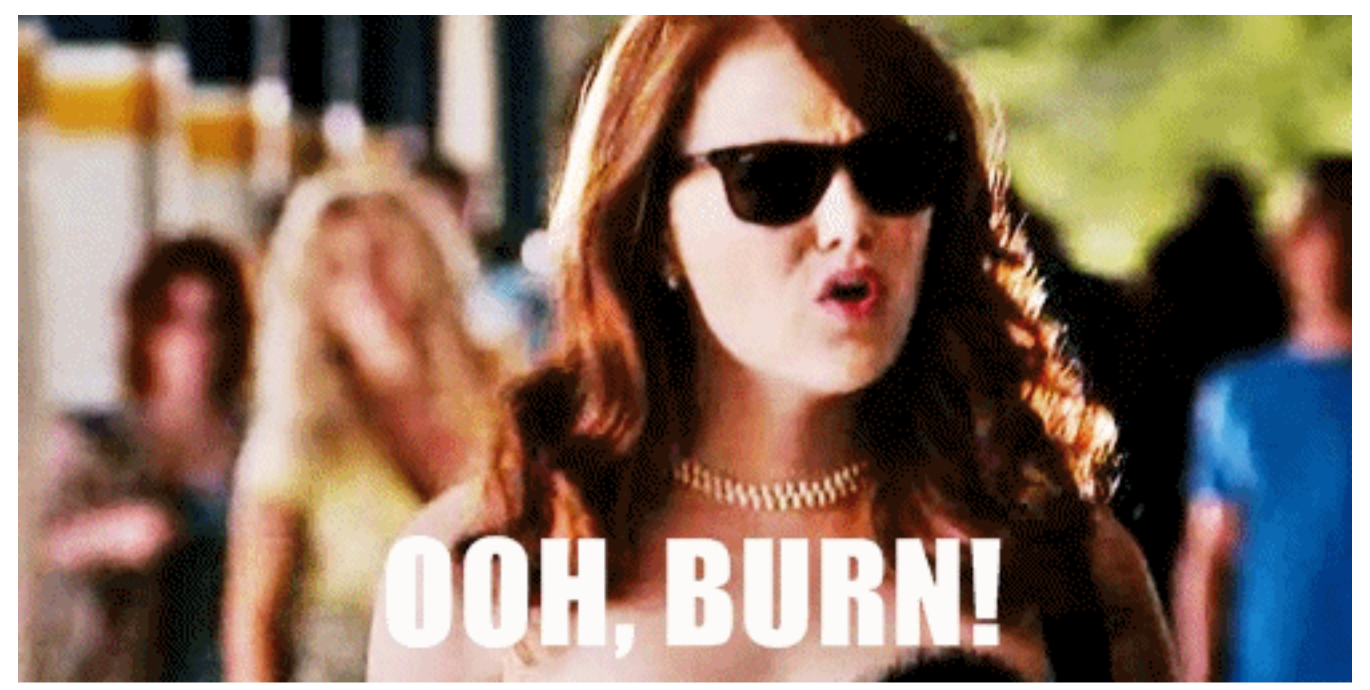
THE TYRANNY OF
STRUCTURELESSNESS
HOW MORE MEANINGFUL CODE CAN MAKE YOUR PROJECT MORE RESILIENT & MAINTAINABLE

For a number of years I have been familiar with the observation that **the quality of programmers is a decreasing function of the density of GOTO statements** in the programs they produce

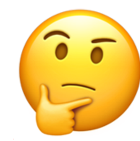
EDSGER DIJKSTRA

For a number of years I have been familiar with the observation that **the quality of programmers is a decreasing function of the density of GOTO statements** in the programs they produce

EDSGER DIJKSTRA



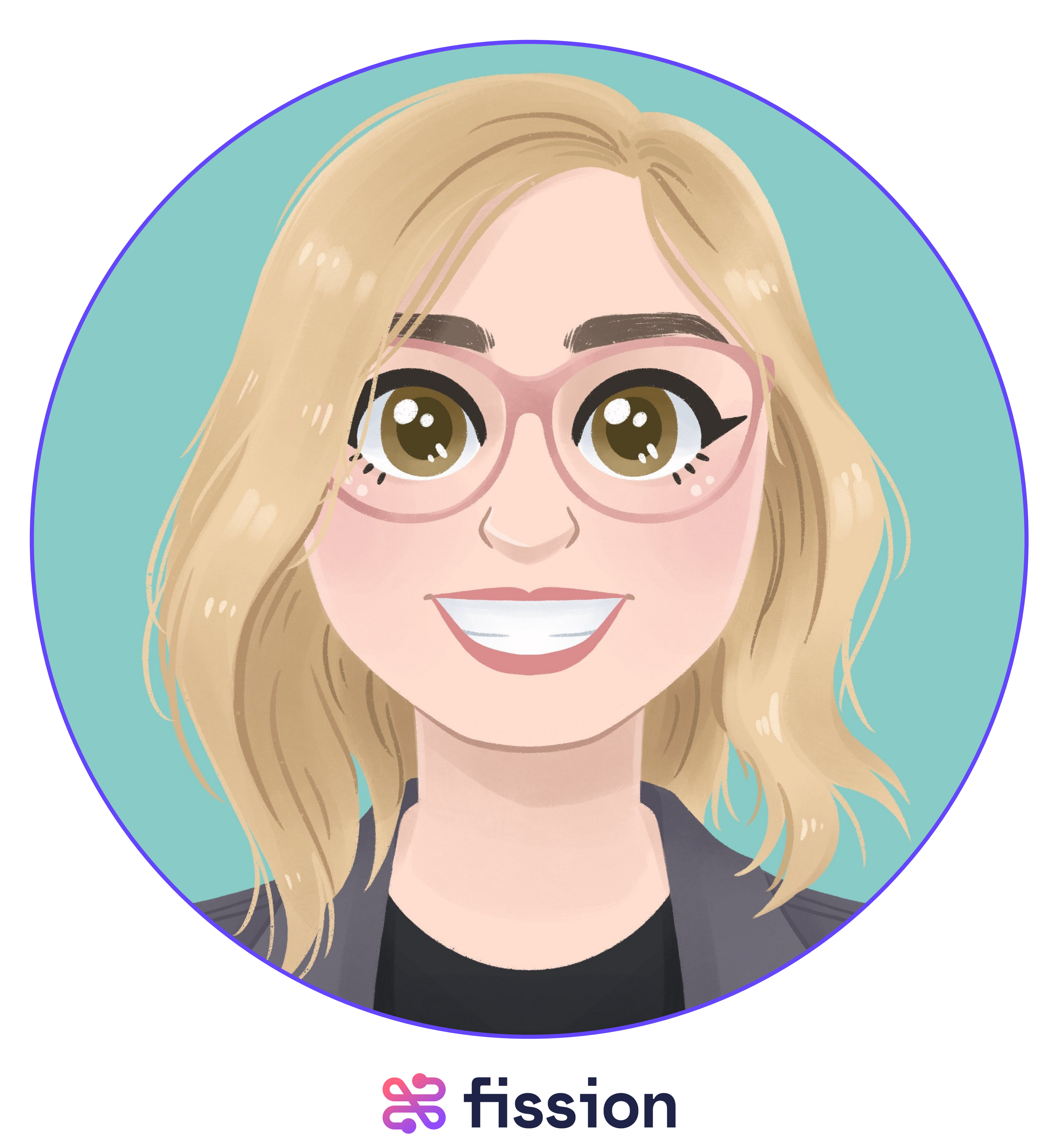
What’s she on about? Elixir doesn’t have GOTOs…



THIS AUDIENCE

THE TYRANNY OF STRUCTURELESSNESS

BROOKLYN ZELENKA, @expede



THE TYRANNY OF STRUCTURELESSNESS

BROOKLYN ZELENKA, @expede

- Cofounder/CTO at Fission
  - `https://fission.codes`
  - Make DevOps & Backend obsolete 😈
  - Spending a *lot* of time with IPFS & DIDs
- PLT & VM enthusiast
- Ethereum Core Developer
- Primary author of Witchcraft Suite & Exceptional

THE TYRANNY OF STRUCTURELESSNESS

BROOKLYN ZELENKA, @expede

- Cofounder/CTO a
  - https://fiss
  - Make DevOps
  - Spending a *lot*
- PLT & VM enthus
- Ethereum Core D
- Primary author of Witchcraft Suite & Exceptional

THE **BIG** IDEA

THE **BIG** IDEA

THE BIG IDEA

ONE-LINER

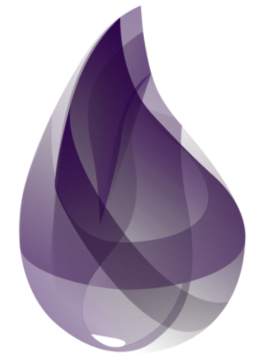
THE BIG IDEA

ONE-LINER

✈️ Work at a higher level 🛸

THE BIG IDEA

LANGUAGE DESIGN REFLECTS INTENDED USE

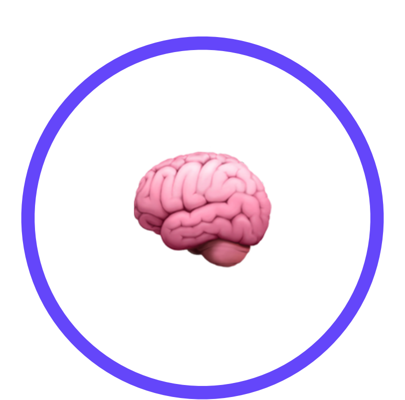


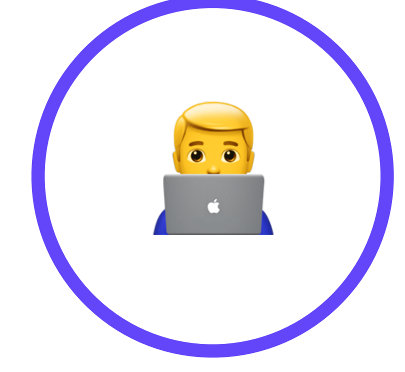
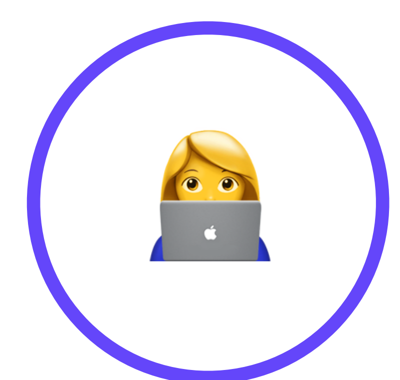
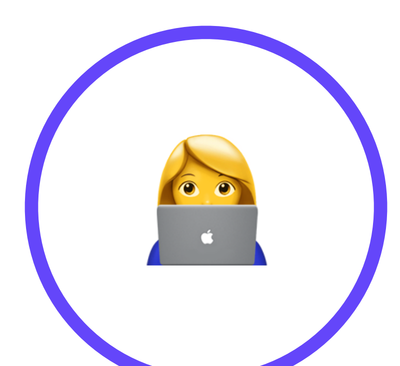
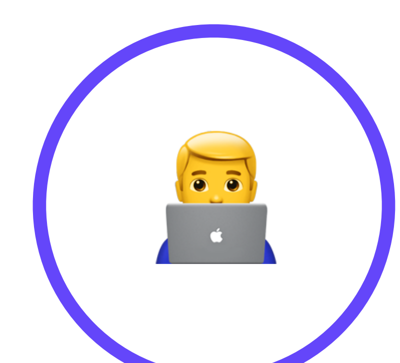
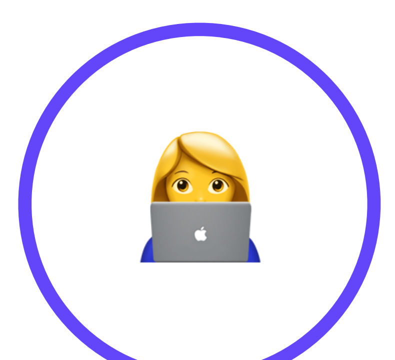
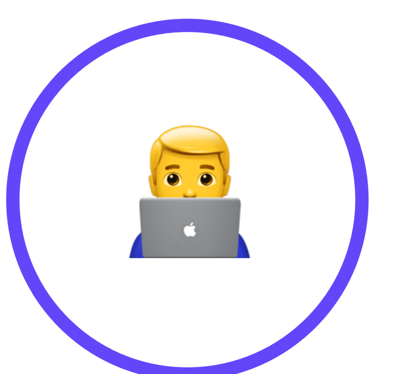
THE BIG IDEA

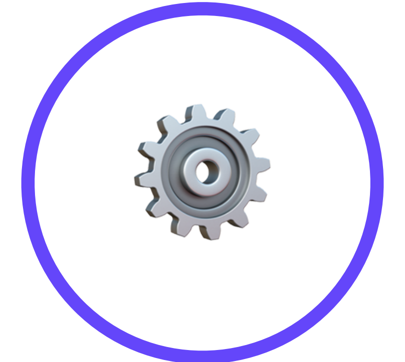
WHO'S ORG LOOKS LIKE THIS?

THE BIG IDEA

WHO'S ORG LOOKS LIKE THIS?

