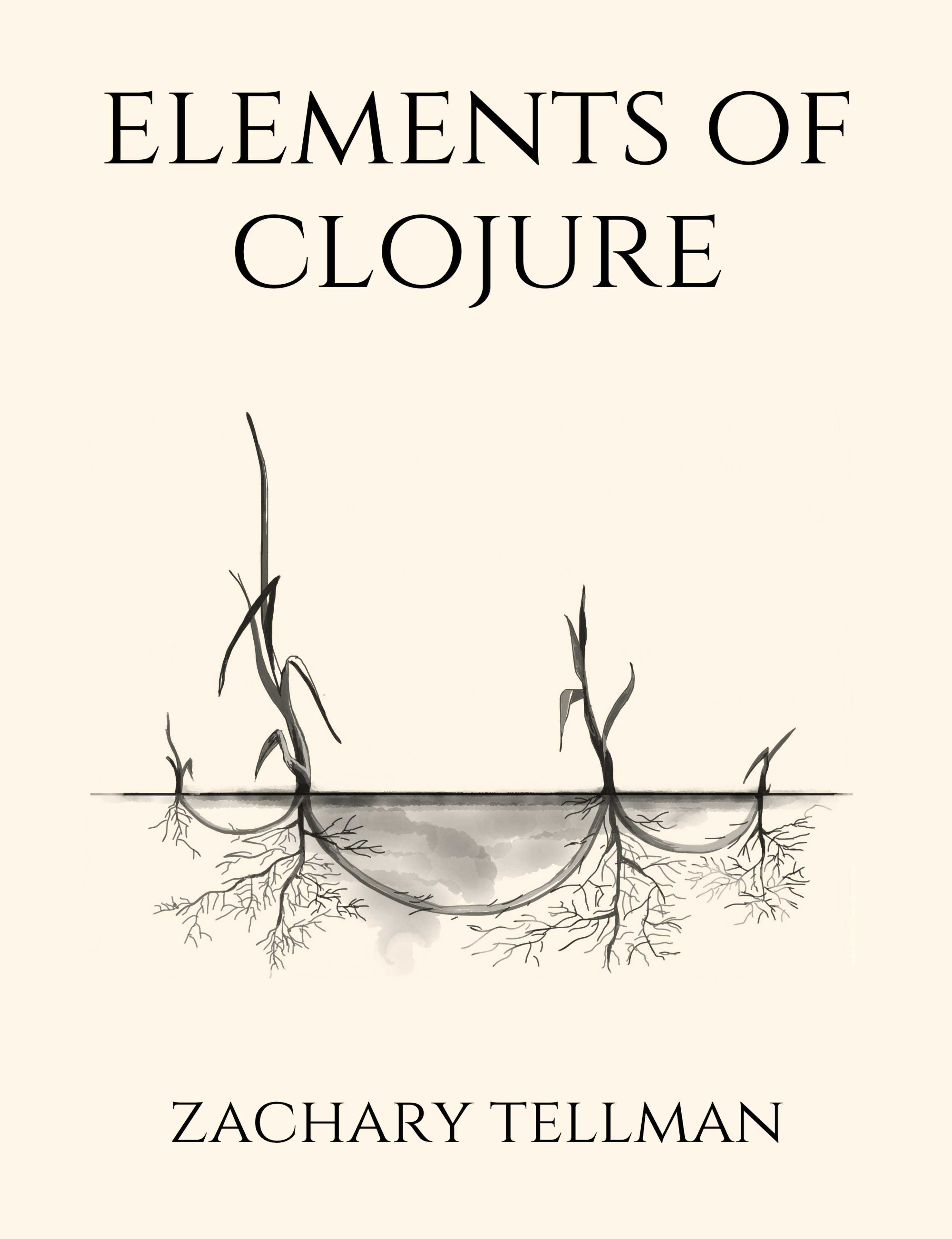
On Composition

Zach Tellman

@ztellman

I've been writing a book

It tries to put words to what most experienced engineers already know

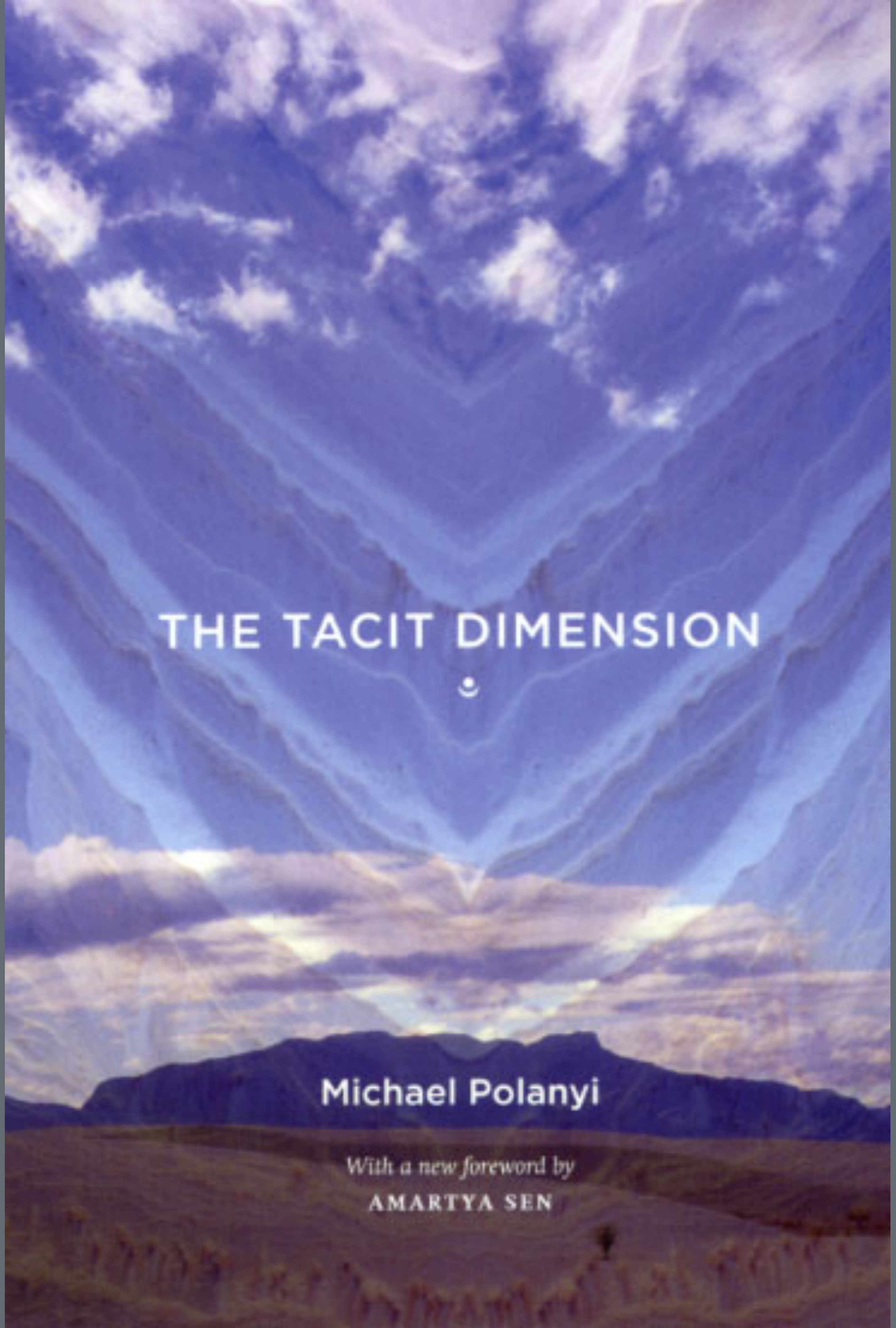


ELEMENTS OF CLOJURE

ZACHARY TELLMAN

"We can know more than we can say"

Tacit knowledge only suffices until we fail



We invent **names**, but do not study naming

We create **abstractions**, but struggle to even define the word

I've written about abstractions

To abstract is to treat things which are different as equivalent

An abstraction must be judged within a **context**

We can't say much about it in isolation

By placing abstractions together, we begin to define their context

Composition is applied abstraction

I. The Goal of Composition

We're not here to talk about `(comp f g)`

Functions are a means, not an end unto themselves

We are trying to construct a **process**

We must **pull** in data, **transform** it, and **push** the result elsewhere

Properties of a process:

- Sequential actions
- Execution isolation (**when** it runs)
- Data isolation (**where** it runs)

Some examples of a process:

- A thread
- A linear chain of callbacks
- Early UNIX processes
- Carl Hewitt's actors
- Erlang's processes
- Smalltalk-72's objects

