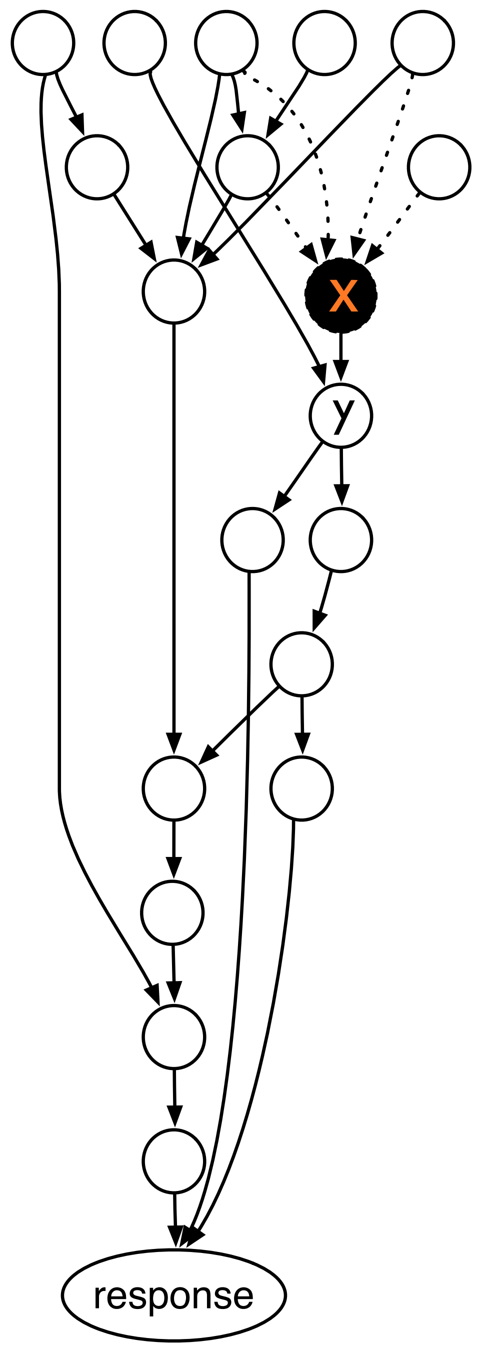
Before: feed builder

- Real-time personally ranked feeds
- 100-line fn expressed core composition logic, ~20 params
  - several nested lets, escape hatches
- Component polymorphism (10 flavors of feeds)
  - kludge of cases
  - ball of multimethods
  - protocols + hacks



Feed builder in Graph

- Default parameters
- Graph with 'holes' captures shared logic



```
(def default-params
  {:alpha 0.7
   ...
   :phasers :stun})


(def partial-graph
  {:query (fnk ...)
   ...
   :y (fnk [a x] ..)
   ...
   :resp (fnk ...)})
```

Feed builder in Graph

- Each feed type specifies
  - updated parameters
  - missing/new graph nodes
- To make feed fn, just
  - merge in updates
  - compile resulting graph

```
(def default-params ..)
(def partial-graph ..)

(def topic-feed
  (compile-feed-fn
    {:alpha 0.2}
    {:x (fnk ...)
     :q (fnk ...)}))
```

```
(defn compile-feed-fn [params nodes]
  (let [p (merge default-params params)
        g (compile (merge partial-graph nodes))]
    (fn feed [req] (g (merge p req)))))
```

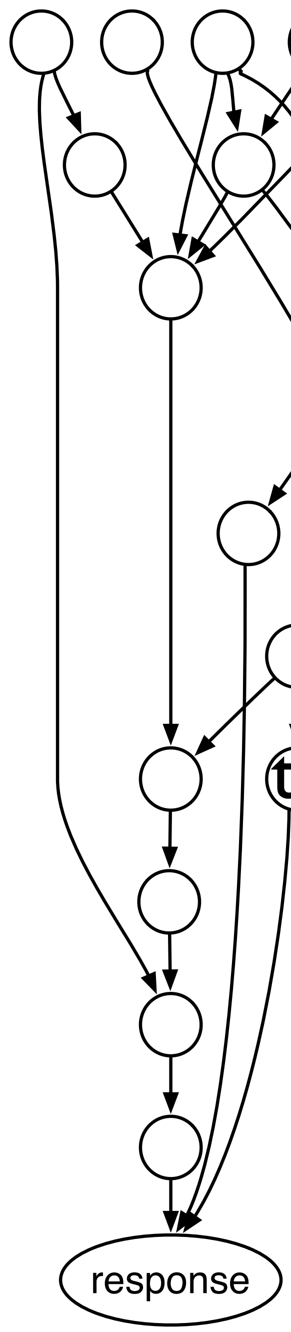
After: feed builder

- Simpler, cleaner code
- Polymorphism is trivial

```
(def topic-feed
  (compile-feed-fn
    {:alpha 0.2}
    {:x (fnk ...)
     :q (fnk ...)}))

(def home-feed
  (compile-feed-fn
    {:alpha 0.4}
    {:x (fnk ...)
     :r (fnk ...)
     :s (fnk ...)}))
```