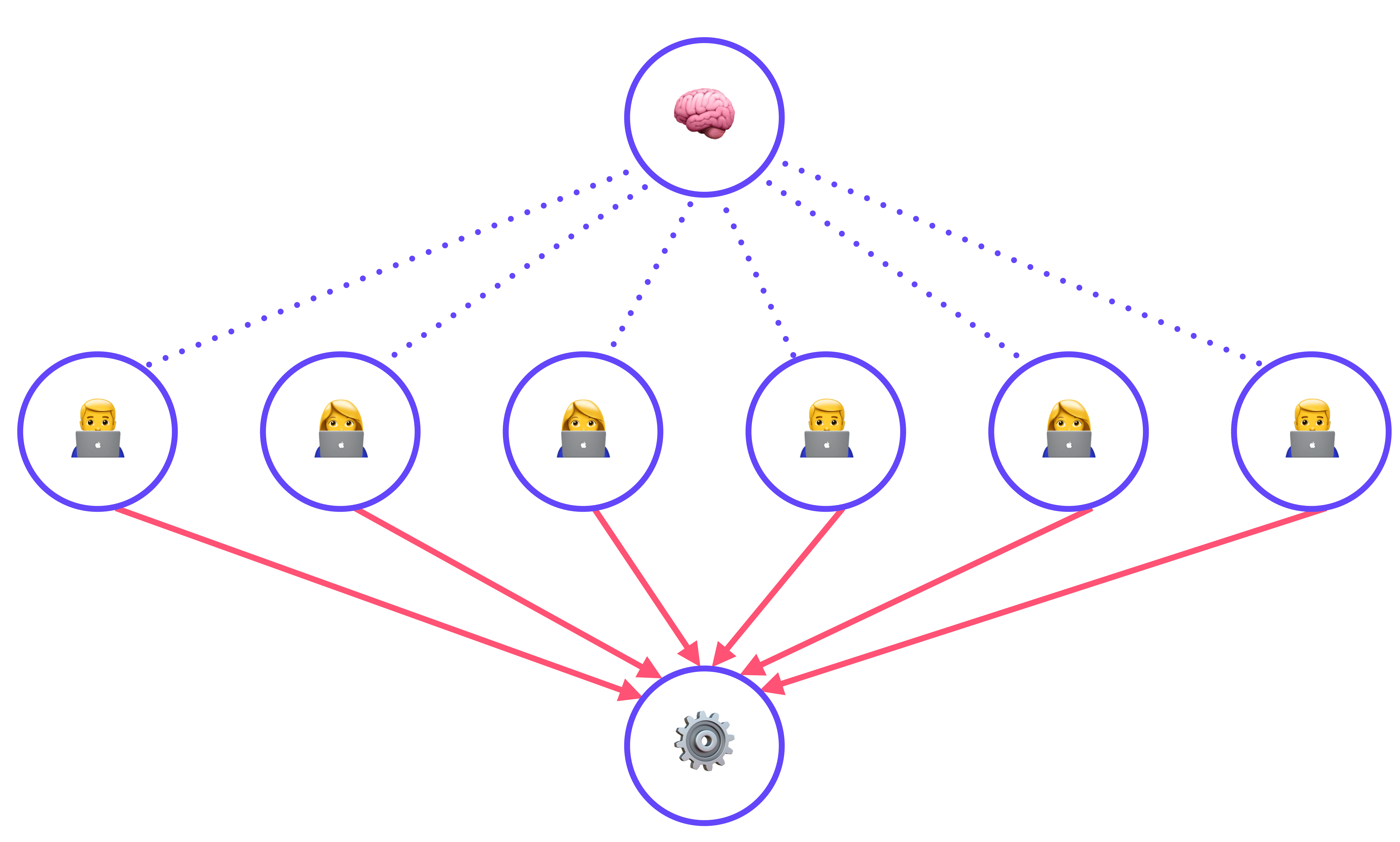




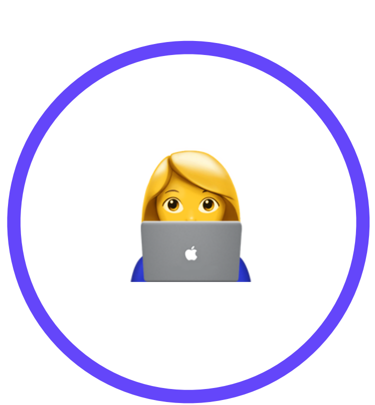
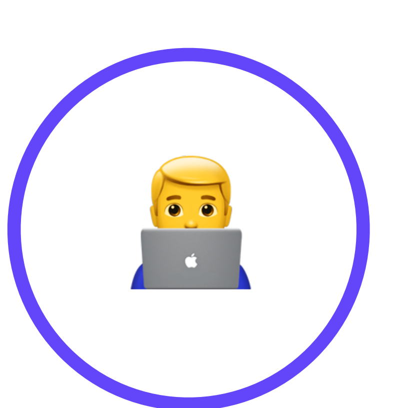
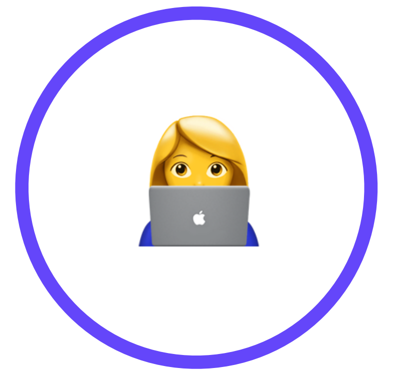
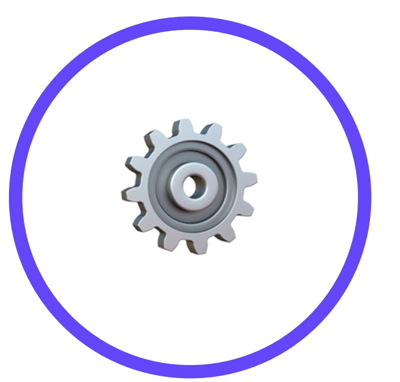
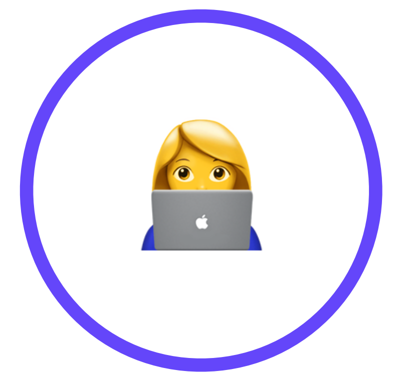
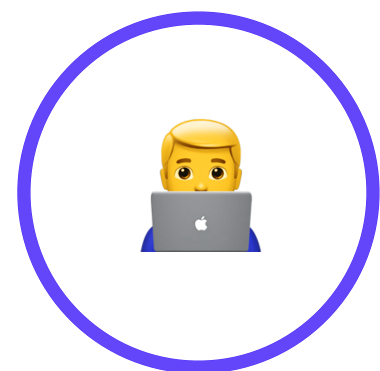
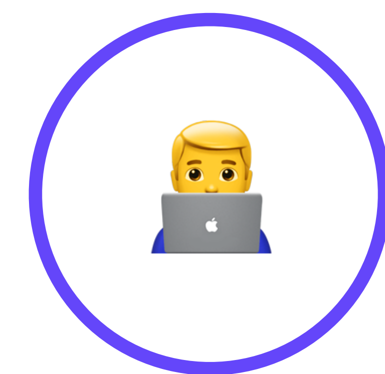
THE BIG IDEA

WHO'S ORG LOOKS LIKE THIS?



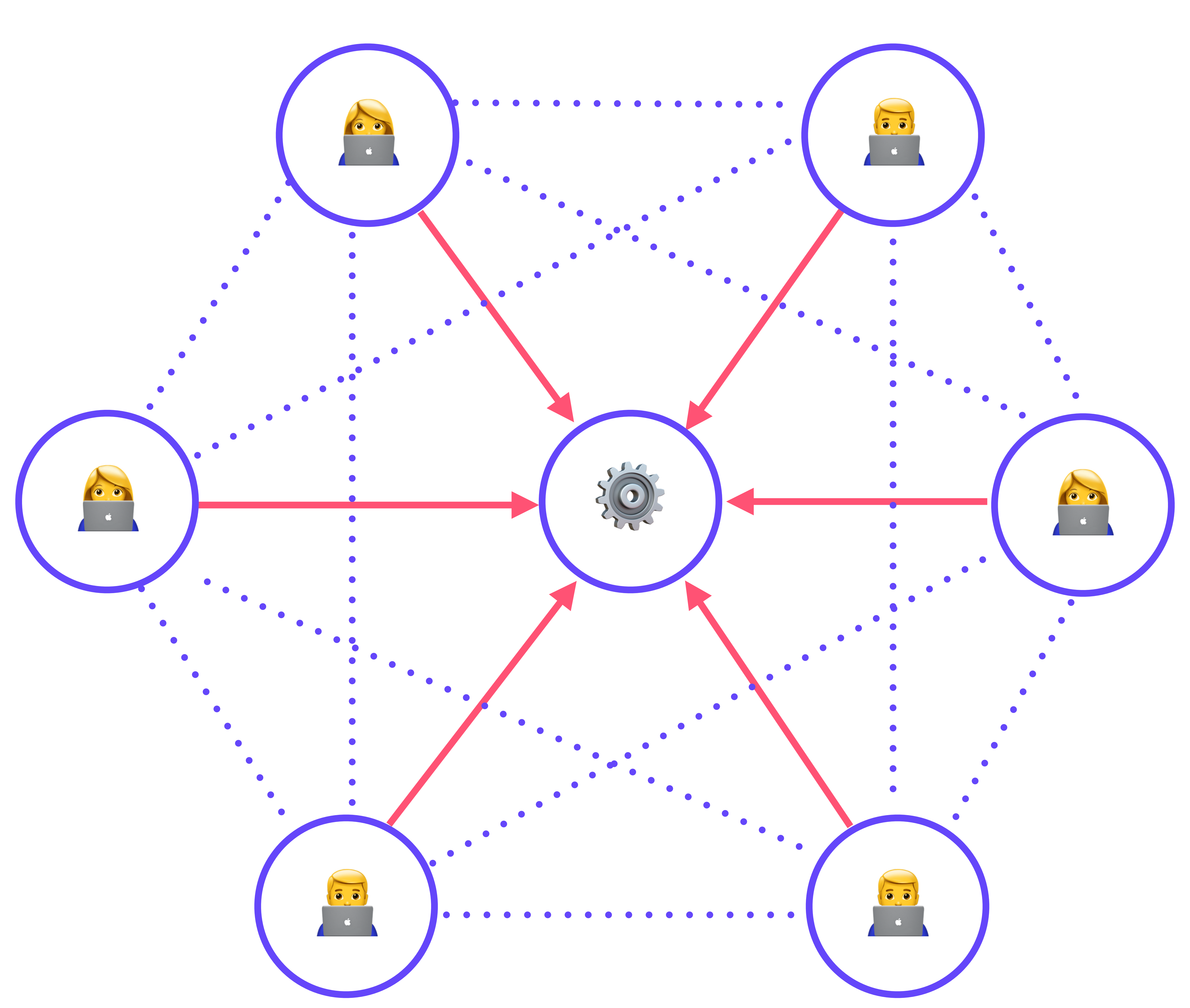
THE BIG IDEA

HOW ABOUT THIS?

THE BIG IDEA

HOW ABOUT THIS?



THE BIG IDEA

QUESTIONS

THE BIG IDEA

QUESTIONS

We want more type of features over time.
As a result, **complexity grows at an exponential rate**.

THE BIG IDEA

QUESTIONS

We want more type of features over time.
As a result, **complexity grows at an exponential rate**.

How do you make Elixir code
more flexible and easier to **reason about at scale**?

THE BIG IDEA

QUESTIONS

We want more type of features over time.
As a result, **complexity grows at an exponential rate**.

How do you make Elixir code
more flexible and easier to **reason about at scale**?

Do you think that the patterns we use today are
the **best possible patterns** for software?

THE BIG IDEA

QUESTIONS

We want more type of features over time.
As a result, **complexity grows at an exponential rate**.

How do you make Elixir code
more flexible and easier to **reason about at scale**?

Do you think that the patterns we use today are
the **best possible patterns** for software?

How will you write code in 2020, 2025, and 2050?