A process is the smallest standalone unit of computation

It has to pull, transform, **and** push

□ pull

□ transform

☑ push

```
(defn yes []
  (loop []
    (println "y")
    (recur)))
```

☑ pull

□ transform

□ push

```
(defn dev-null []
  (loop []
    (read-line)
    (recur)))
```

☑ pull

□ transform

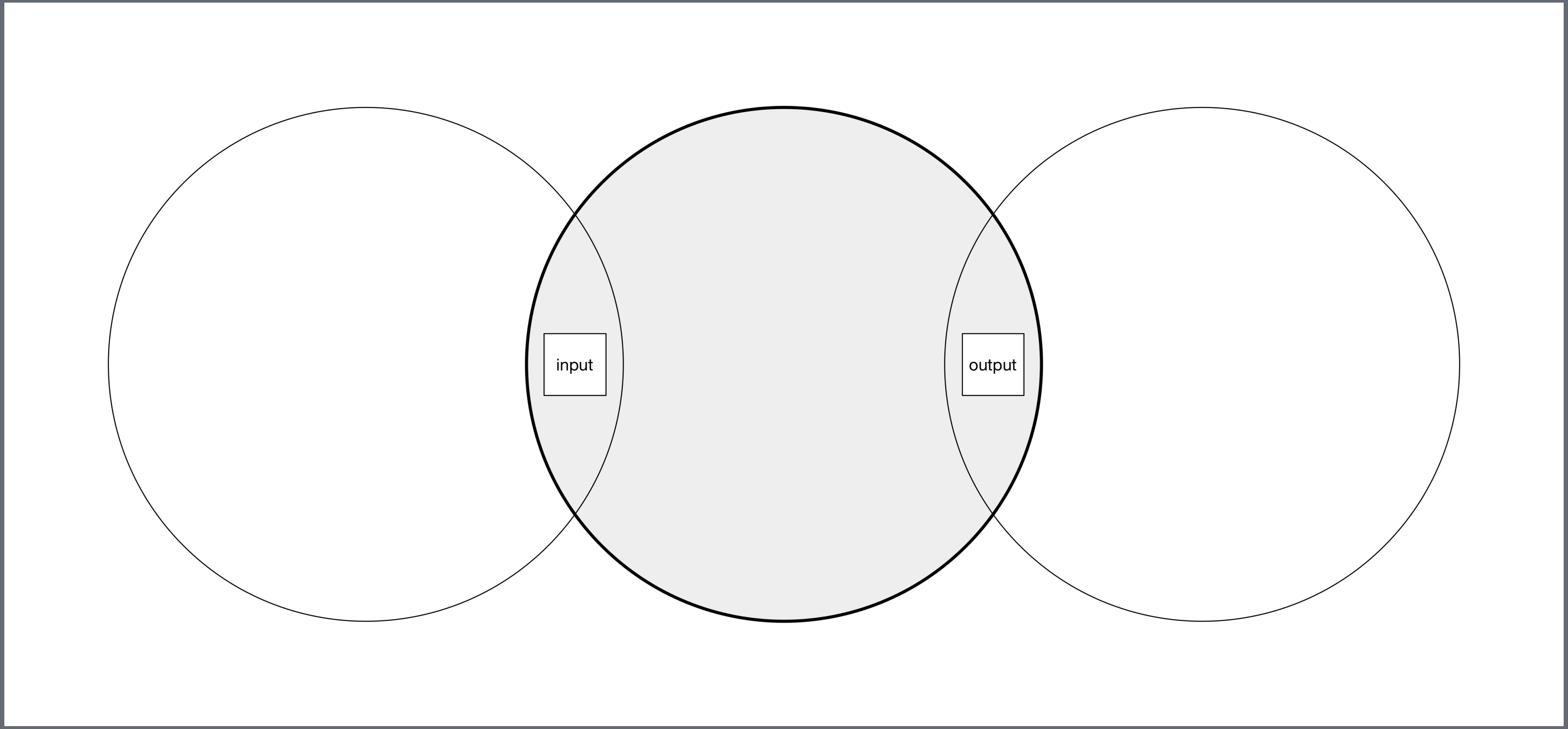
☑ push

```
(defn cat [& filenames]
  (doseq [f filenames]
    (doseq [l (->> f io/reader line-seq)]
      (println l))))
```

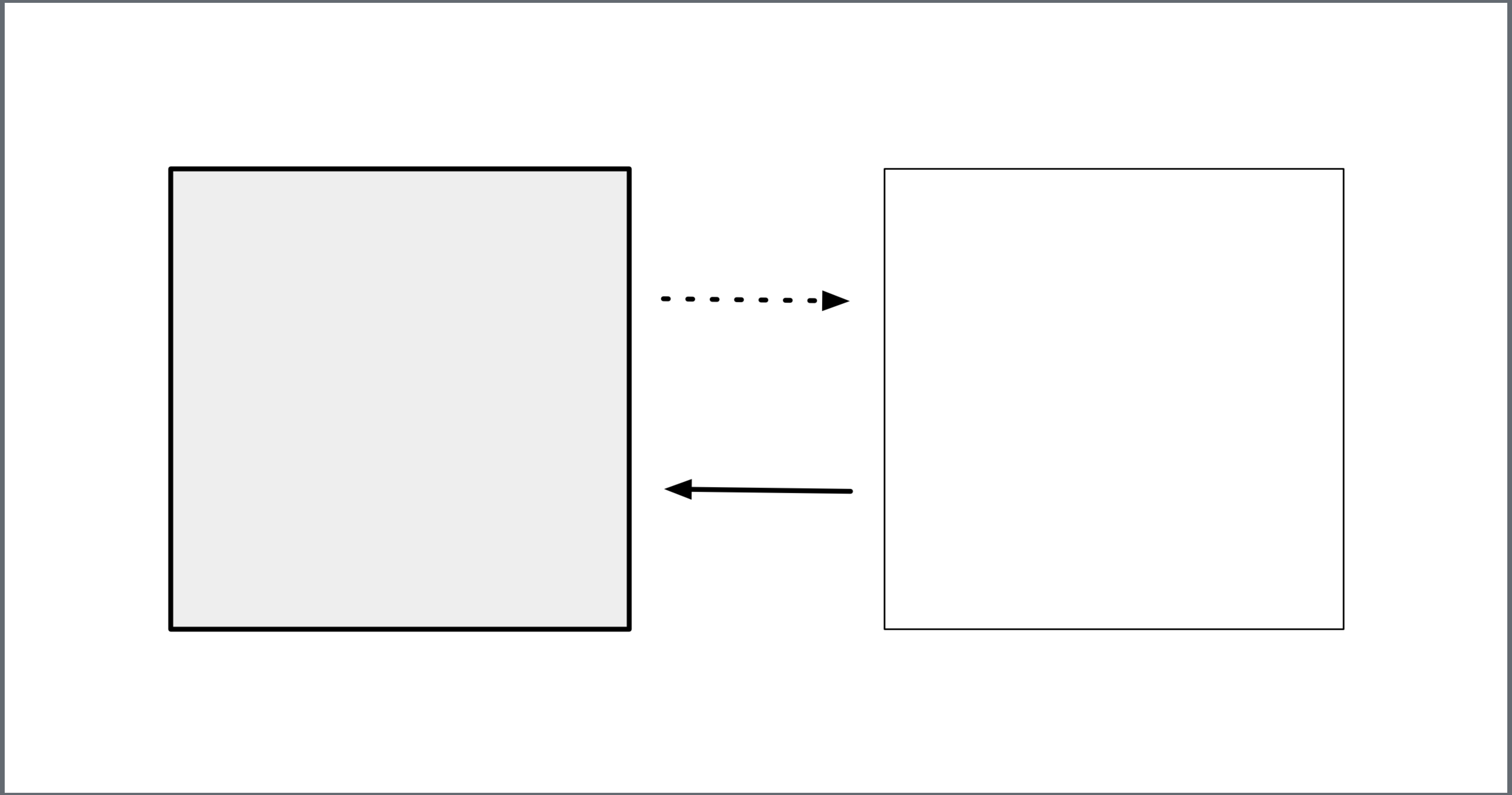
- □ pull
- ☑ transform
- ☑ push

```
(defn yes
  ([]
    (yes "y"))
  ([expletive]
    (loop []
      (println expletive)
      (recur))))
```