After: feed builder

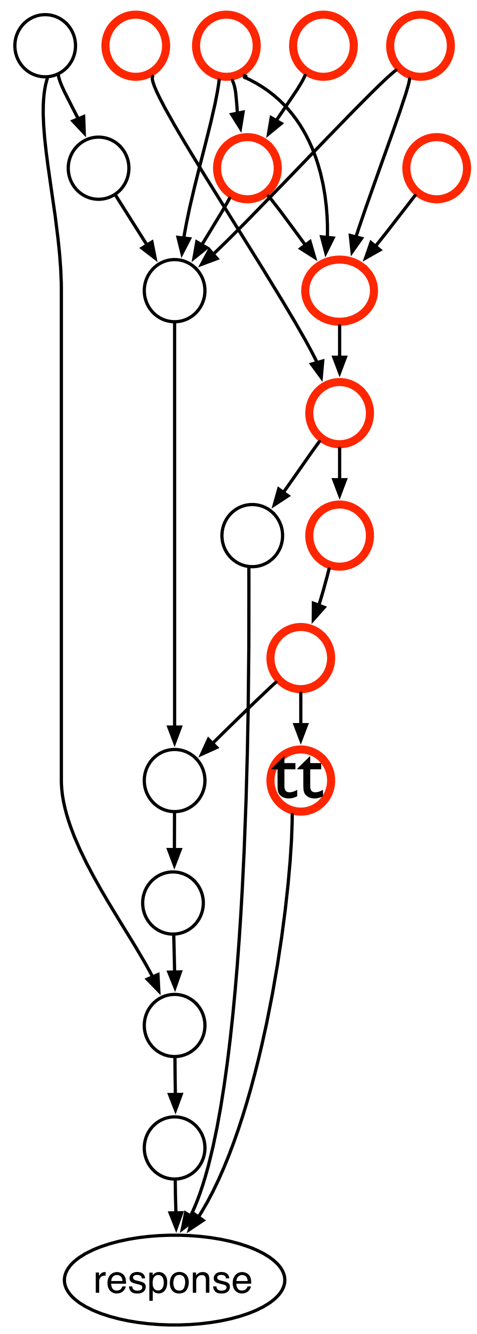
- Simpler, cleaner code
- Polymorphism is trivial
- Early stopping for free via lazy compilation



After: feed builder

- Simpler, cleaner code
- Polymorphism is trivial
- Early stopping for free via lazy compilation

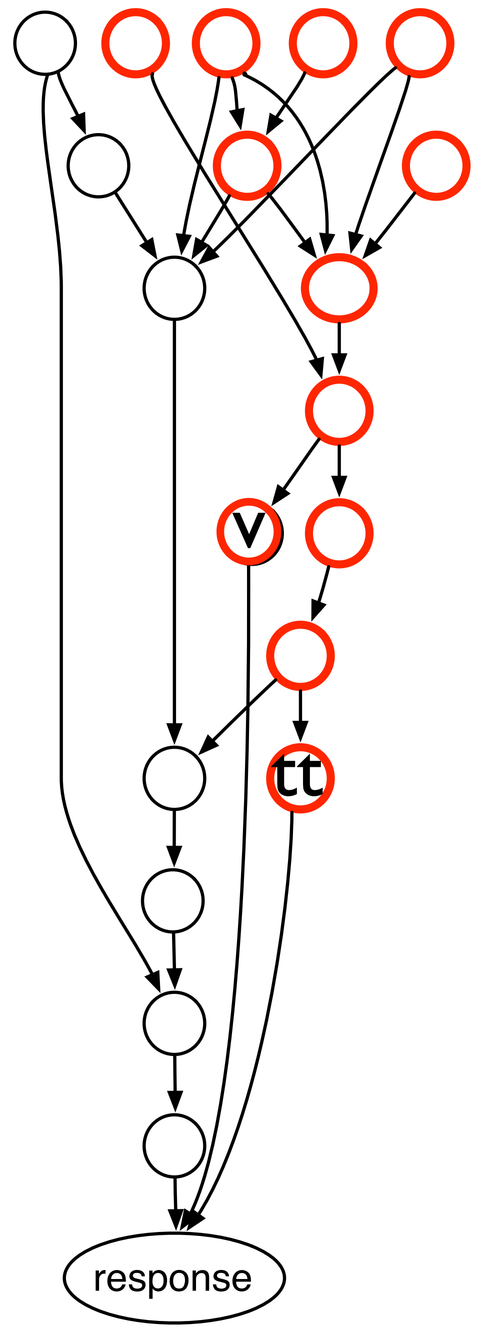
```
(let [h (home-feed req)]
  (:tt h))
```