THE BIG IDEA

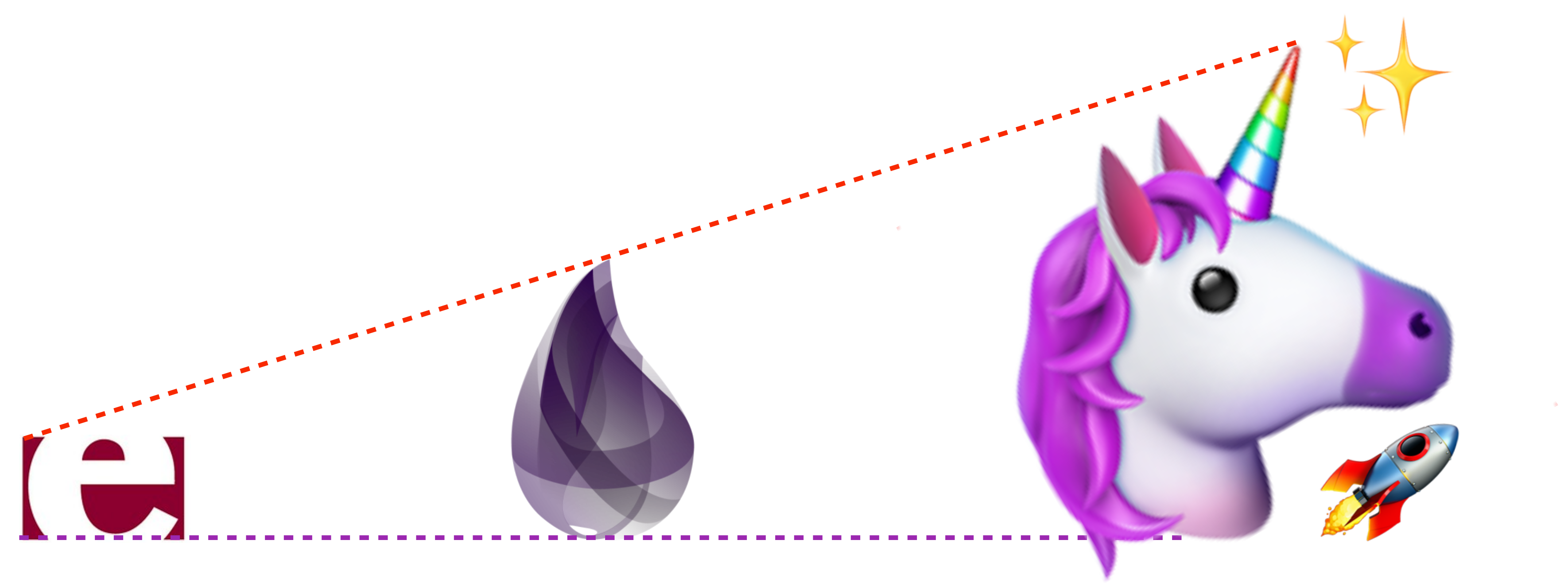
CORE

We need to evolve our approach:
focus on **domain** and **structure!**

THE BIG IDEA

CORE

We need to evolve our approach:
focus on **domain** and **structure!**



IN THE LARGE

IN THE LARGE

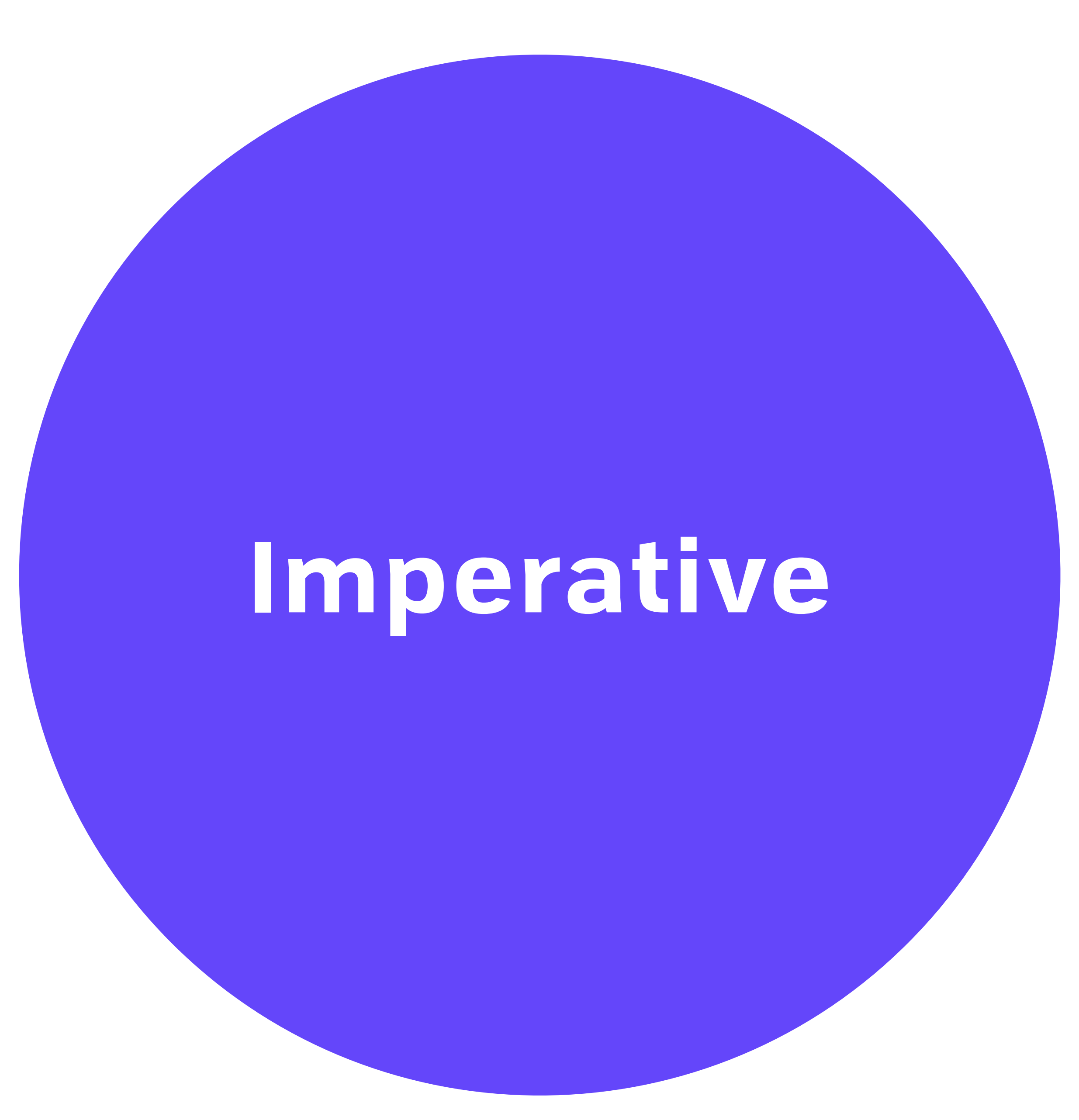
🌟

IN THE LARGE

CODE YOU USED TO WRITE

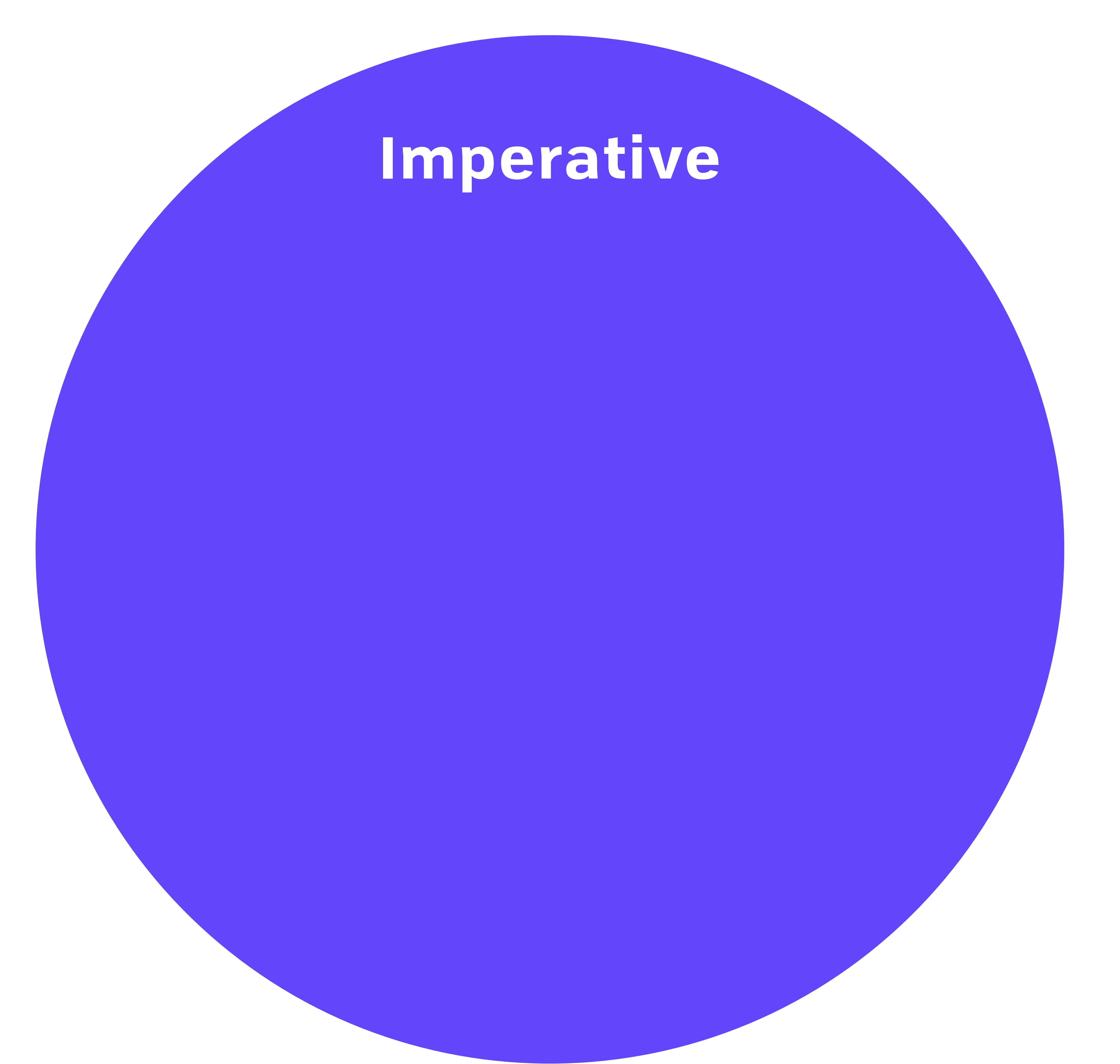
IN THE LARGE

CODE YOU USED TO WRITE



IN THE LARGE

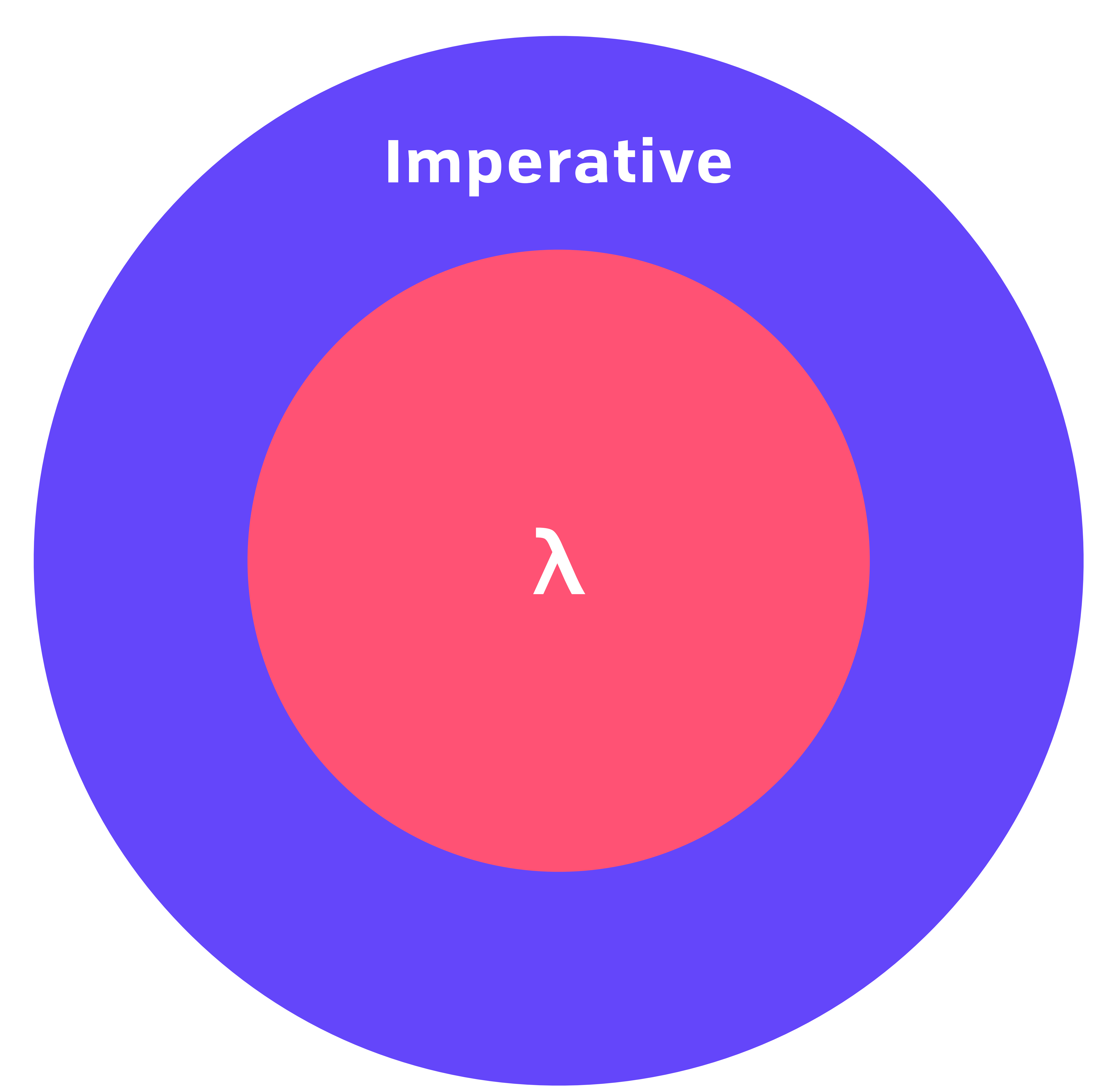
“GOOD” ELIXIR



* Functional core, imperative shell

IN THE LARGE

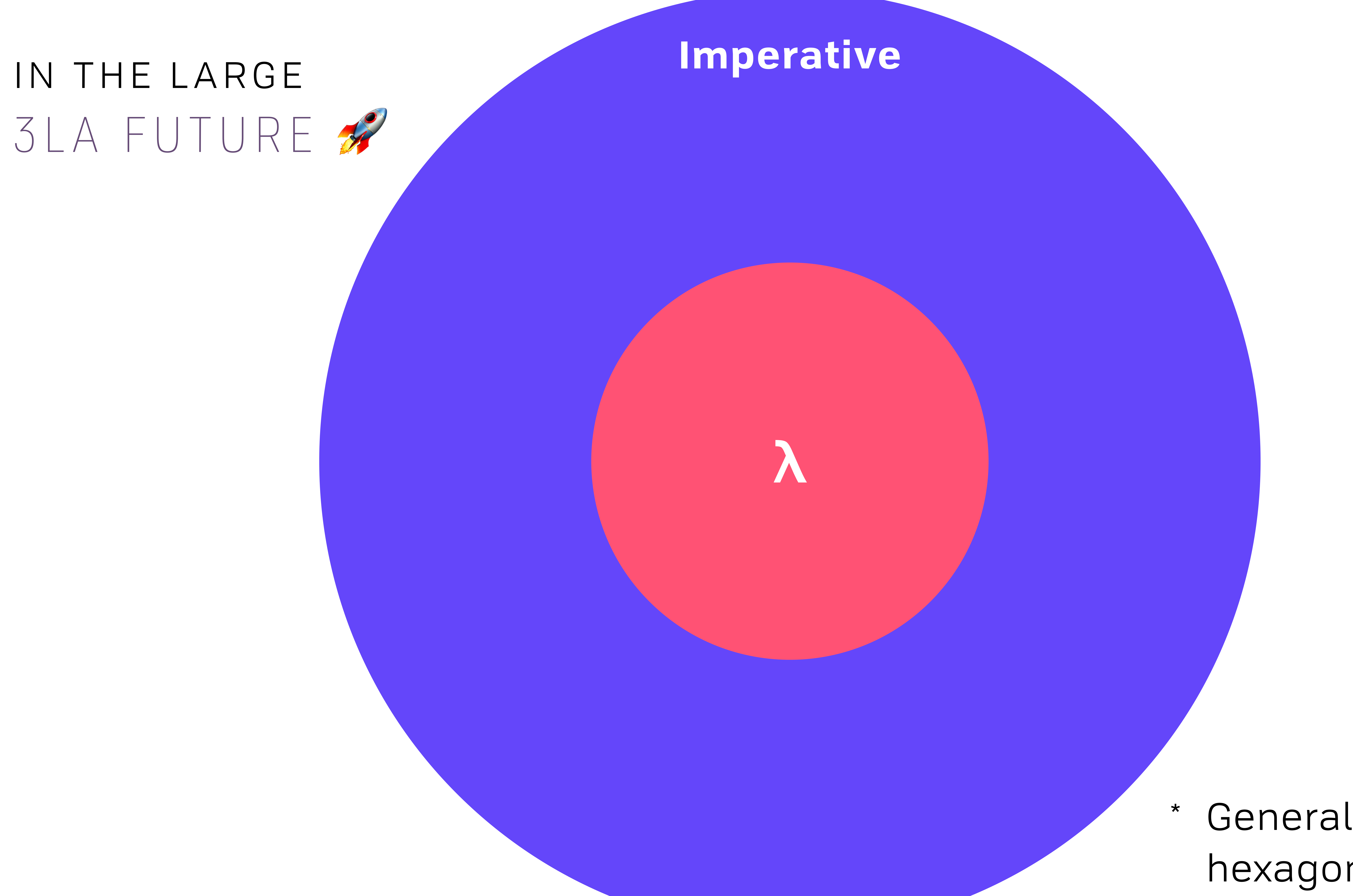
"GOOD" ELIXIR



* Functional core, imperative shell

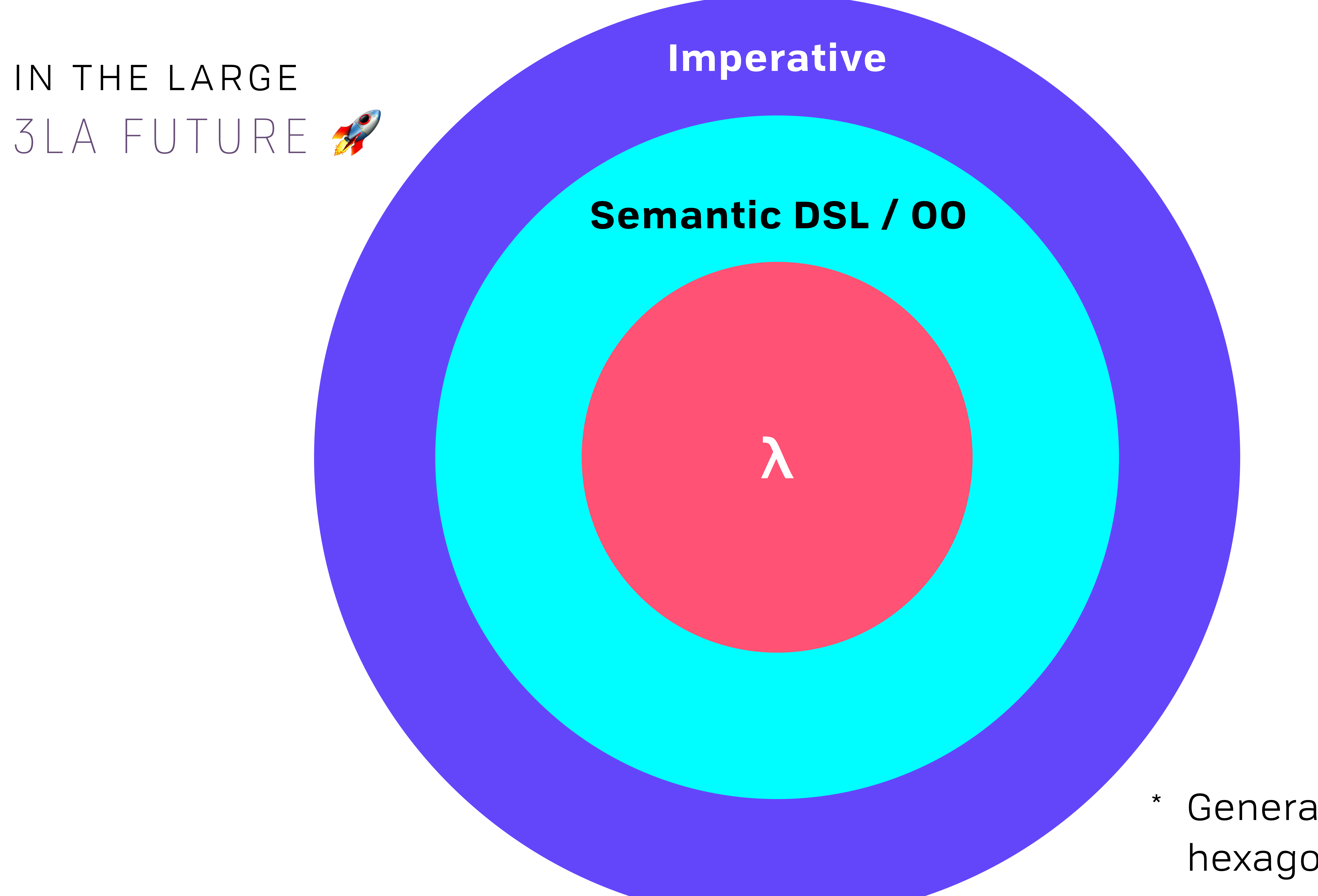
IN THE LARGE

3LA FUTURE 🚀



* Generalization of hexagonal/12-factor

IN THE LARGE
3LA FUTURE 🚀



* Generalization of hexagonal/12-factor

IN THE LARGE

PROP + MODEL TEST

```elixir
defmodule ListTest do
  use ExUnit.Case, async: true
  use ExUnitProperties

  property "++ is associative" do
    check all list_a <- list_of(term()),
              list_b <- list_of(term()),
              list_c <- list_of(term()) do

      ab_c = (list_a ++ list_b) ++ list_c
      a_bc = list_a  ++ (list_b ++ list_c)
      assert ab_c == a_bc
    end
  end
end
```

GOTOS CONSIDERED
HARMFUL

WHAT'S SO BAD ABOUT HAVING CONTROL?

GOTOS CONSIDERED HARMFUL

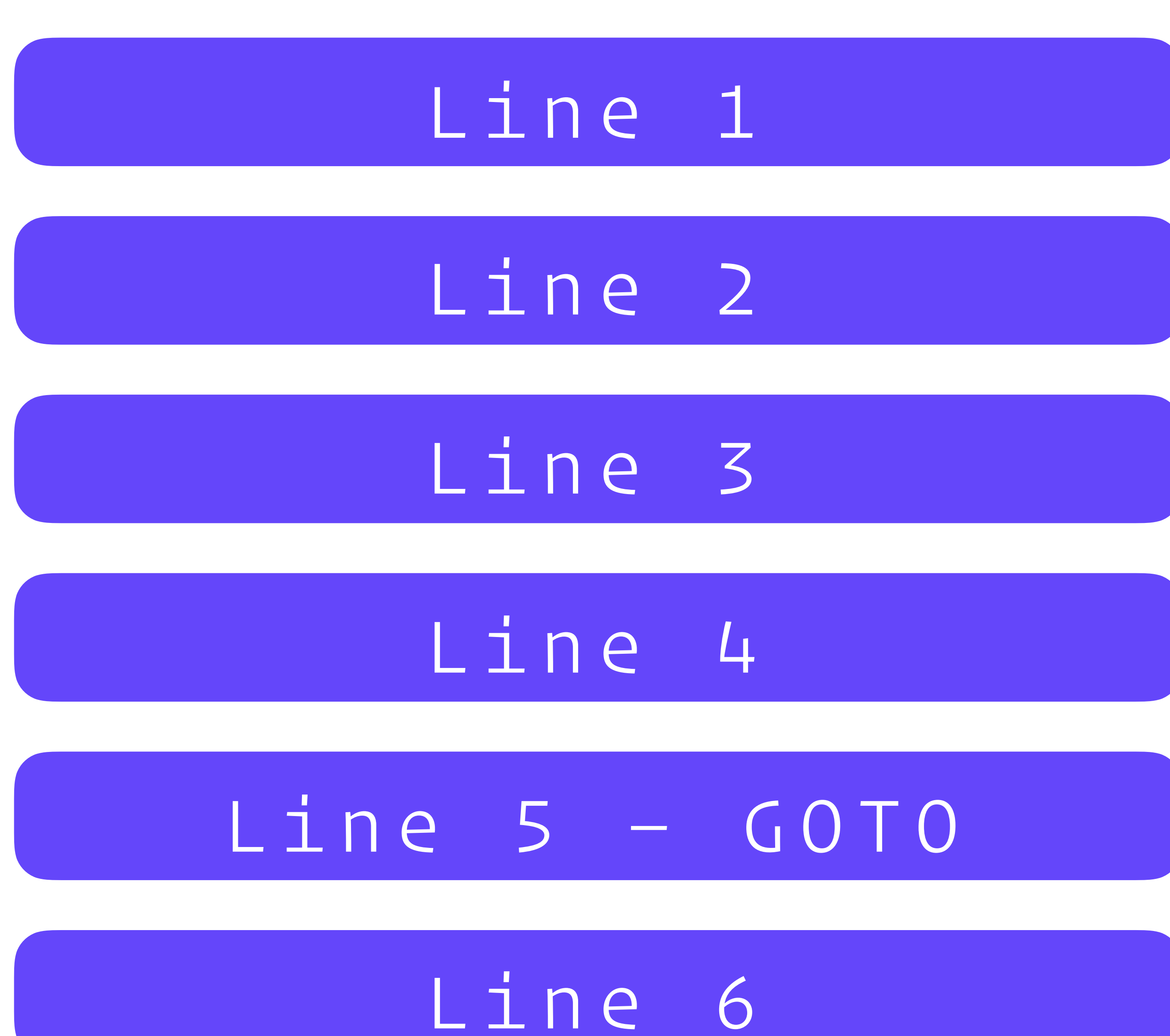
WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- *Extremely* flexible
- Highly concrete
- *Huge* number of **implicit** states