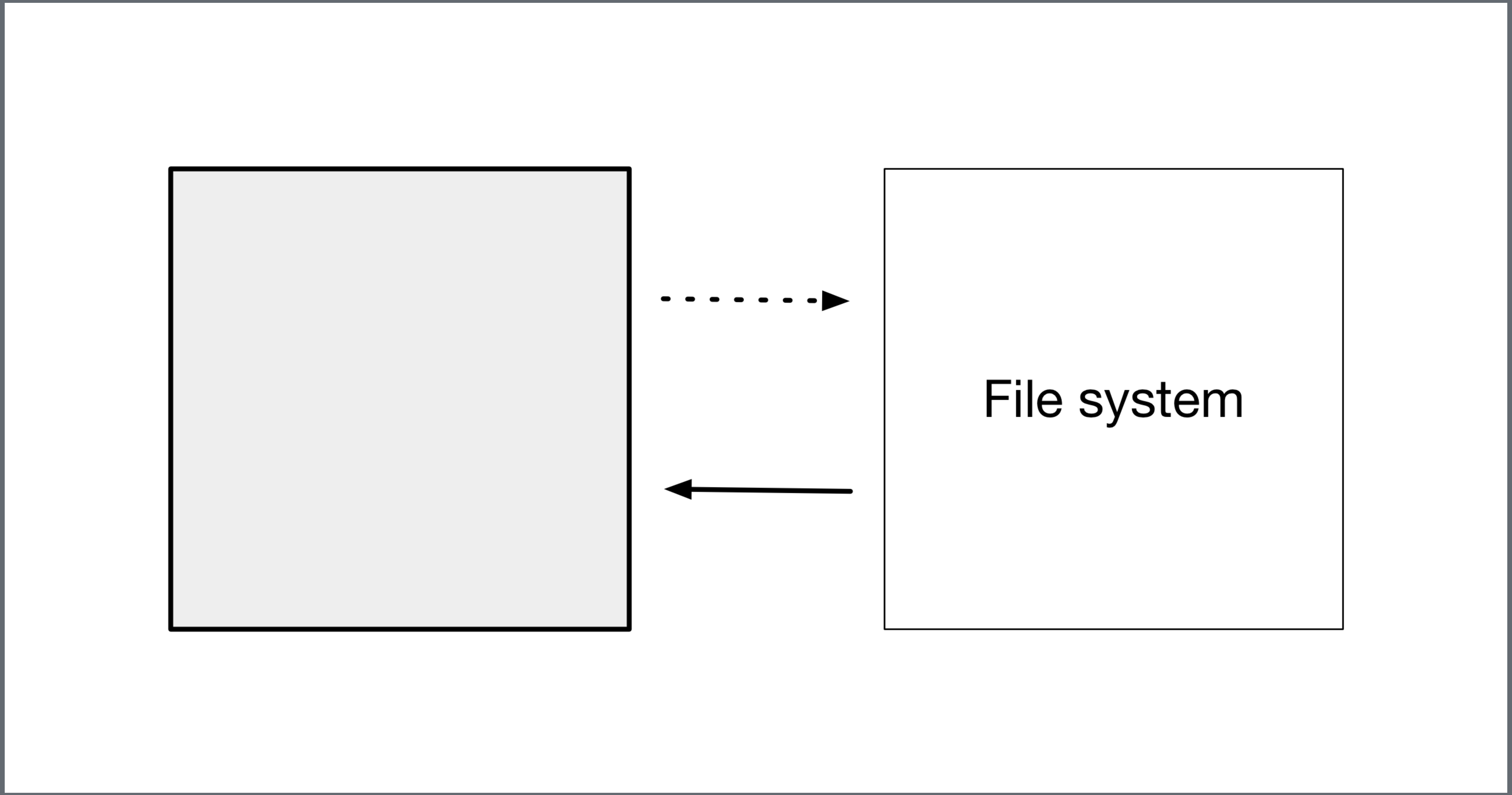
Processes provide (some) data isolation



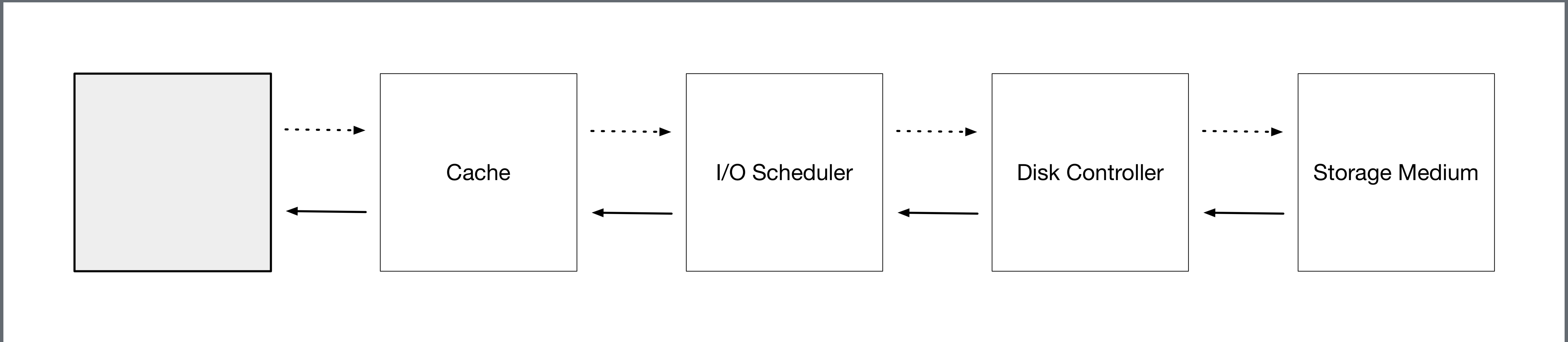
Processes provide (some) execution isolation