After: feed builder

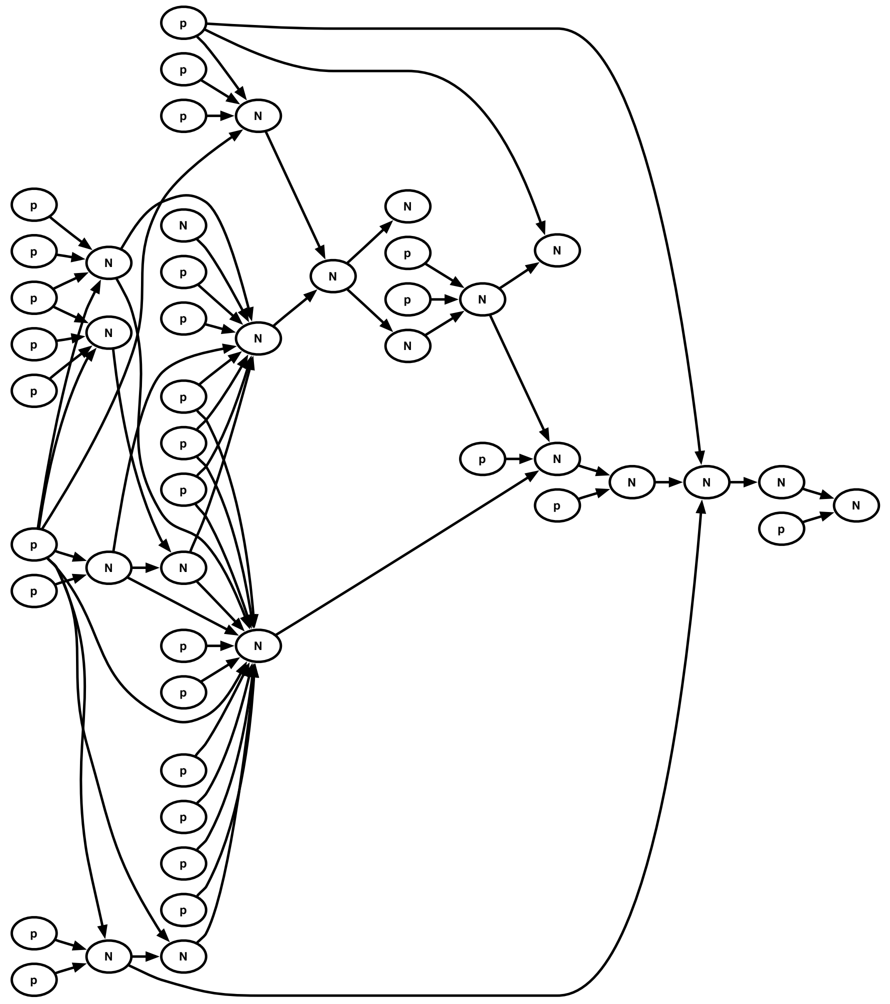
- Simpler, cleaner code
- Polymorphism is trivial
- Early stopping for free via lazy compilation

```
(let [h (home-feed req)]
  [(:tt h)
   (:v h)])
```



Also: easy to analyze

- Detect mis-wirings at graph compile time
  - positional constructor
- Avoid wrong # of args errors, arg ordering bugs
- Visualize graphs in 5 loc



```
(defn edges [graph]
  (for [[k f] graph
        :let [{:keys [req-ks opt-ks]} (meta f)]
        parent (concat req-ks opt-ks)]
    [parent k]))
```

Also: easy to monitor

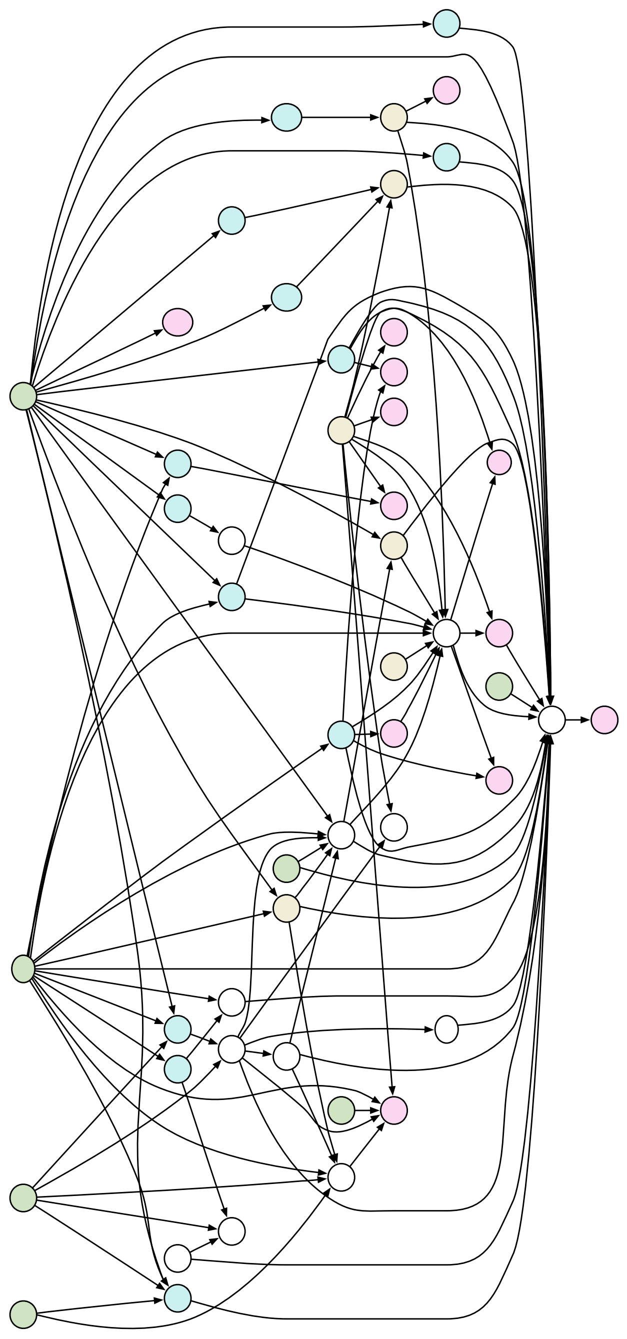
- Add monitoring and error reporting by mapping over fnks