GOTOS CONSIDERED HARMFUL

WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

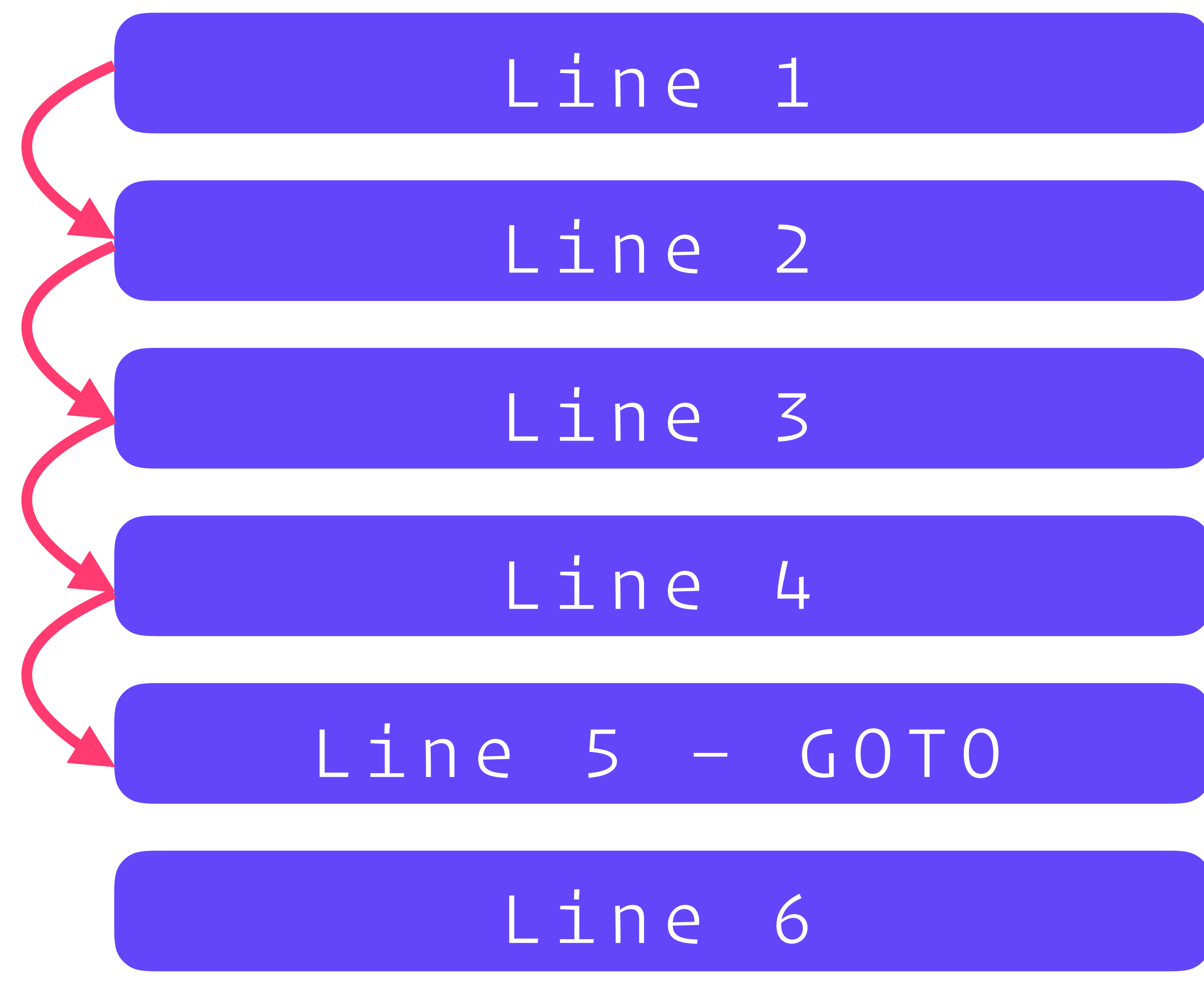
- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- *Extremely* flexible
- Highly concrete
- *Huge* number of **implicit** states



GOTOS CONSIDERED HARMFUL

WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

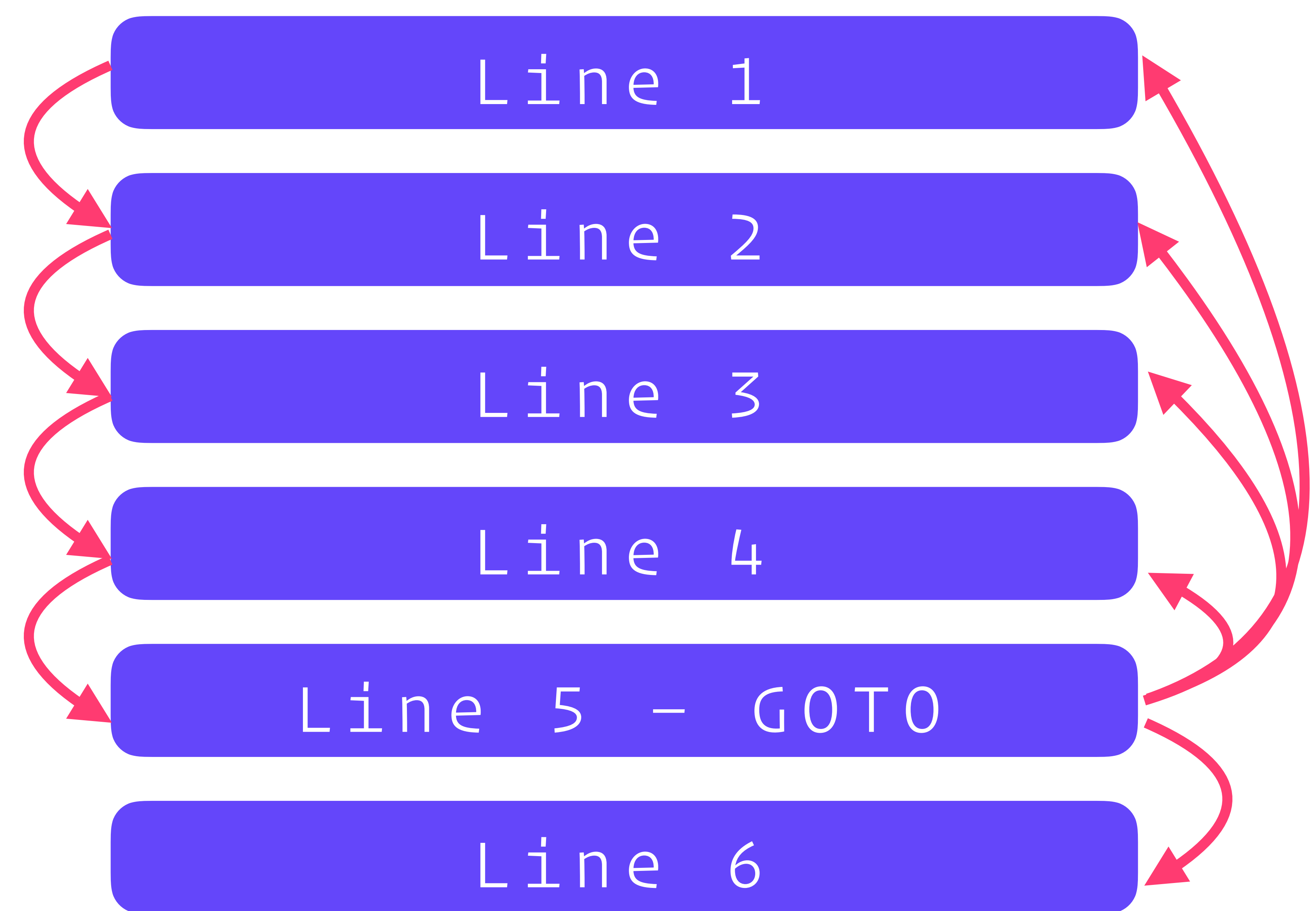
- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- *Extremely* flexible
- Highly concrete
- *Huge* number of **implicit** states



GOTOS CONSIDERED HARMFUL

WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

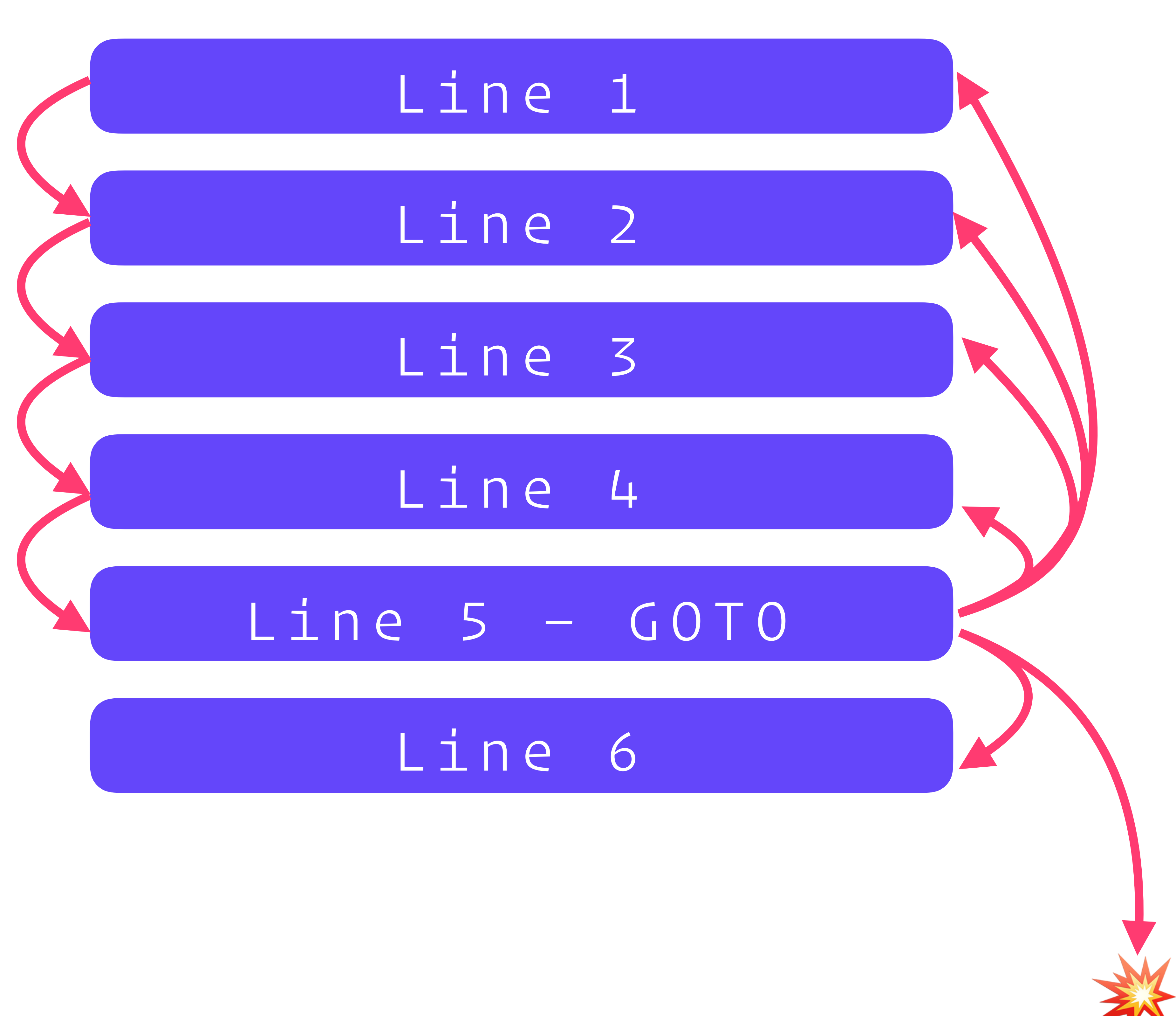
- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- *Extremely* flexible
- Highly concrete
- *Huge* number of **implicit** states



GOTOS CONSIDERED HARMFUL

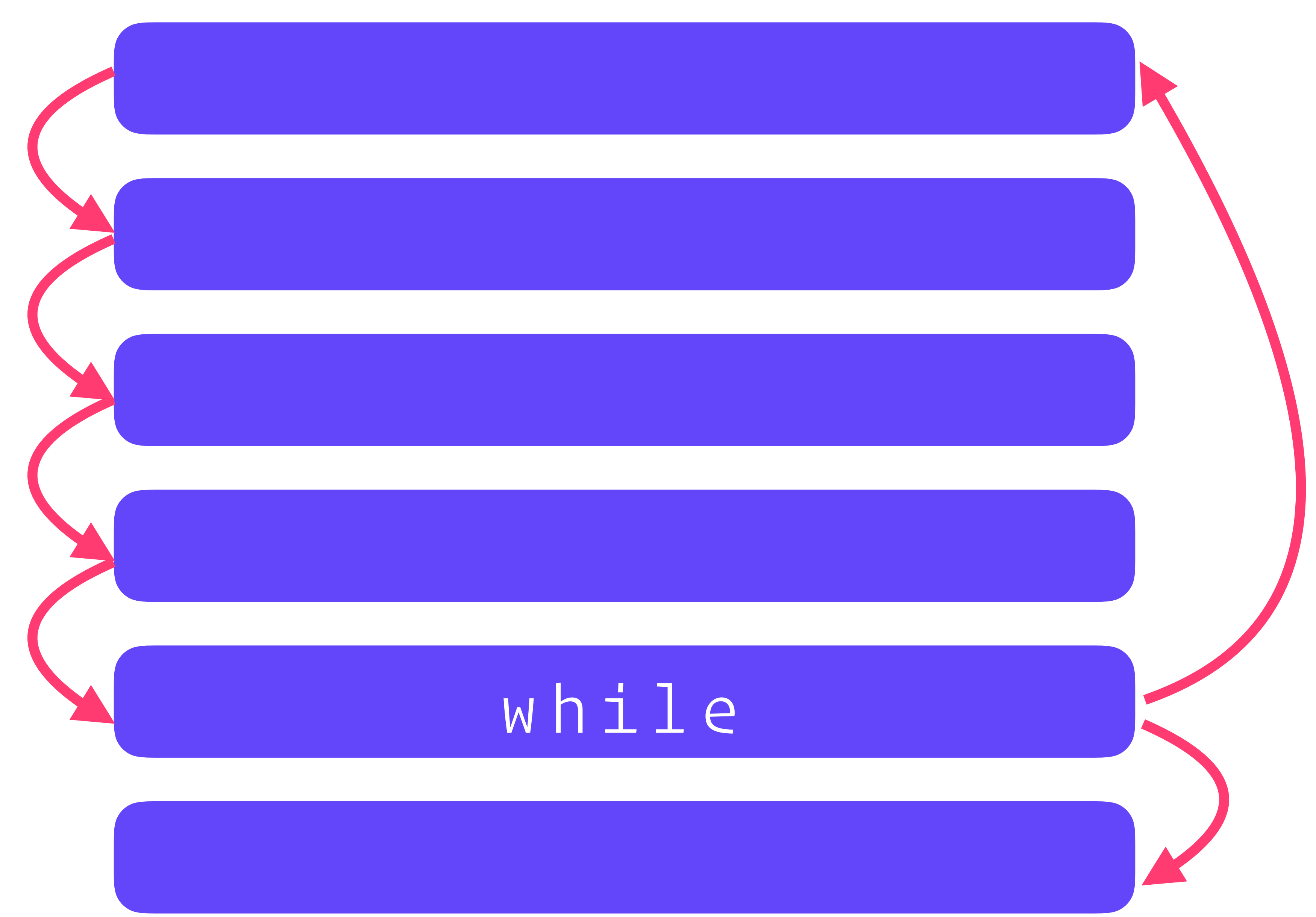
WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- *Extremely* flexible
- Highly concrete
- *Huge* number of **implicit** states



GOTOS CONSIDERED HARMFUL

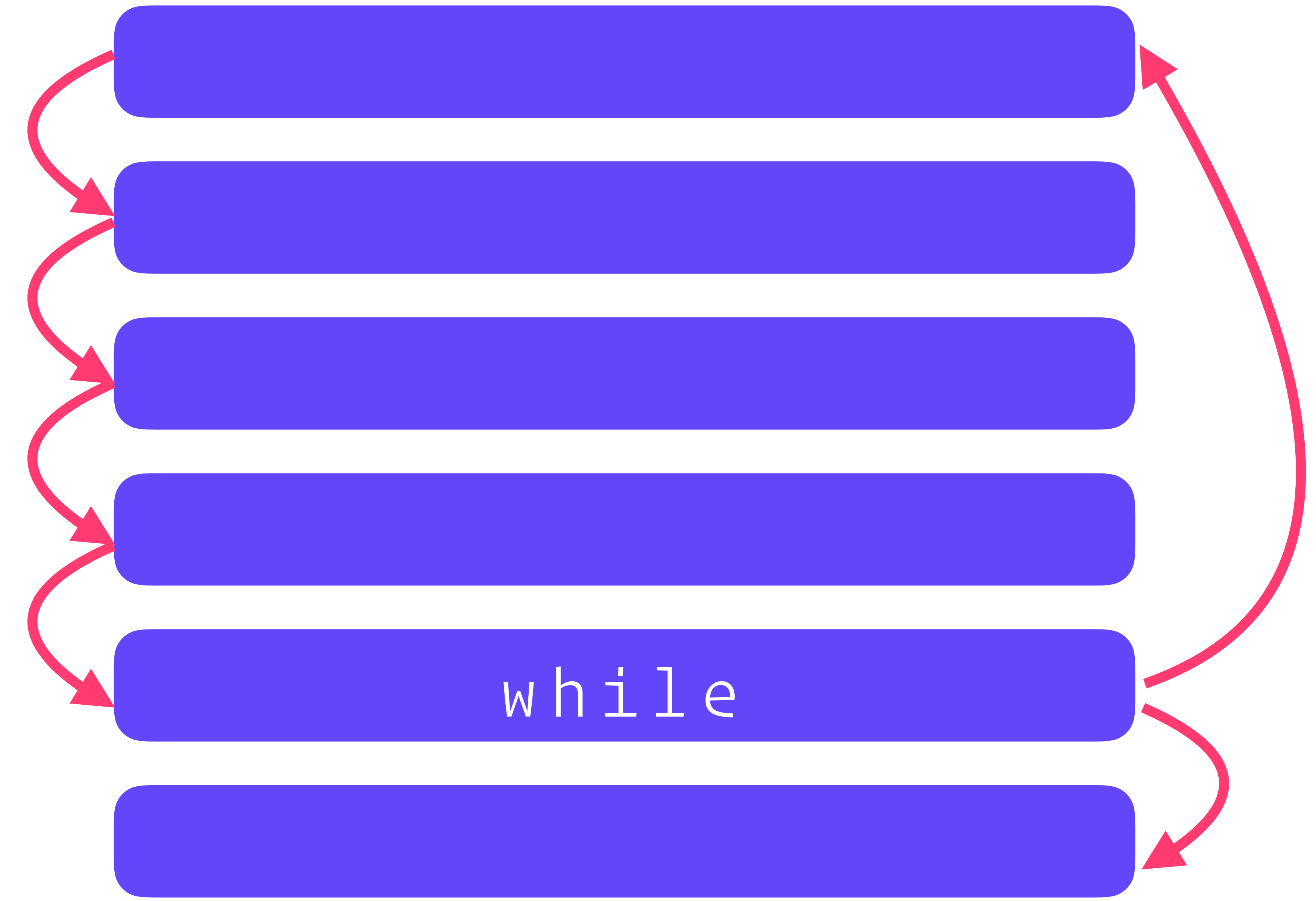
STRUCTURED PROGRAMMING



GOTOS CONSIDERED HARMFUL

STRUCTURED PROGRAMMING

- Subroutines
- Loops
- Switch/branching
- Named routines



GOTOS CONSIDERED HARMFUL

THE NEXT GENERATION 🚀

GOTOS CONSIDERED HARMFUL

THE NEXT GENERATION 🚀

- Functions
- Map
- Reduce
- Filter
- Constraint solvers

TRADEOFFS

TRADEOFFS

- Exchange granular control for structure
- **Meaning over mechanics**
- More human than machine
- *Safer!*
- Spectrum
  - Turing Tarpit
  - Church Chasm
  - Haskell Fan Fiction