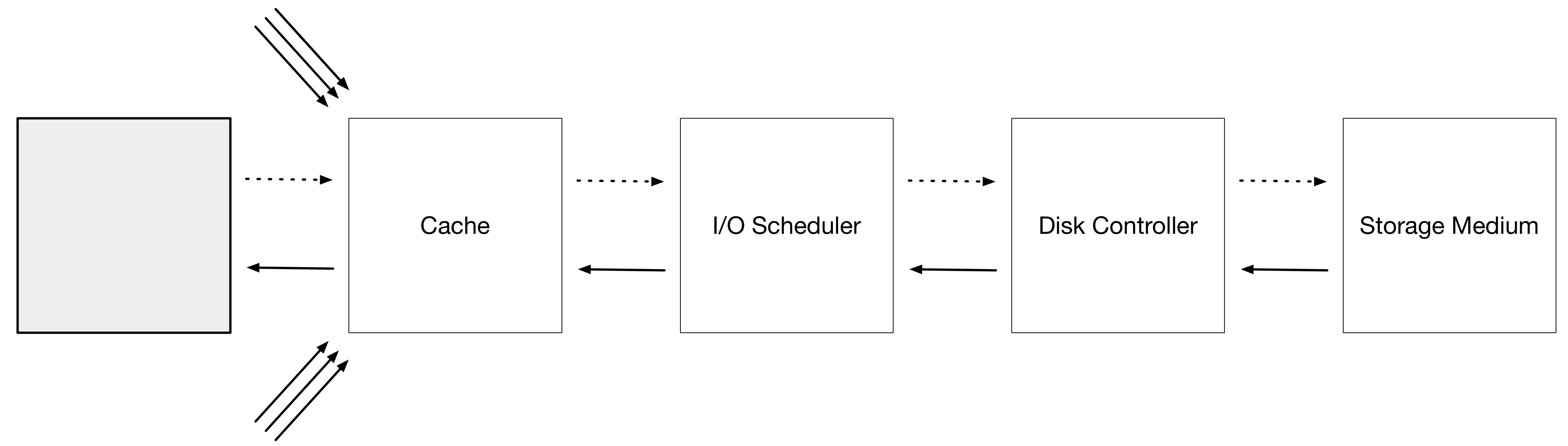
How do we read from a file?



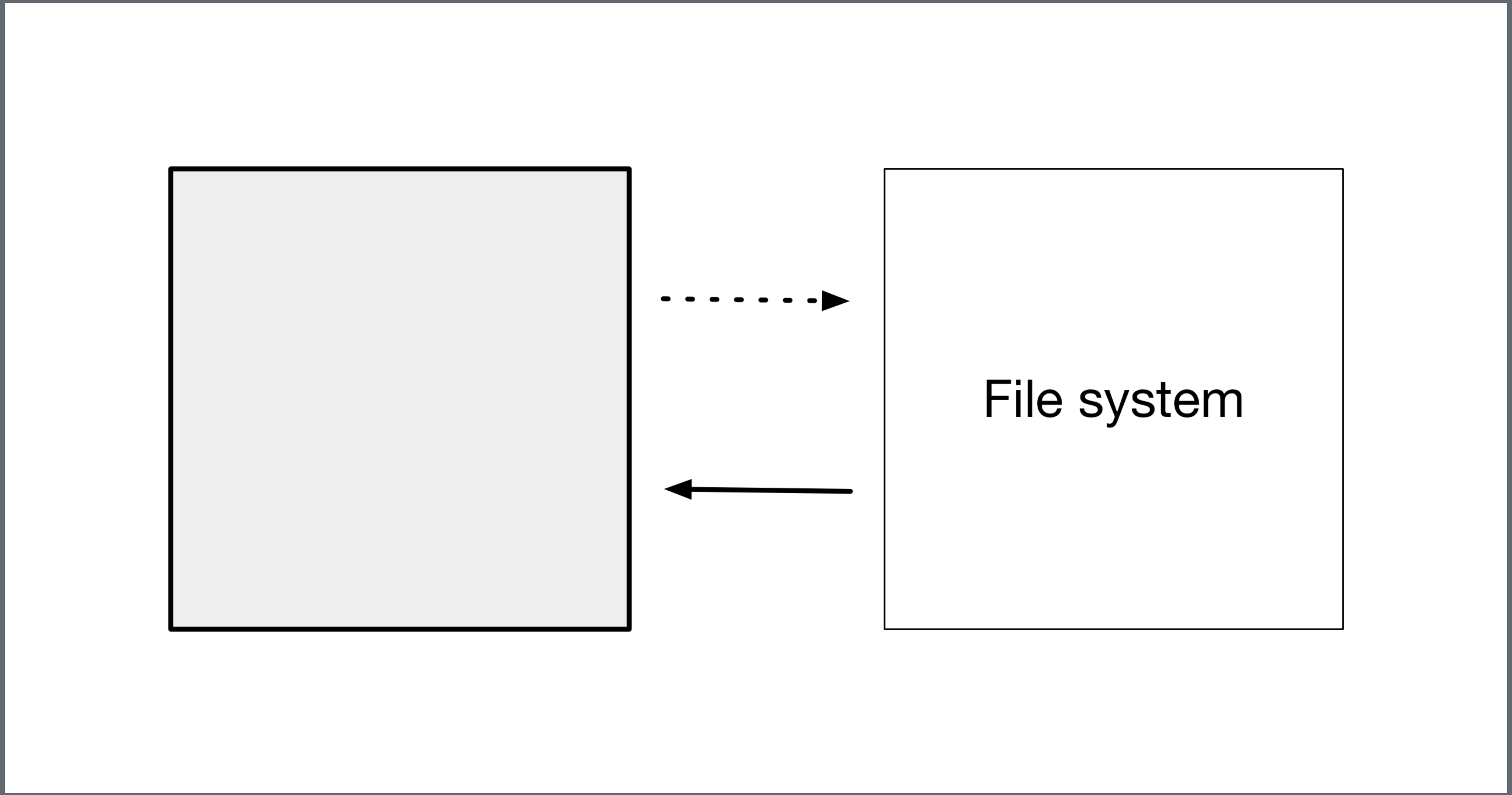
How do we read from a file?