- Since a Graph is a Map, can just use map-vals

node	n	avg ms	errors
:fetch	2500	1.5	0
:rank	1001	150.0	1
:client	1000	70.0	0

```
(defn observe-graph [g]
  (into {}
   (for [[k f] g]
    [k
     (with-meta
       (fn [m]
         (let [v (f m)]
           (print k m v)
           v))
       (meta f))])))
```

Example 2: production API service



```
(def api-service
  (service
    {:service-name "api"
     :backend-port 42424
     :server-threads 100}
    {:store1 (instance store
               {:type :s3 ...})
     :memo   (fnk [store1]
               {:resource ...})
     ...
     :api-server (...)}))
```

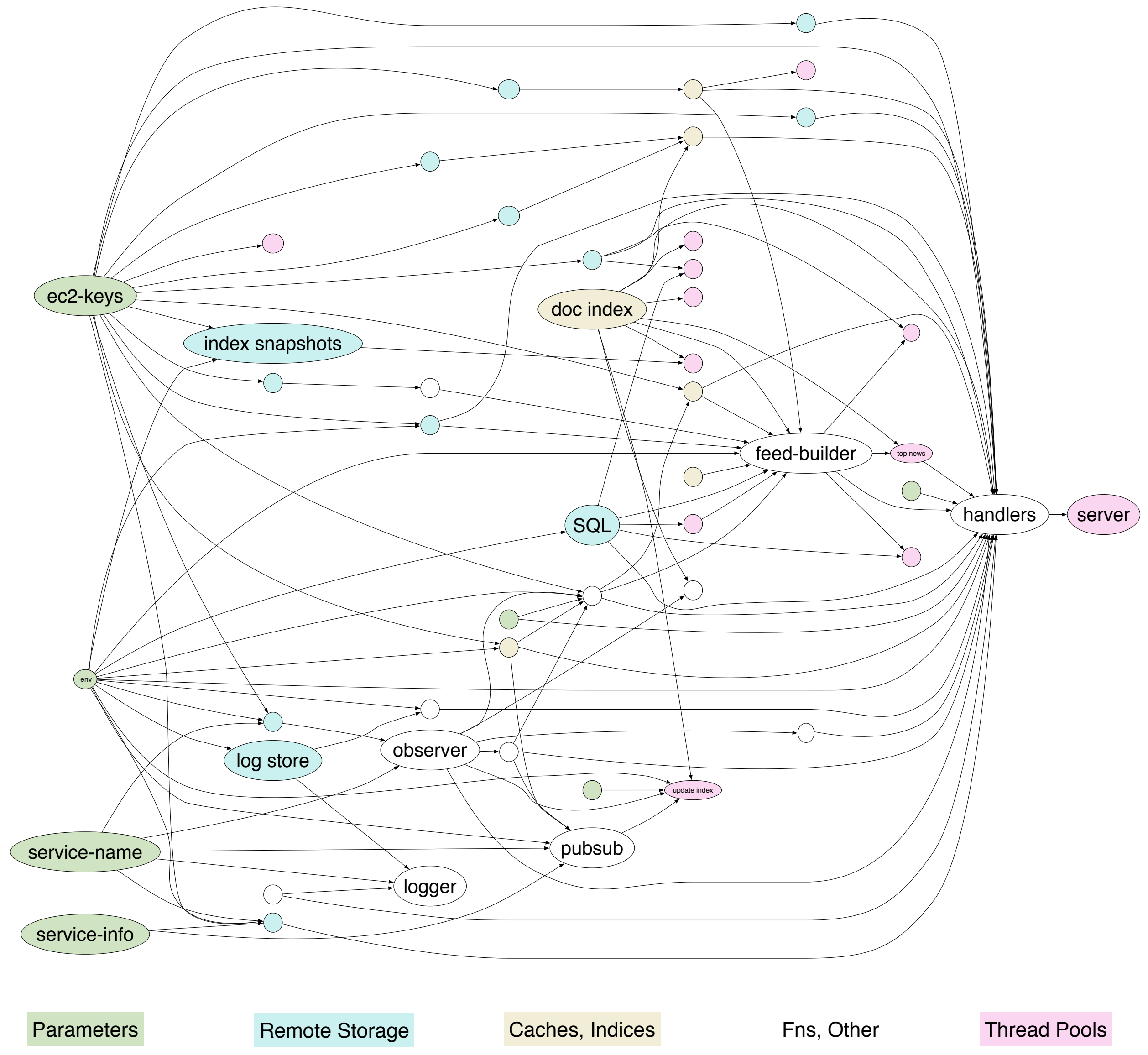
Service definitions

- Service definition =
  - parameter map +
  - resource graph
- Crane reads params for provisioning, deployment
- Graph = service code
  - parameters are args
  - cron jobs, handlers at leaves

```
(def api-service
  (service
    {:service-name “api”
     :backend-port 42424
     :server-threads 100}
    {:store1 (instance store
                {:type :s3 ...})
     :memo   (fnk [store1]
                {:resource ...})