GOTOS CONSIDERED HARMFUL

PAYOFF

Structured

Unstructured

GOTOS CONSIDERED HARMFUL

PAYOFF

Structured

Unstructured

COMPLEXITY

TIME

GOTOS CONSIDERED HARMFUL

PAYOFF

COMPLEXITY

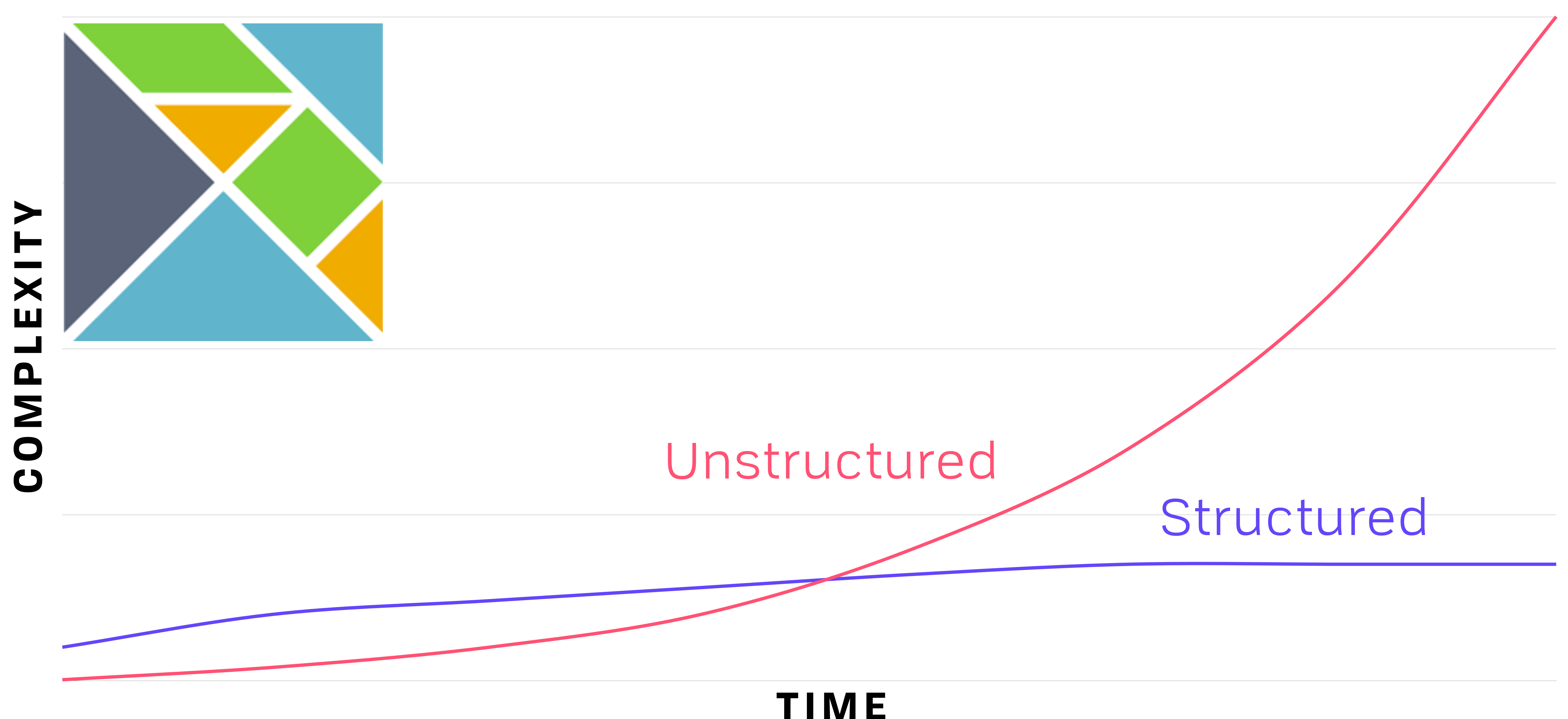
Unstructured

Structured

TIME

GOTOS CONSIDERED HARMFUL

PAYOFF



GOTOS CONSIDERED HARMFUL

PAYOFF

COMPLEXITY

Unstructured

Structured

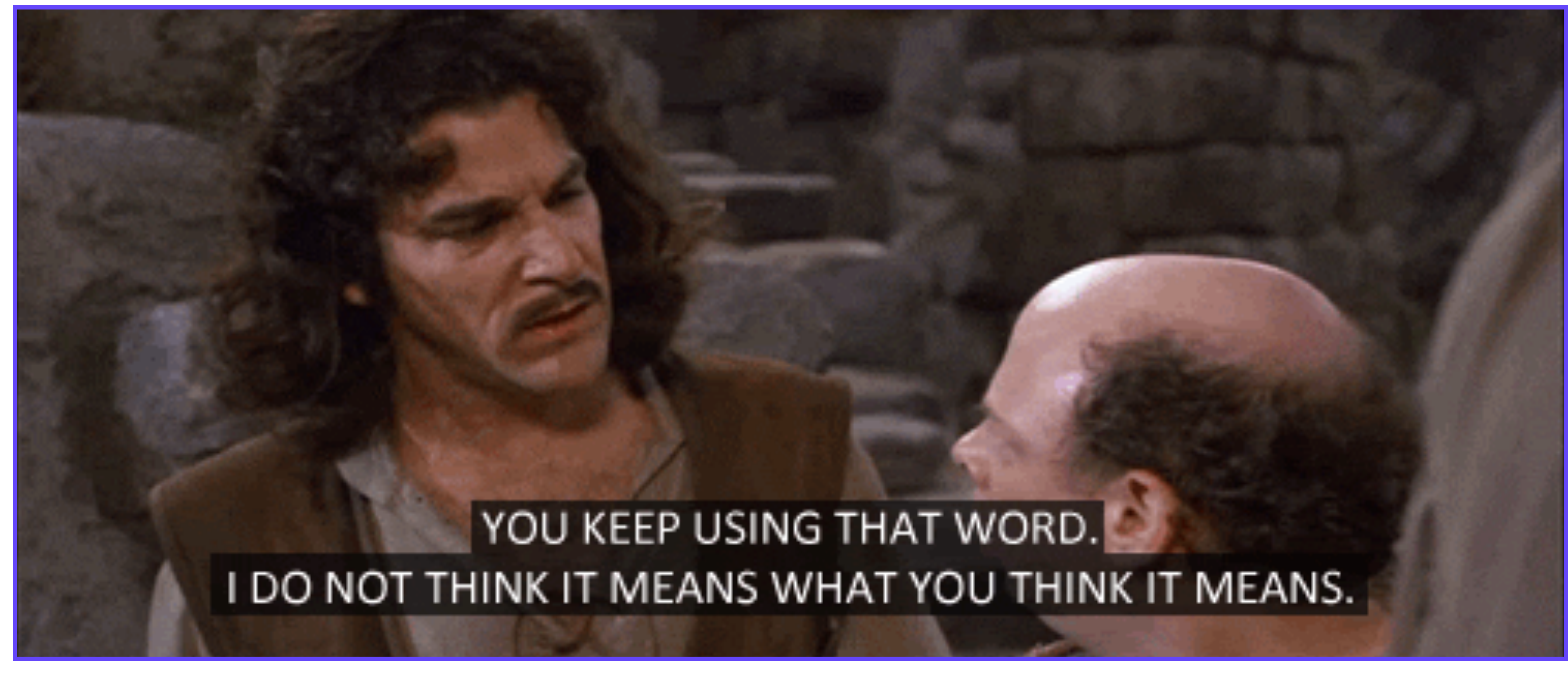
TIME

ON COMPLEXITY

ON COMPLEXITY

COMPLEXITY

COMPLEXITY



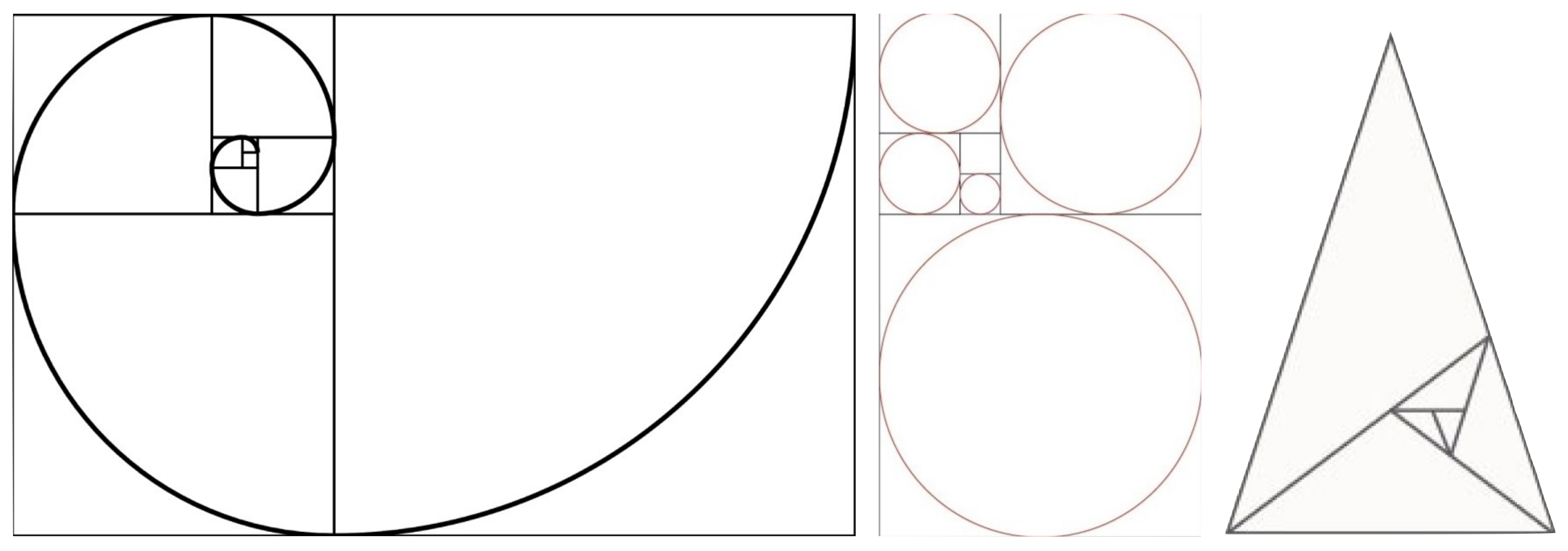
COMPLEXITY
THE BAD KIND ☠️

COMPLEXITY
THE BAD KIND ☠️

- Probably pretty familiar with this
- Euphemism for
  - *Complicated*
  - Inconsistent
  - No plan
  - “Unstructured mess”

COMPLEXITY

GOOD COMPLEXITY = DEEP



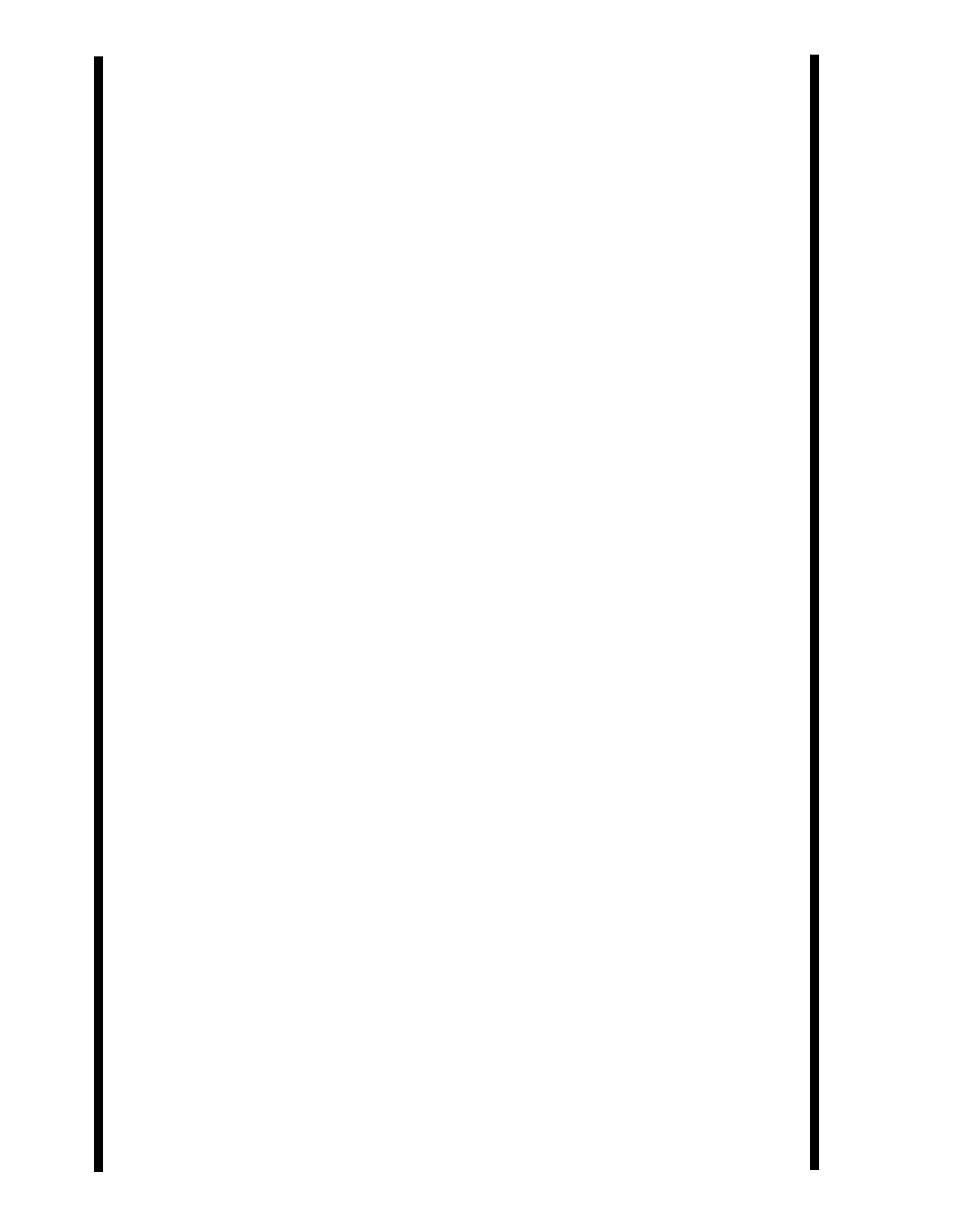
What do these have in common? $(a+b)/a \sim a / b$