How do we read from a file?



Ignorance is bliss



Have low expectations

Plan for disappointment

Your spec may call for a system
which is fast and reliable

Let's hope the system
fits in your head

II. Real-life Examples

Example: a REPL

```
(defn repl []
  (loop []
    (->> (read)
      eval
      print)
    (recur)))
```

Pulls too slowly:

```
(print (eval '(reduce + (range 1e9))))
```

Pushes too quickly:

```
(print (eval '(range 1e9)))
```

Every push has a little pull, and vice-versa.

Example: a web service

```
(defn handler [request]
  (->> request
    request->query
    query-db!
    result->response))
```

- **pull** in an encoded request from the client

- **pull** in an encoded request from the client
- **transform** the encoded request into a Ring request

- **pull** in an encoded request from the client
- **transform** the encoded request into a Ring request
- **transform** the Ring request into a database query

- **pull** in an encoded request from the client
- **transform** the encoded request into a Ring request
- **transform** the Ring request into a database query
- **pull** the result of that query from the database

- **pull** in an encoded request from the client
- **transform** the encoded request into a Ring request
- **transform** the Ring request into a database query
- **pull** the result of that query from the database
- **transform** the database result into a Ring response

- **pull** in an encoded request from the client
- **transform** the encoded request into a Ring request
- **transform** the Ring request into a database query
- **pull** the result of that query from the database
- **transform** the database result into a Ring response
- **transform** the Ring response into an encoded response

- **pull** in an encoded request from the client
- **transform** the encoded request into a Ring request
- **transform** the Ring request into a database query
- **pull** the result of that query from the database
- **transform** the database result into a Ring response
- **transform** the Ring response into an encoded response
- **push** the encoded response to the client

Sometime a framework defines the edges of our process

Example: a frontend application

```
(on-click refresh-button
  (fn []
    (query-service
      (fn [data]
        (update-dom data)))))
```

Example: a frontend application

```
(def refreshing? (atom false))

(on-click refresh-button
  (fn []
    (when-not @refreshing?
      (reset! refreshing? true)
      (query-service
        (fn [data]
          (update-dom data)
          (reset! refreshing? false))))))
```

Example: a frontend application

```
(on-click refresh-button
  (fn []
    (disable! refresh-button)
    (query-service
      (fn [data]
        (update-dom data)
        (enable! refresh-button))))
```

At the edges, we have to deal with neighbors who are **flaky** and **demanding**

Our strategy is called an **execution model**

Queues use backpressure, which **pauses** demanding neighbors

This forces them to share an execution model

"Separation of concerns" is an **operational** property of our software

By themselves, queues do not separate code in motion

Low expectations,
timeouts,
and **explicit failure modes**
separate code in motion

III. How to Construct a Process

Components belong to a single phase

```
(defn repl [read eval print]
  (loop []
    (->> (read)
      eval
      print)
    (recur)))
```