     ...
     :api-server (...)}))
```

Service definitions

- Service definition =
  - parameter map +
  - resource graph
- Crane reads params for provisioning, deployment
- Graph = service code
  - parameters are args
  - cron jobs, handlers at leaves



Service built-ins

- Parameters and graph nodes available by convention
- Interface with deployment, other services, dashboard
- Smartly reconfigure with env -- test/staging/prod

parameters

```
{:env          :prod
 :instance-id "i-123abc"
 :ec2-keys     ...       }
```

resources

```
{:nameserver   ...
 :observer     ...
 :pubsub       ...       }
```

Nodes build Resources

- Resource = component
  - e.g., database, cache, fn
  - Plus metadata for shutdown, handlers, ...
  - Represent as a map
- Library of resources that work with builtins
  - data stores
  - processing queues
  - recurring tasks
  - ...

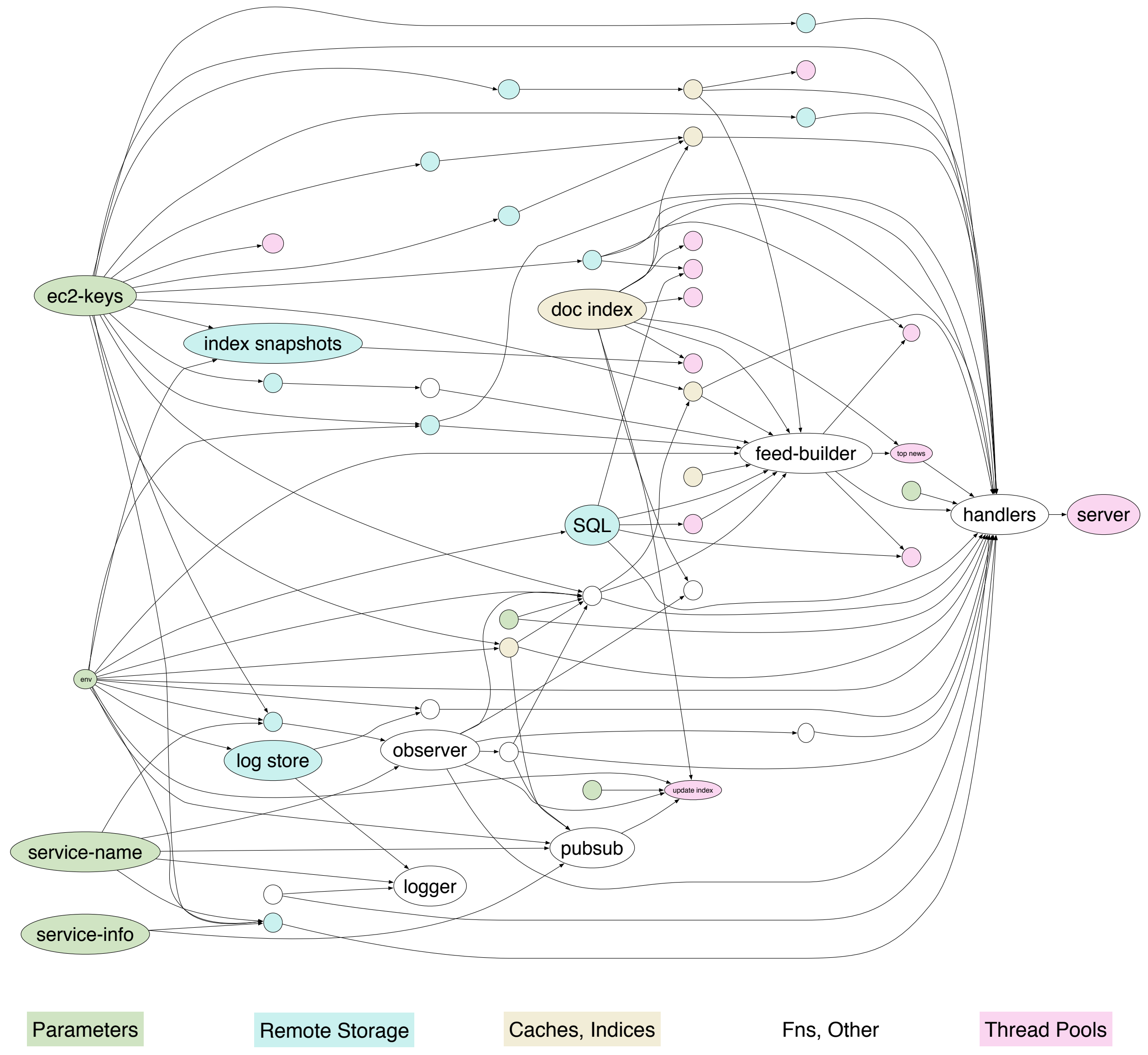
```
(defnk refreshing-atom
 [f period]
 (let [a (atom (f))
       e (Exec/newExec)]
   (.schedAtFixedRate e
    #(reset! a (f))
    period)
   {:res a
    :shutdown #(.sd e)}))
```