COMPLEXITY

ORTHOGONAL COMPLECTING

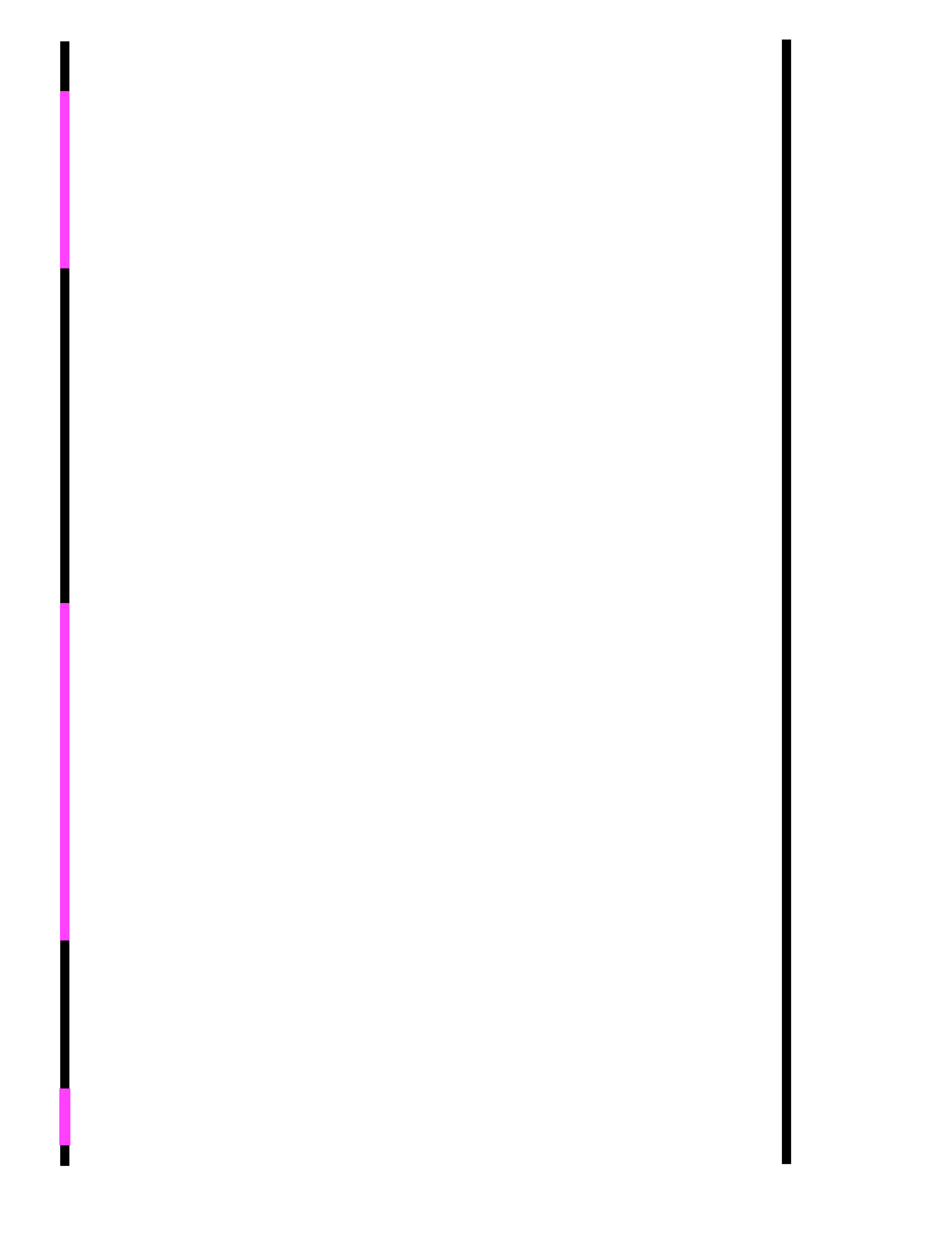
COMPLEXITY

ORTHOGONAL COMPLECTING



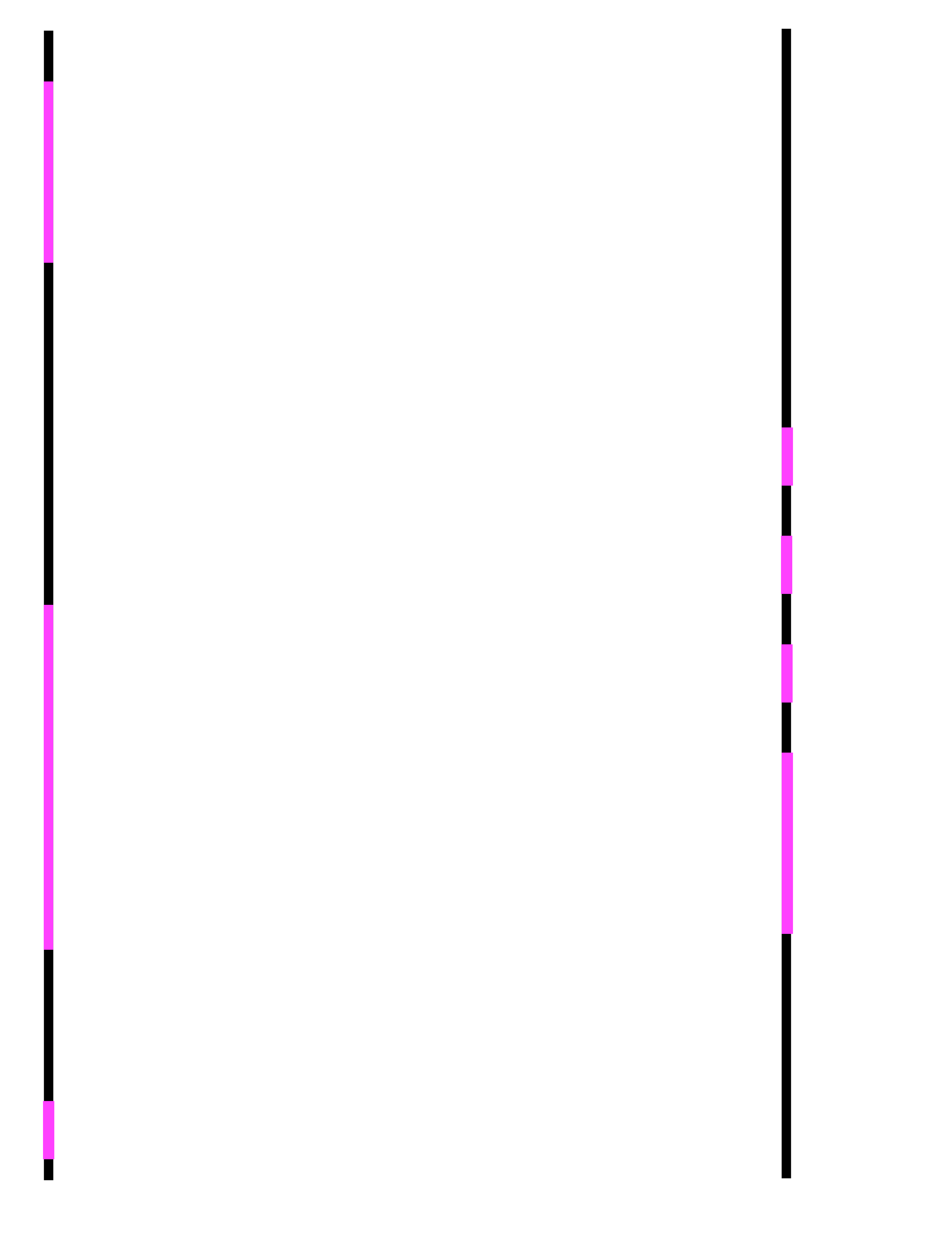
COMPLEXITY

ORTHOGONAL COMPLECTING



COMPLEXITY

ORTHOGONAL COMPLECTING

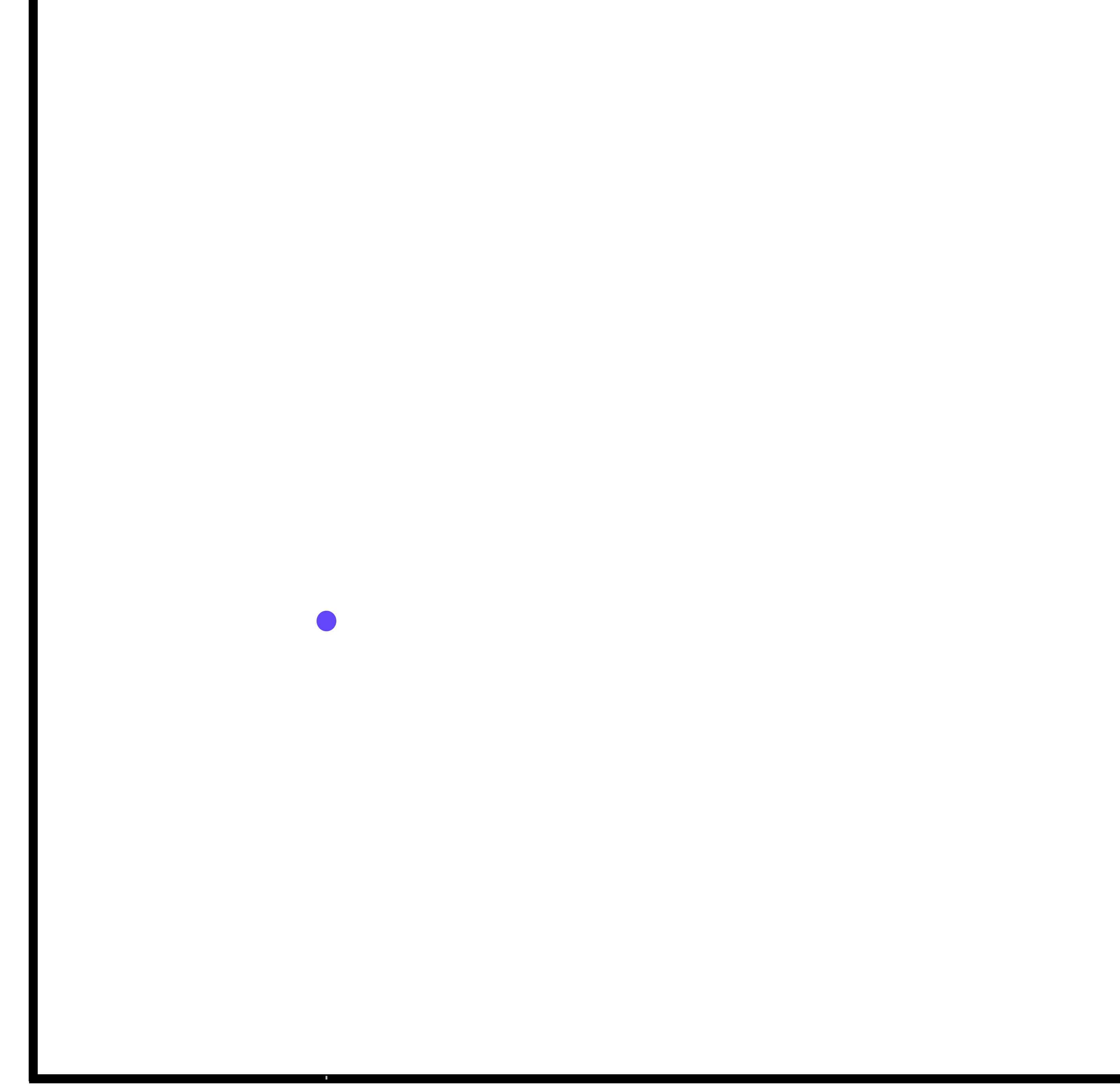


COMPLEXITY

ORTHOGONAL COMPLECTING

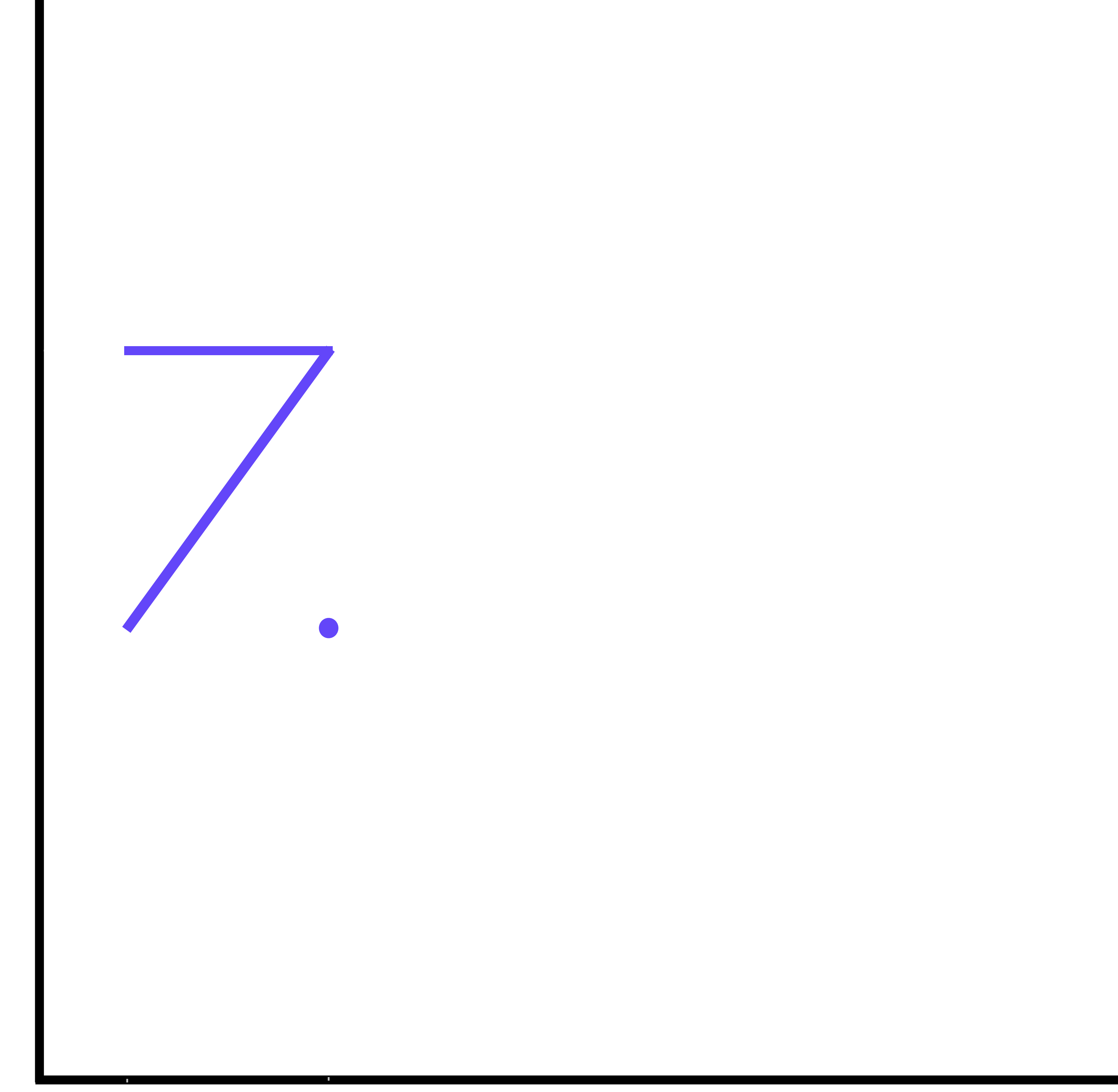
COMPLEXITY

ORTHOGONAL COMPLECTING



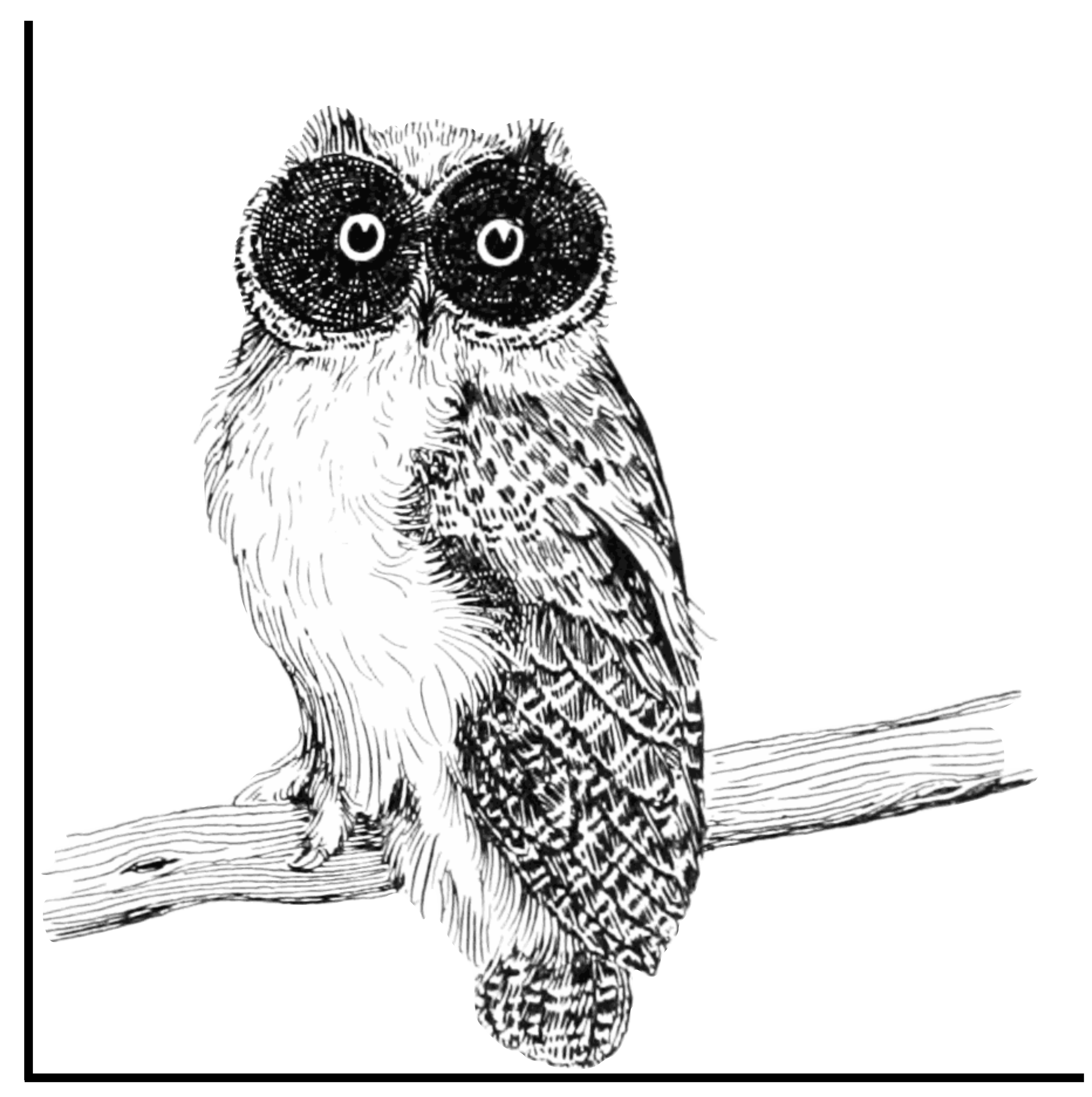
COMPLEXITY

ORTHOGONAL COMPLECTING



COMPLEXITY

ORTHOGONAL COMPLECTING

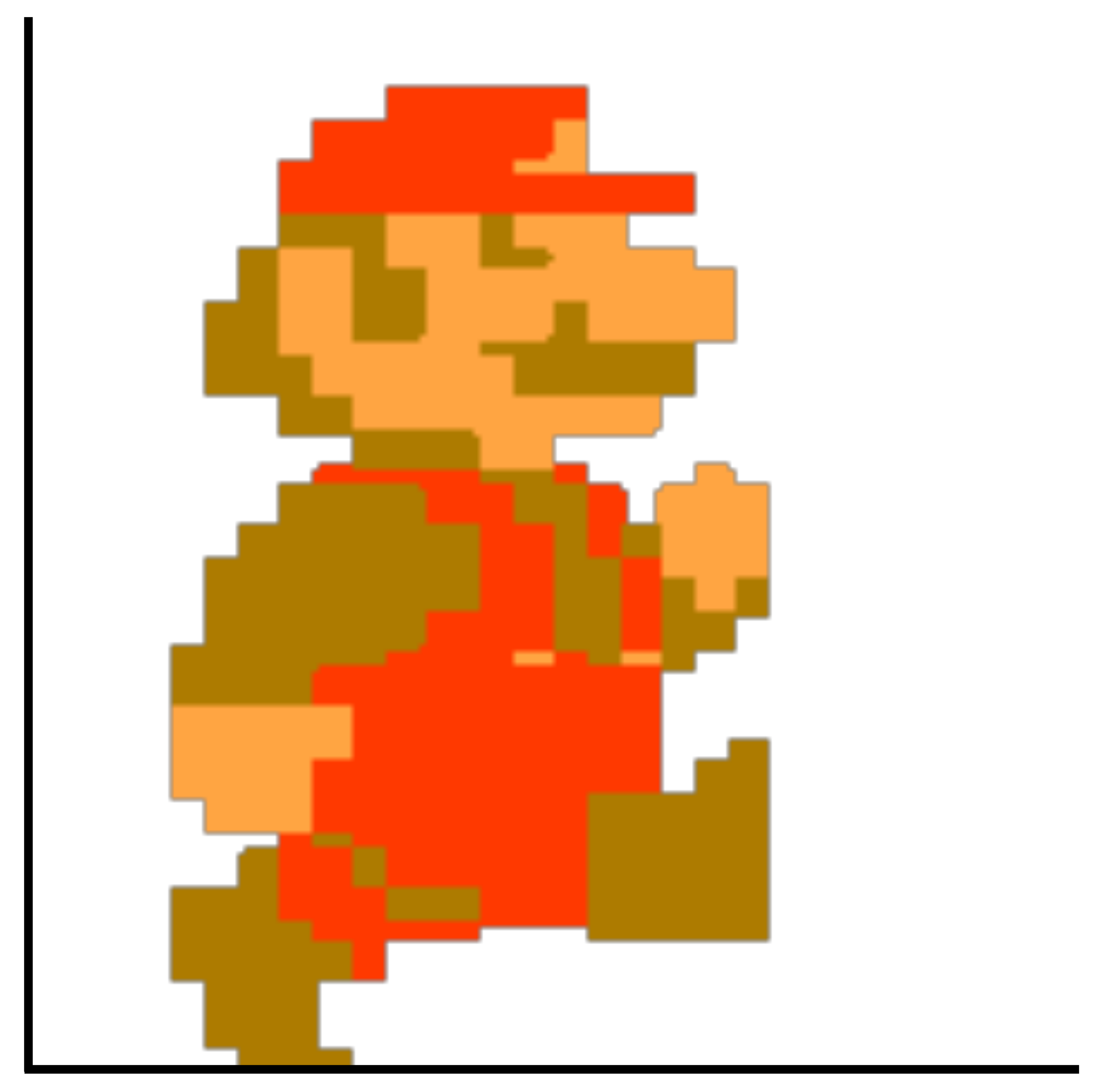


COMPLEXITY

ORTHOGONAL COMPLECTING

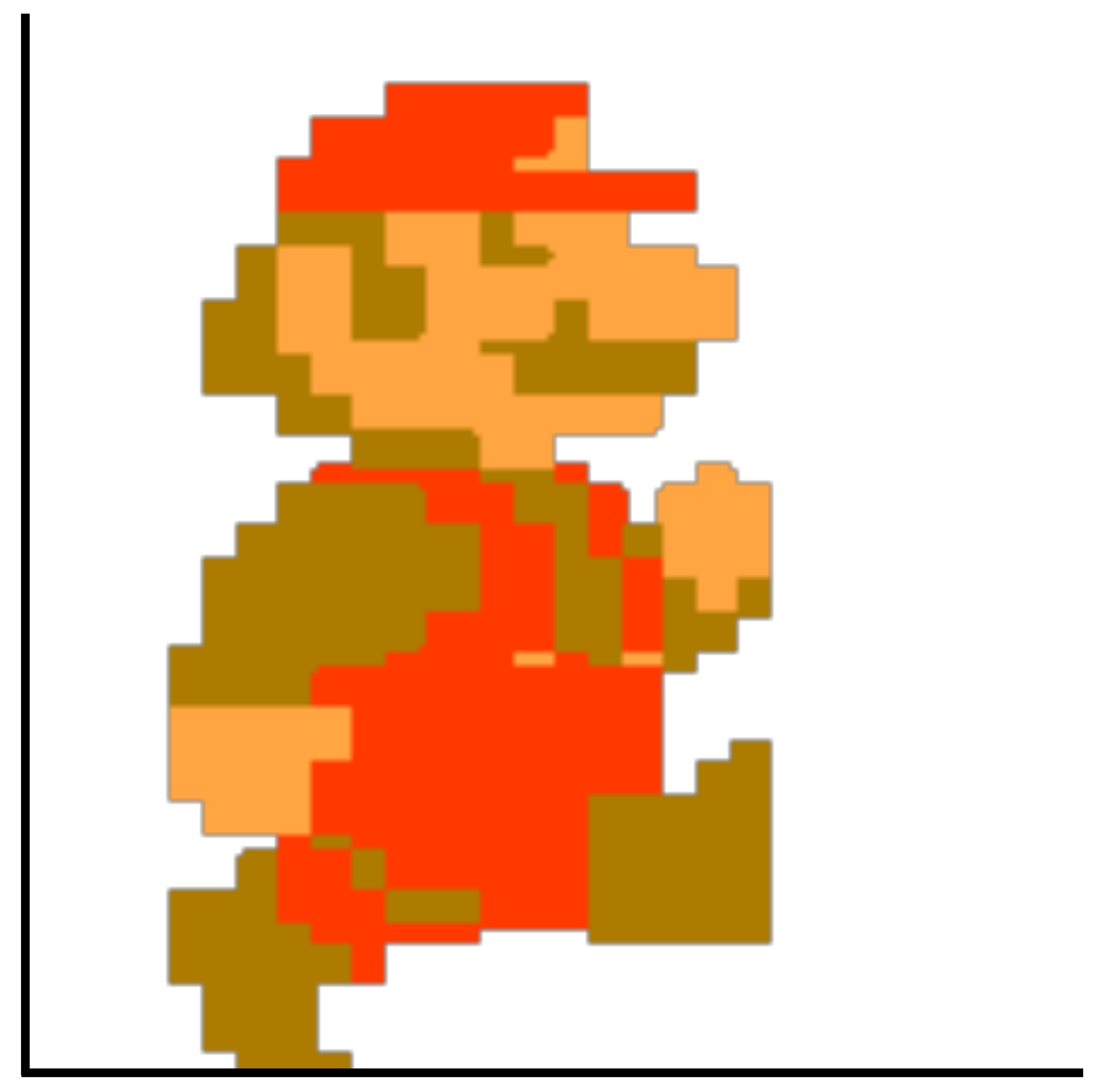
COMPLEXITY

ORTHOGONAL COMPLECTING



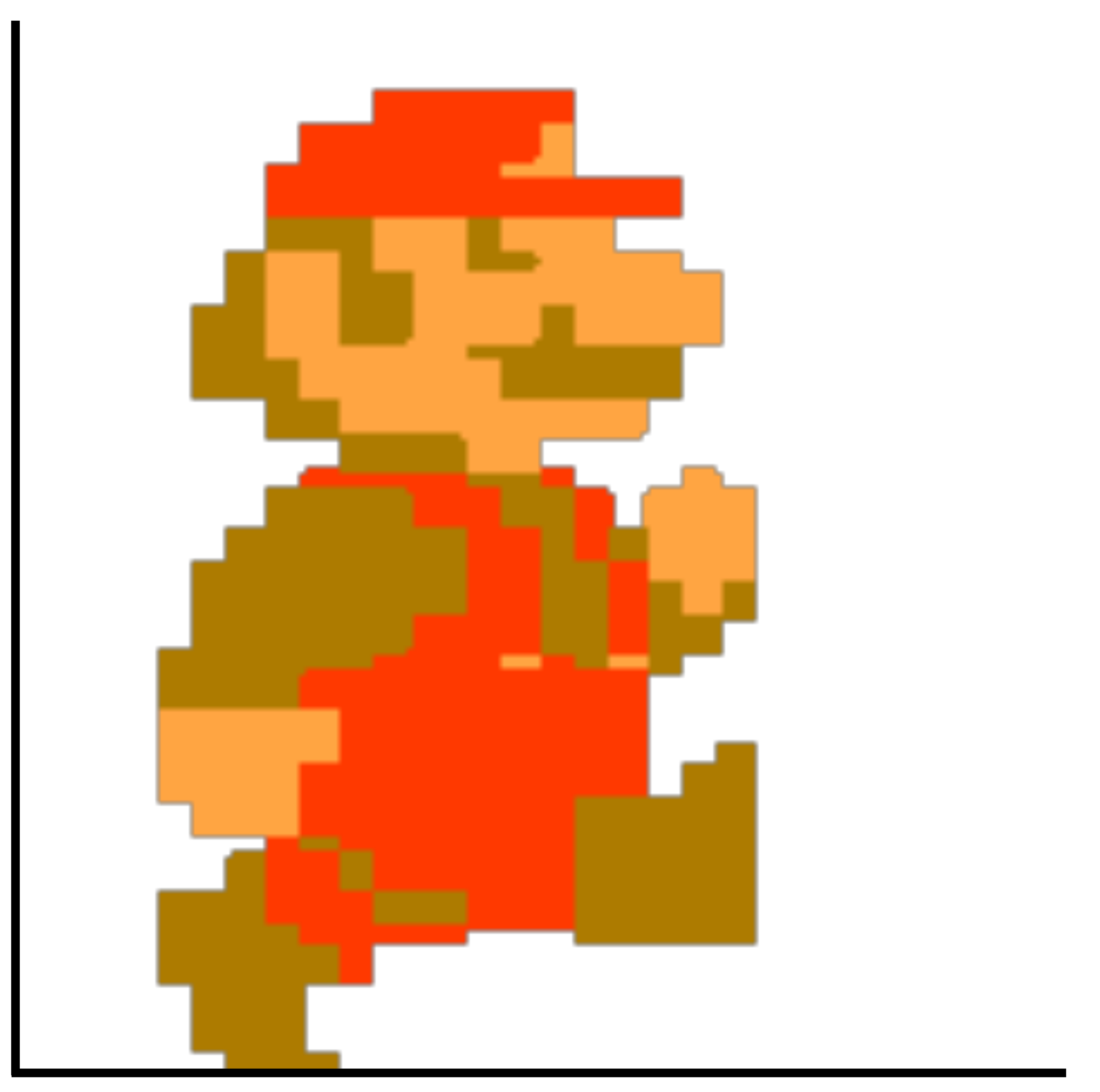
COMPLEXITY

ORTHOGONAL COMPLECTING



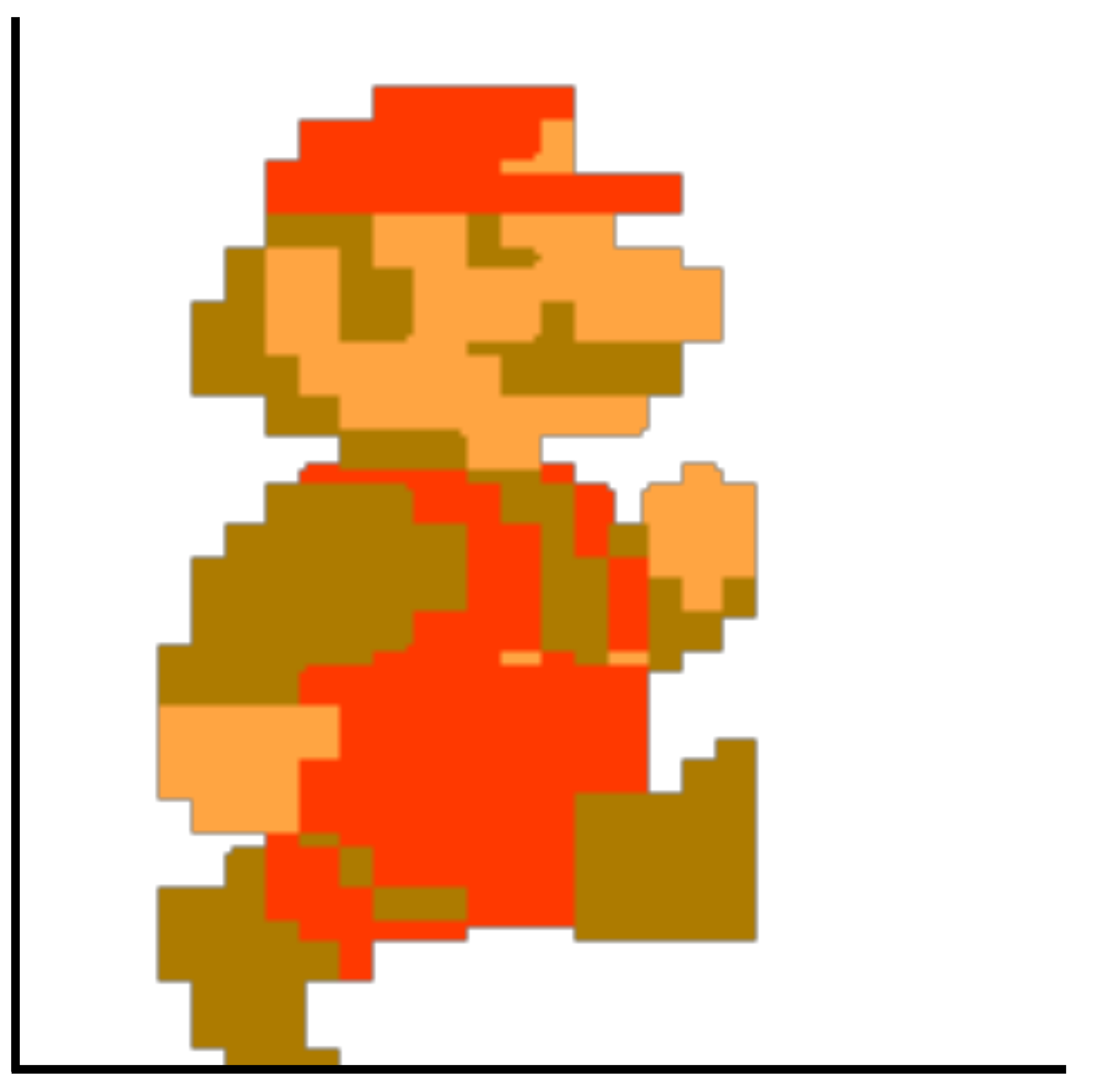
COMPLEXITY

ORTHOGONAL COMPLECTING



COMPLEXITY

ORTHOGONAL COMPLECTING



Structures: 4

Results: effectively limitless

COMPLEXITY

COMPLEX != COMPLICATED

COMPLEX != COMPLICATED

- Complex: interconnected parts
- Complicated: difficult to understand

COMPLEXITY

ACTOR ABYSS

COMPLEXITY

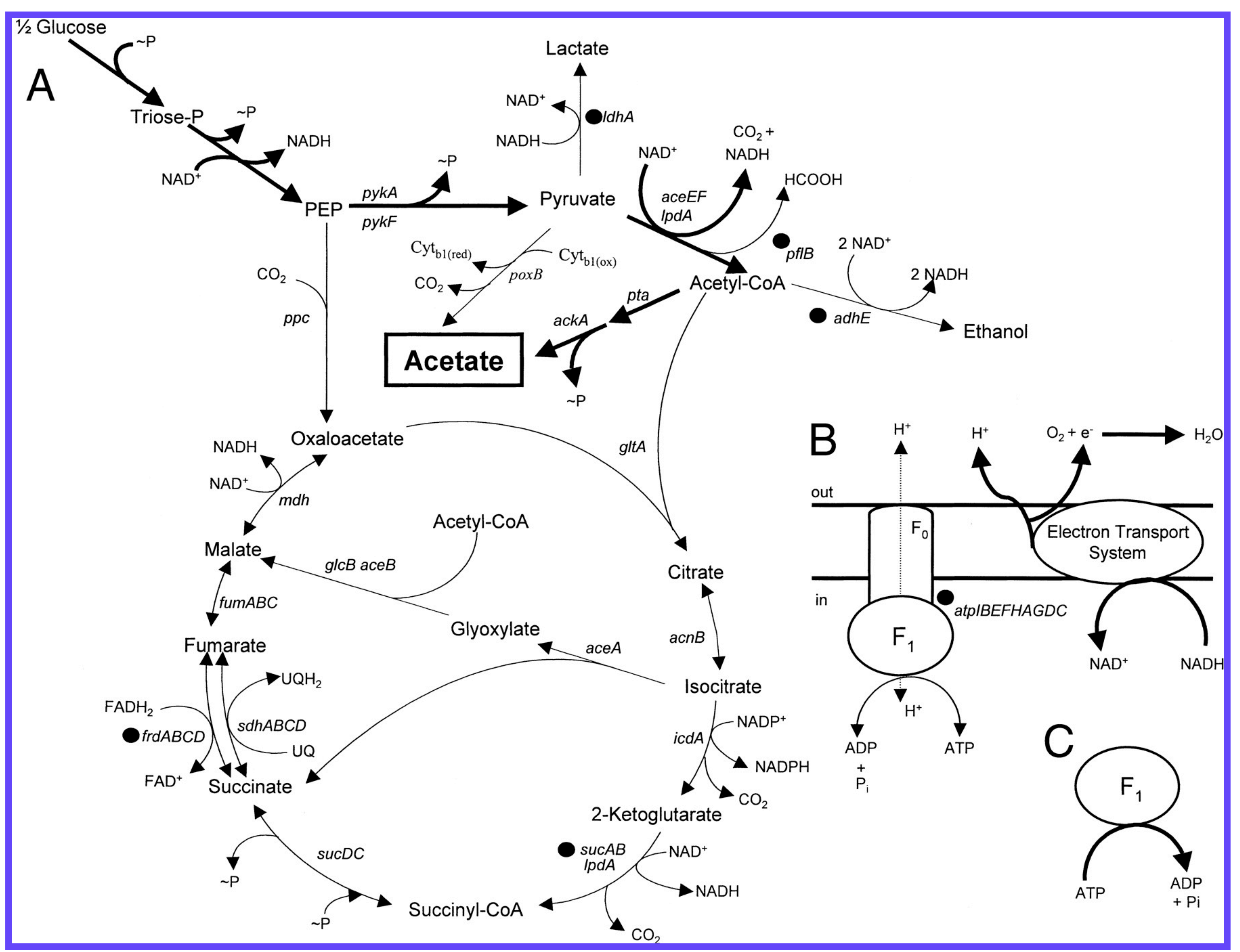
ACTOR ABYSS

1. Each step is very simple
2. Reasoning about dynamic organisms is hard
    1. Remember to store your data for crash recovery
    2. Called collaborator may not be there
3. Complexity grows faster than linear
4. Find common factors — your abstraction

COMPLEXITY

ACTOR ABYSS

1. Each step is very simple
2. Reasoning about dynamic organisms is hard
    1. Remember to store your data for crash recovery
    2. Called collaborator may not be there
3. Complexity grows faster than linear
4. Find common factors — your abstraction



FIGHTING GenSoup

FIGHTING GenSoup

FIGHTING GenSoup

GOOD INTERFACES != GOOD ABSTRACTIONS

GOOD INTERFACES != GOOD ABSTRACTIONS

- `GenServer` & co are actually *pretty low level*
  - Please add some semantics!
- Don’t reinvent the wheel every time 🎡
- Let’s look at a very common example

FIGHTING GenSoup

ABSTRACTION

ABSTRACTION

```
defprotocol KeyValue do
  def init(proxy)
  def get(db, value)
  def set(db, key, value)
end
```

FIGHTING GenSoup

SIMPLE CASE

```
defimpl KeyValue, for: Map do
  def init(_), do: %{}
  def get(db, value), do: Map.get(db, value, :not_found)
  def set(db, key, value), do: Map.put(db, key, value)
end
```

FIGHTING GenSoup

ASYNC CASE — BASE LAYER

```
defmodule ProcDB do
  use Agent

  defstruct [:pid]

  # Works with any inner data type!
  def start_link(starter), do: Agent.start_link(fn -> starter end)

  def get(pid, key) do
    Agent.get(pid, fn state -> KeyValue.get(state, key) end)
  end

  def set(pid, key, value) do
    Agent.update(pid, fn state -> KeyValue.set(state, key, value) end)
  end
end
```

FIGHTING GenSoup

ASYNC CASE — IMPLEMENTATION

```
defimpl KeyValue, for: %ProcDB do
  def init(_) do
    {:ok, pid} = ProcDB.startLink()
    %MyDB{pid: pid}
  end

  def get(%ProcDB{pid: pid}, key), do: ProcDB.get(pid, key)
  def set(%ProcDB{pid: pid}, key, value), do: ProcDB.set(pid, key, value)
end
```

WHAT DID WE GET?

FIGHTING GenSoup

WHAT DID WE GET?

- Common interface
- Encapsulate the detail
- Don’t have to think about mechanics anymore

FIGHTING GenSoup

ABSTRACTION = FOCUS/ESSENCE

FIGHTING GenSoup

ABSTRACTION = FOCUS/ESSENCE

```
defprotocol KeyValue do
  def init(proxy)
  def get(db, value)
  def set(db, key, value)
end
```

FIGHTING GenSoup

ABSTRACTION = FOCUS/ESSENCE

```
defprotocol KeyValue do
  def init(proxy)
  def get(db, value)
  def set(db, key, value)
end
```

ON ABSTRACTION & DSLS

ON ABSTRACTION & DSLS

NOT GETTING TRAPPED IN THE DETAILS

ABSTRACT & DSLS

COMMONALITIES

ABSTRACTION & DSLS

COMMONALITIES

```
list = [
         "a",
         "b",
         "c"
       ]
```

```
tuples = [
           {0, "a"}
           {1, "b"}
           {2, "c"}
         ]
```

```
map = %{
         0 => "a",
         1 => "b"
         2 => "c"
       }
```

ABSTRACTION & DSLS

COMMONALITIES

- They clearly have a similar structure
  - NOT equally expressive
  - `Enumerable`
    - Always converted to List
  - `Witchcraft.Functor`

```elixir
list = [
         "a",
         "b",
         "c"
       ]
```

```elixir
tuples = [
           {0, "a"}
           {1, "b"}