Push and pull phases are **operational**

Transform phases are **functional**

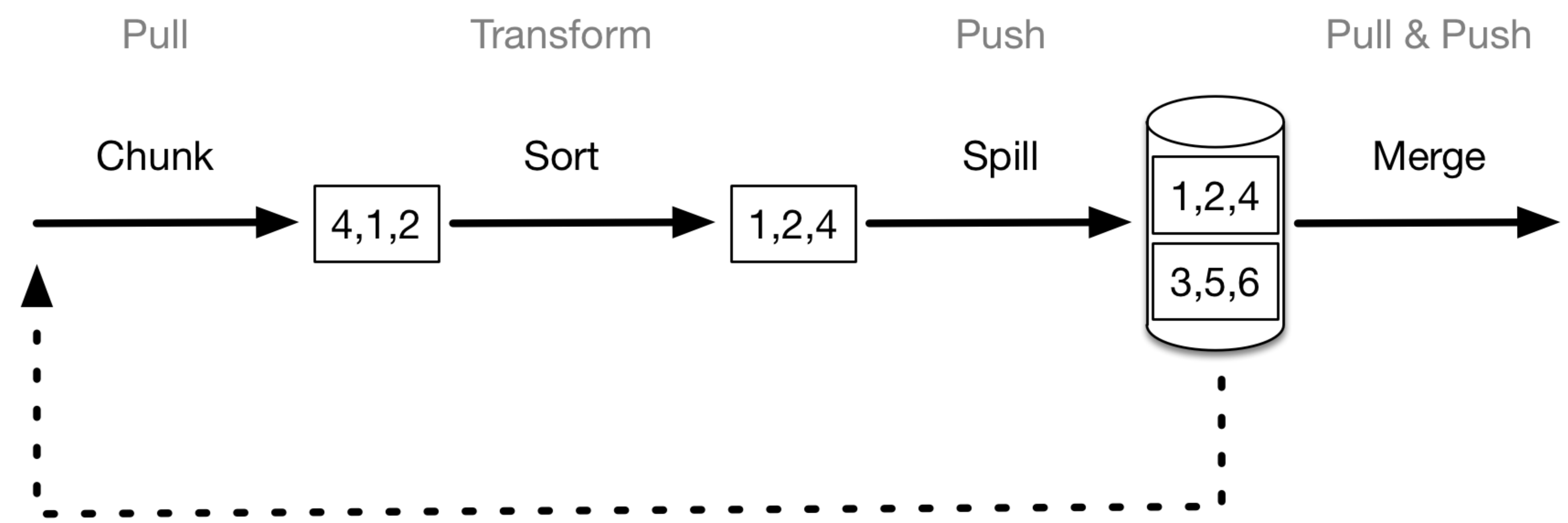
"Returns a sorted sequence of the items in coll."

```
(sort coll)
```