Starting and Stopping

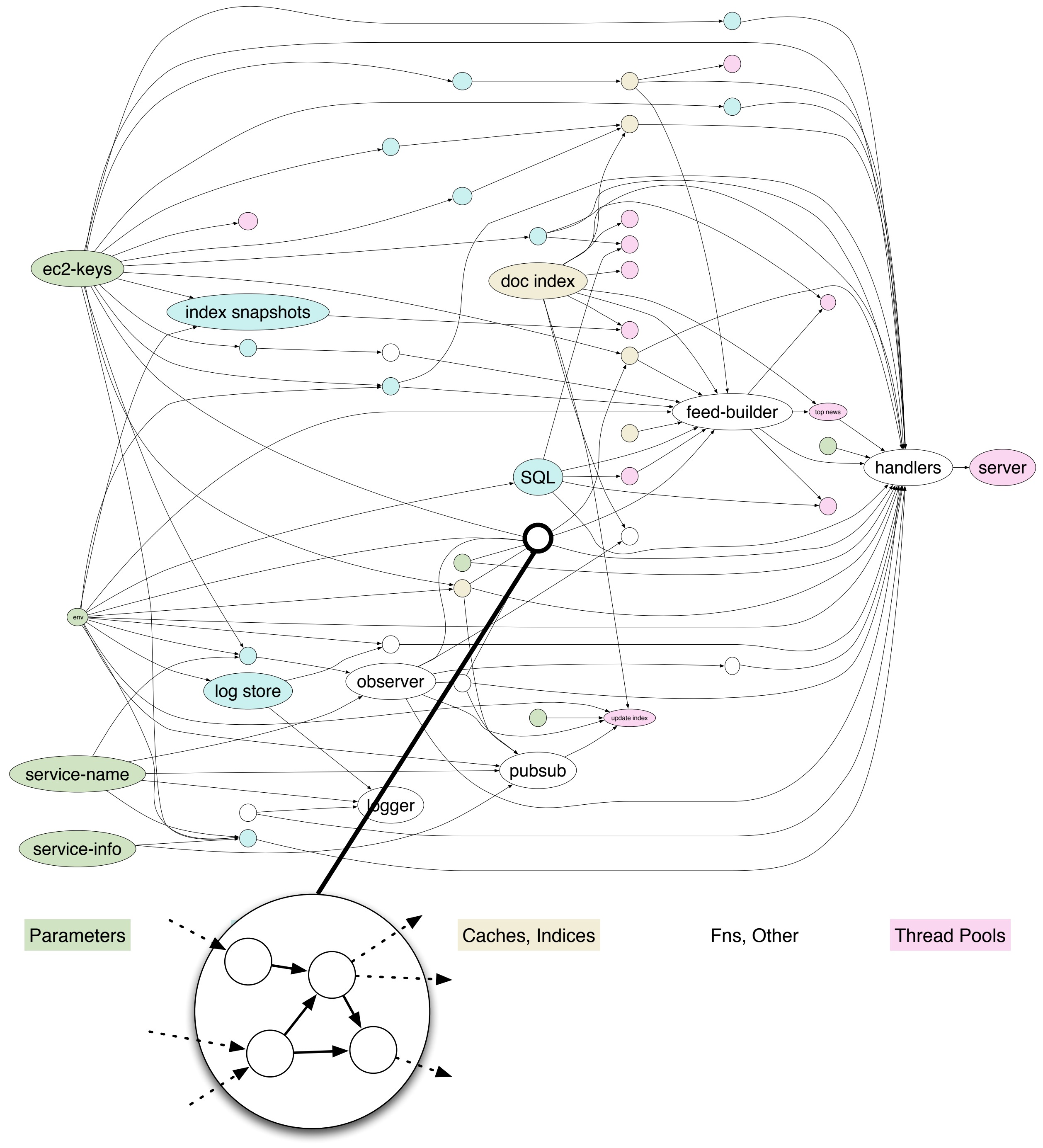
- Transform resource graph to ordinary graph
  - map over leaves, pull out :resource
  - assoc new :shutdown key
- Run graph to start service, get clean shutdown hook

```
(defn start-service [spec]
 ((->> (:graph spec)
       resource-transform
       compile)
  (:parameters spec)))

(def api
  (start-service
    api-service))

((:shutdown api))
```