           {2, "c"}
         ]
```

```elixir
map = %{
        0 => "a",
        1 => "b"
        2 => "c"
      }
```

ABSTRACTION & DSLS

COMMONALITIES

ABSTRACTION & DSLS

COMMONALITIES

```
def seq_fun(input) do
  position = input / 5

  one_more = input + 1
  bang = inspect(one_more) <> "!"
  string = "#{one_more}#{bang}"

  String.at(string, round(position))
end
```

ABSTRACTION & DSLS

COMMONALITIES

```
def seq_fun(input) do
  position = input / 5

  one_more = input + 1
  bang = inspect(one_more) <> "!"
  string = "#{one_more}#{bang}"

  String.at(string, round(position))
end
```

```
def par_fun do
  position = Task.async(fn -> input / 5 end)
  string = Task.async(fn ->
    one_more = input + 1
    bang = inspect(one_more)  <> "!"
    "#{one_more}#{bang}"
  end)

  String.at(Task.await(string), round(Task.await(position)))
end
```

ABSTRACTION & DSLS

COMMONALITIES

- Different, but also have similar structure
  - Not very pipeable because 2 paths
  - …lots of duplicate code

```
def seq_fun(input) do
  position = input / 5

  one_more = input + 1
  bang = inspect(one_more) <> "!"
  string = "#{one_more}#{bang}"

  String.at(string, round(position))
end
```

```
def par_fun do
  position = Task.async(fn -> input / 5 end)
  string = Task.async(fn ->
    one_more = input + 1
    bang = inspect(one_more)  <> "!"
    "#{one_more}#{bang}"
  end)

  String.at(Task.await(string), round(Task.await(position)))
end
```

ABSTRACTION & DSLS

COMMONALITIES

- Different, but also have similar structure
  - Not very pipeable because 2 paths
  - …lots of duplicate code
- Why limit to only to two ways?

```elixir
def seq_fun(input) do
  position = input / 5

  one_more = input + 1
  bang = inspect(one_more) <> "!"
  string = "#{one_more}#{bang}"

  String.at(string, round(position))
end
```

```elixir
def par_fun do
  position = Task.async(fn -> input / 5 end)
  string = Task.async(fn ->
    one_more = input + 1
    bang = inspect(one_more)  <> "!"
    "#{one_more}#{bang}"
  end)

  String.at(Task.await(string), round(Task.await(position)))
end
```

ABSTRACTION & DSLS

START FROM RULES

START FROM RULES

- Describe **what** the overall solution looks like

ABSTRACTION & DSLS

START FROM RULES

- Describe **what** the overall solution looks like
- Choose **how** it gets run contextually

ABSTRACTION & DSLS

TWO-PHASE

TWO-PHASE

- Always a two-phase process
- Abstract, then concrete
- Do concretion at application boundary

ABSTRACTION & DSLS

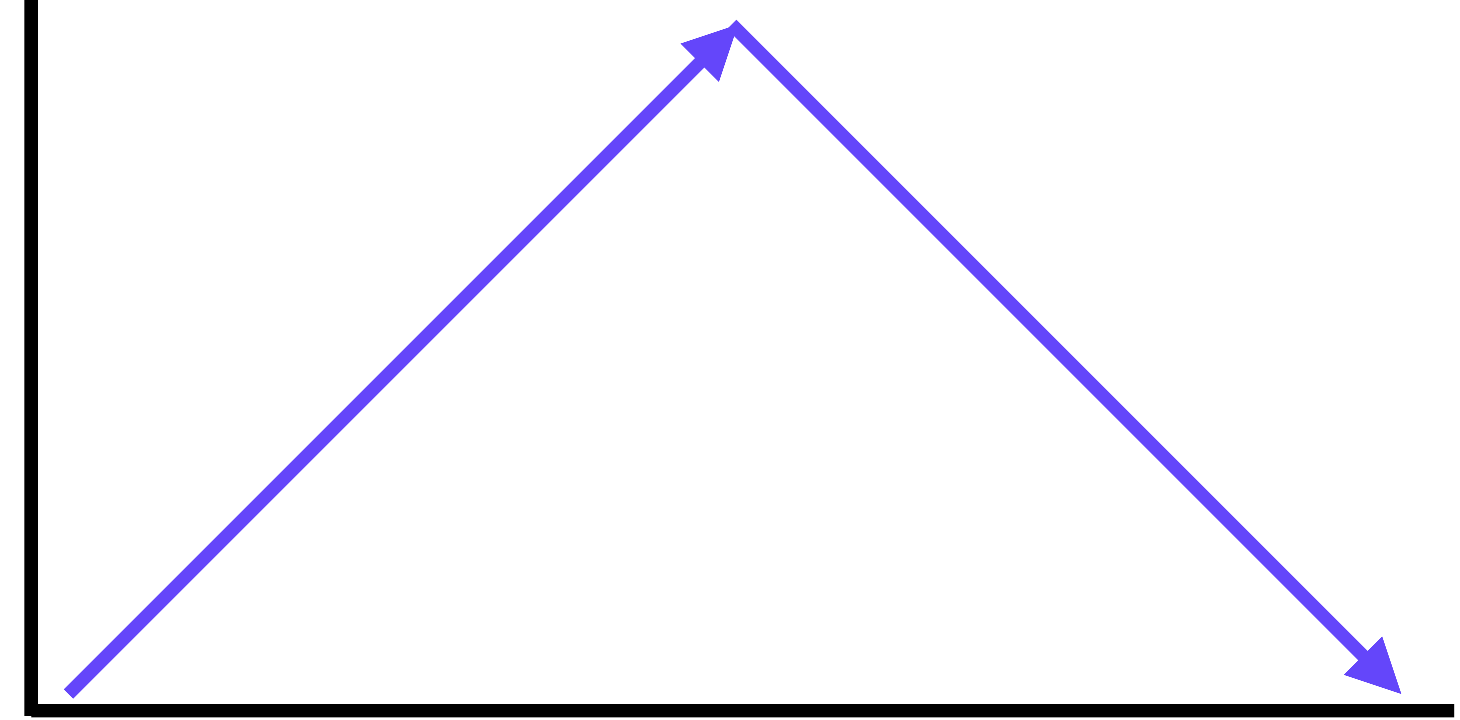
TWO-PHASE

- Always a two-phase process
- Abstract, then concrete
- Do concretion at application boundary

ABSTRACTION & DSLS

TWO-PHASE

- Always a two-phase process
- Abstract, then concrete
- Do concretion at application boundary



ABSTRACTION & DSLS

IMPROVING `Kernel`

ABSTRACTION & DSLS

IMPROVING `Kernel`

- Fallback keys
- Bang-functions

IMPROVING `Kernel` — FALLBACK KEYS

ABSTRACTION & DSLS

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition

```
def get(map, key, default \\ nil)
```

IMPROVING Kernel — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4
```

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!

```elixir
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4
```

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let’s abstract default values!

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value
```

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4
```

ABSTRACTION & DSLS

IMPROVING Kernel — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4
```

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let’s abstract default values!

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4

[] |> List.first() |> fallback(:empty)
#=> :empty
```

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4

[] |> List.first() |> fallback(:empty)
#=> :empty
```

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4

[] |> List.first() |> fallback(:empty)
#=> :empty
```

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)
  - Ad hoc function extension

```
def get(map, key, ~~default \\ nil)~~

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4

[] |> List.first() |> fallback(:empty)
#=> :empty
```