"Throws an `OutOfMemoryException`. You monster."

```
(sort (range 1e10))
```

GNU Sort



Pull
Transform
Push
Pull & Push
Chunk
4,1,2
Sort
1,2,4
Spill
1,2,4
3,5,6
Merge

The pull phase ensures the data is appropriately **sized** and **shaped**

It also defines what happens when data isn't available

The pull phase does not simply yield a `lazy-seq`

The transform phase turns
well-shaped data
into different
well-shaped data

We can **accrete** data

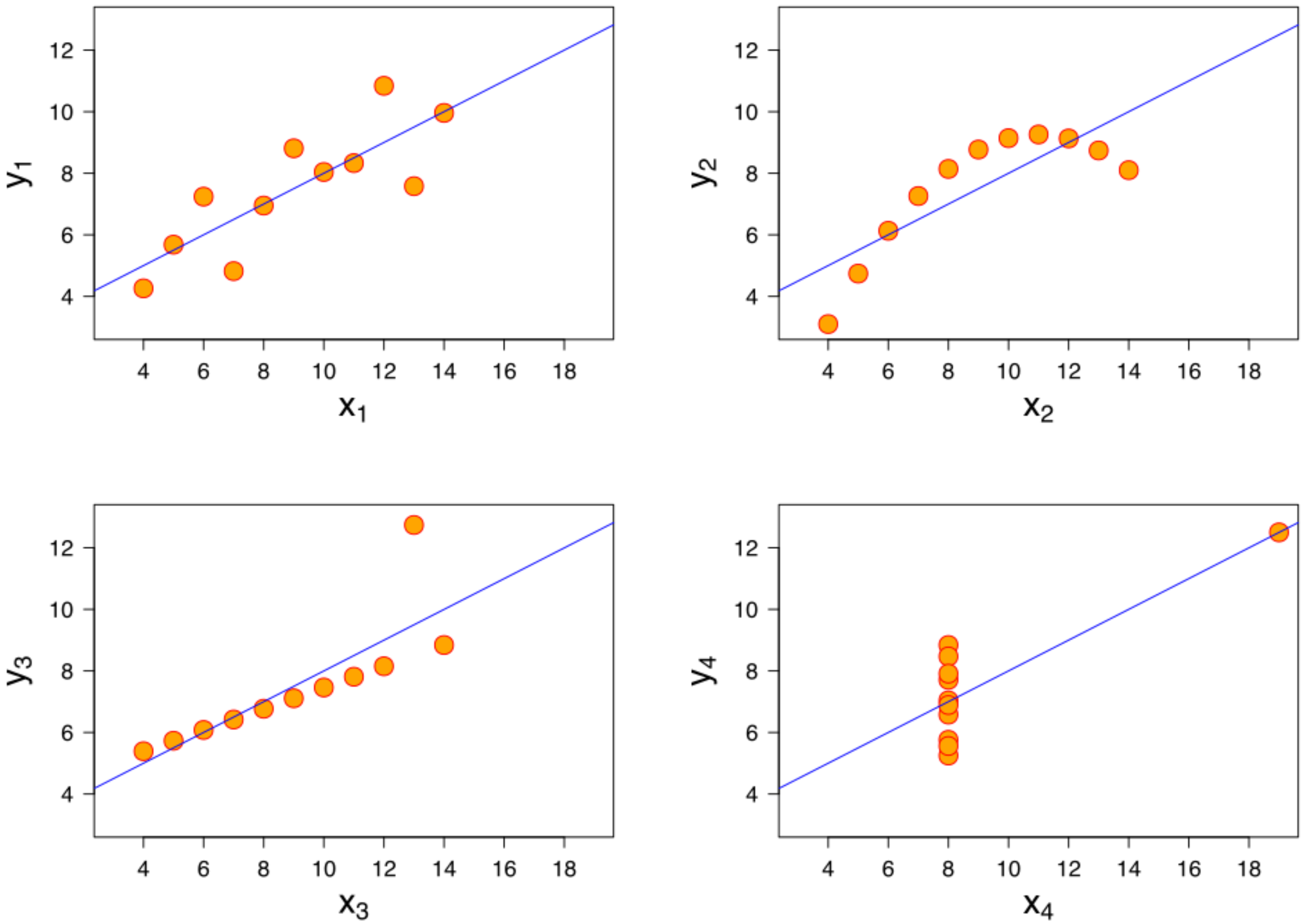
We can **reduce** data

We can **reshape** data

We accrete data when we need to know more

We reduce data when differences don't matter

This is abstraction!



12
10
8
6
4
y_1
4
6
8
10
12
14
16
18
x_1
12
10
8
6
4
y_2
4
6
8
10
12
14
16
18
x_2
12
10
8
6
4
y_3
4
6
8
10
12
14
16
18
x_3
12
10
8
6
4
y_4
4
6
8
10
12
14
16
18
x_4

We reshape data to aid accretion and reduction

This is **not** abstraction!

databases
vs
flat files

&

```
{1 2, 3 4, 5 6}
```

vs

```
[[1 2] [3 4] [5 6]]
```

Reshaping should always be a separate operation

Accrete and reduce should be separate where possible

We transform data into a description of our effects

```
{:url    "http://example.com"
 :method :post
 :body   "how's it going?"}
```

We transform data into a description of our effects

```
{:url               "http://example.com"
 :method            :post
 :body              "how's it going?"
 :follow-redirects? true
 :throw-execptions? true}
```

We transform data into a description of our effects

```