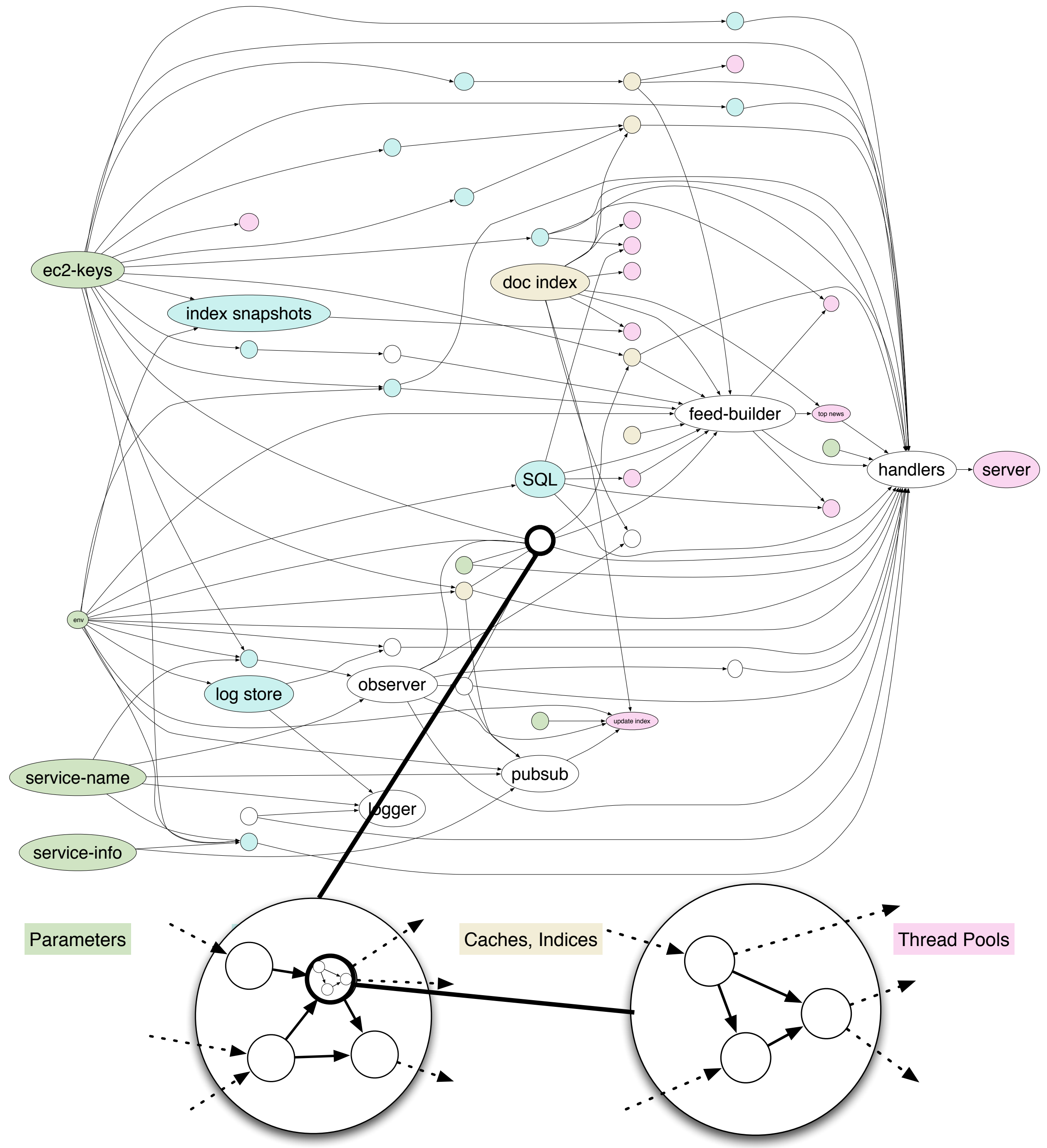
Sub-Components



Sub-Components



Sub-Components



Sub-Components

- Nodes can themselves be Graphs
  - just nested maps
- Package components as sub-graphs
- Sub-graphs are transparent
  - debugging
  - monitoring
  - imperfect abstractions

```
(def write-back-cache
 {:store
    (instance store ...)
  :write-queue
    (instance queue ...)
  :periodic-prune
    (instance task ...)})
```

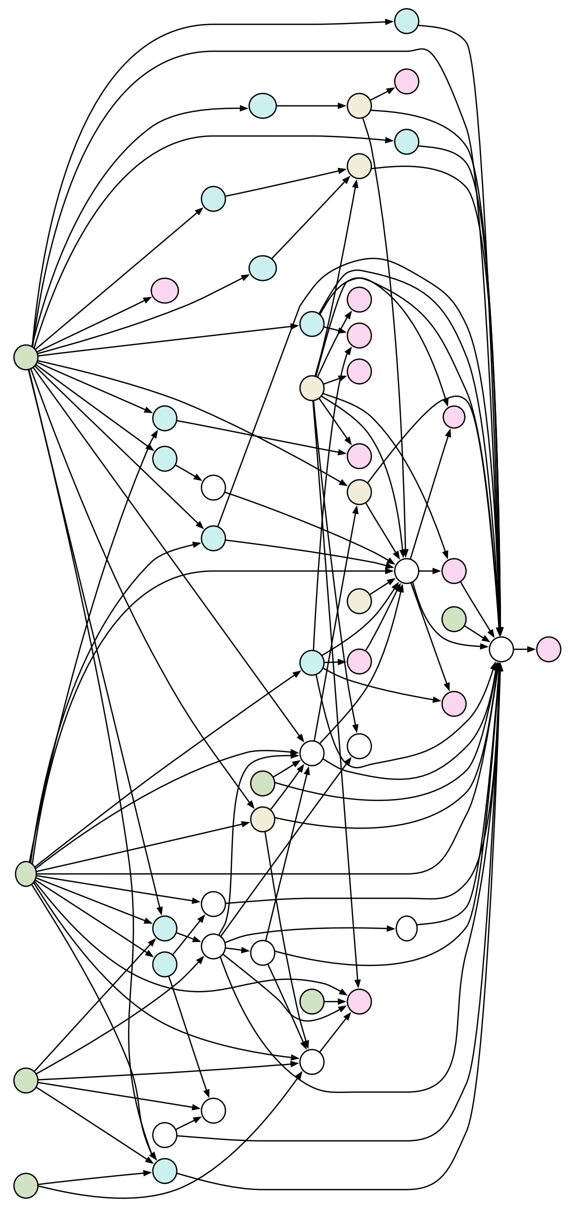
Easy system testing

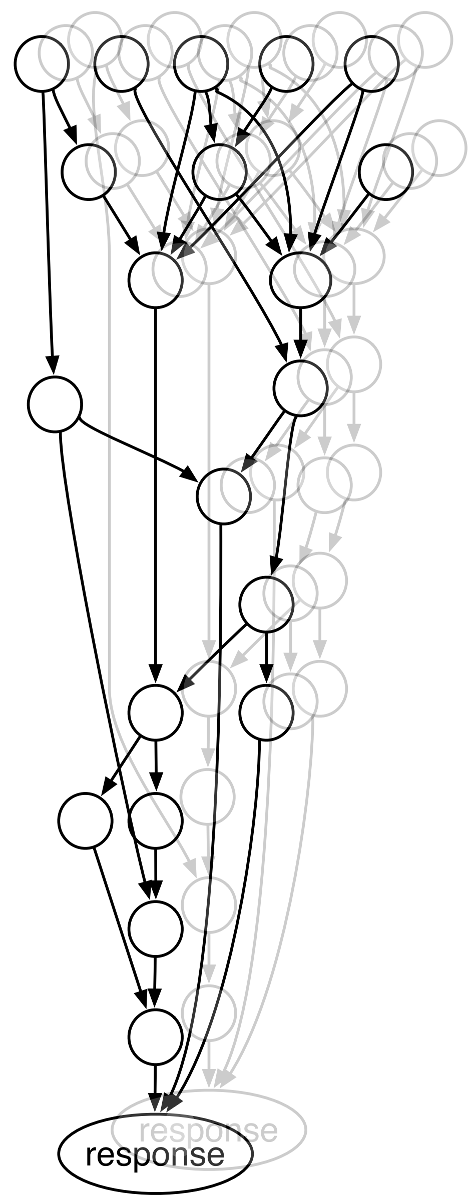
- Old xxx-line `lets` were impossible to test
- With graph, just merge in mock node fnks
  - no elaborate mocks objects or redefs
  - automatic, safe shutdown

```
(deftest home-feed-systest
 (test-service
   (assoc api-service
     :doc-index
       (fnk [] {:res fake-idx})
     :get-user
       (fnk []
        {:res (constantly me)}))
   (is (= (titles (slurp url))
          ["doc1" "doc2"]))))
```

Summary

- Graph = way express complex compositions
  - declaratively
  - simply
- Widely applicable
- Simpler code, better tooling
- Hope to open source soon
  - (we're hiring!)




response
response