{:url                "http://example.com"
 :method             :post
 :body               "how's it going?"
 :follow-redirects?  true
 :throw-execptions?  true
 :max-redirects      99}
```

The transform phase describes effects, but through a layer of indirection

Data has no inherent semantics

Functions have some semantics, but can only accrete

If a function performs an effect, we've moved outside the transform phase

The transform phase can be tested in isolation

It should be as large as possible

The pull and push phases can only be tested in a reasonable facsimile of production

They should be as small as possible

Lots of processes doesn't mean our system can be understood incrementally

We also need strong invariants at the edge

IV. How to Combine Processes

Processes are not a value

We can pass around an **identifier** or **channel**, but not the process itself

In simple systems, adjacent processes are provided up-front

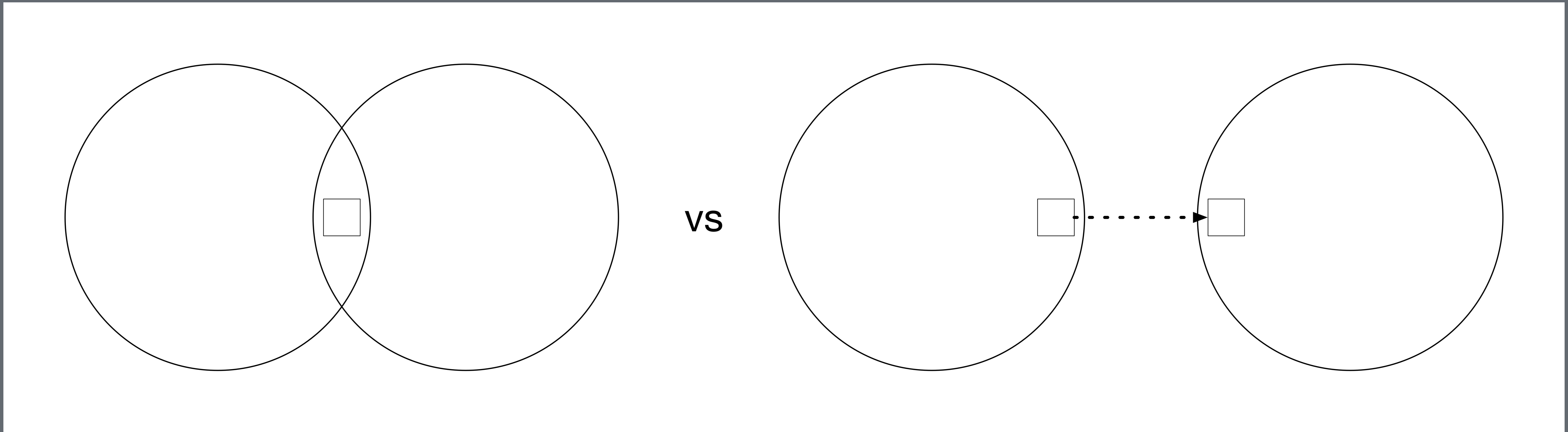
```
cat /tmp/europa | grep 'callisto' > /tmp/ganymede
```

Discovery and **resolution** map abstract identifiers onto more concrete identifiers

Routing exposes a single channel, and distributes the input across many processes

A thread pool is a router, too

Local vs Distributed



We need to acknowledge **actions**, not communication

There's a lot more to say here, obviously

I wish I had a good book to recommend

We compose functions to create **processes**

We compose processes to create **systems**

Questions?