IMPROVING `Kernel` — BANG FUNCTIONS

ABSTRACTION & DSLS

IMPROVING Kernel — BANG FUNCTIONS

```
Map.fetch!(%{a: 1}, :b, fun)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

ABSTRACTION & DSLS

IMPROVING `Kernel` — BANG FUNCTIONS

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

IMPROVING `Kernel` — BANG FUNCTIONS

💣
```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

ABSTRACTION & DSLS

IMPROVING `Kernel` — BANG FUNCTIONS

```
💣Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}
```

IMPROVING `Kernel` — BANG FUNCTIONS

```
💣 Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

💣 ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Abstracted out `foo!/*` from `foo/*`

ABSTRACTION & DSLS

IMPROVING `Kernel` — BANG FUNCTIONS

Abstracted out `foo!/*` from `foo/*`

```elixir
💣 Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

💣 ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1
```

ABSTRACTION & DSLS

IMPROVING `Kernel` — BANG FUNCTIONS

Abstracted out `foo!/*` from `foo/*`

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}
```

ABSTRACTION & DSLS

IMPROVING Kernel — BANG FUNCTIONS

Abstracted out
`foo!/*` from `foo/*`

```
💣 Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

💣 ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

🆗 value ~> (&(&1 + 1))
#=> 2

⏩ error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

💣 error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

ABSTRACTION & DSLS

IMPROVING `Kernel` — BANG FUNCTIONS

Abstracted out
`foo!/*` from `foo/*`

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

- Works everywhere
- Any data
- Any error struct
- Any flow (esp. pipes)
- *Super* easy to test

ABSTRACTION & DSLS

IMPROVING `Kernel` — BANG FUNCTIONS

Abstracted out `foo!/*` from `foo/*`

```
💣 Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

💣 ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

🆗 value ~> (&(&1 + 1))
#=> 2

⏩ error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

💣 error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any error struct

Any flow (esp. pipes)

Super easy to test

BONUS

Disambiguate between `nil` value and actual errors

ABSTRACTION & DSLS

NOTE: METAPHOR

NOTE: METAPHOR

Because it's easier now

ABSTRACTION & DSLS

NOTE: METAPHOR

- **Concept**: Flow-ability is very core to Elixir’s ethos
  - `Kernel.|>/2`

Because it’s easier now

ABSTRACTION & DSLS

NOTE: METAPHOR

- **Concept**: Flow-ability is very core to Elixir's ethos
  - `Kernel.|>/2`
- Consistent flow metaphor / punning on existing metaphor
  - `Exceptional:` `~>/2` and `>>>/2`

Because it's easier now

NOTE: METAPHOR

- **Concept**: Flow-ability is very core to Elixir’s ethos
  - `Kernel.|>/2`
- Consistent flow metaphor / punning on existing metaphor
  - `Exceptional:` `~>/2` and `>>>/2`

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

ABSTRACTION & DSLS

NOTE: METAPHOR

- **Concept**: Flow-ability is very core to Elixir's ethos
  - `Kernel.|>/2`
- Consistent flow metaphor / punning on existing metaphor
  - `Exceptional: ~>/2` and `>>>/2`

```elixir
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

NOTE: METAPHOR

- **Concept**: Flow-ability is very core to Elixir's ethos
  - `Kernel.|>/2`
- Consistent flow metaphor / punning on existing metaphor
  - `Exceptional:` `~>/2` and `>>>/2`

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

ABSTRACTION & DSLS

NOTE: METAPHOR

- **Concept**: Flow-ability is very core to Elixir's ethos
  - `Kernel.|>/2`
- Consistent flow metaphor / punning on existing metaphor
  - `Exceptional:` `~>/2` and `>>>/2`

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

ABSTRACTION & DSLS

WHAT'S GAINED?

ABSTRACTION & DSLS

WHAT'S GAINED?

- Clear
- Composable
- Greater reuse ♻️
- User choice
- Increased testability
  - Simple example: `is_exception?/1`
- Could still add protocol to get even more power

ABSTRACTION

STORYTELLING 📚

ABSTRACTION

STORYTELLING 📚

1. Your code read like a story

2. We even see this in high-level goals of (e.g.) Phoenix

3. Go make some DSLs!

ABSTRACTION

STORYTELLING 📚

1. Your code read like a story

2. We even see this in high-level goals of (e.g.) Phoenix

3. Go make some DSLs!

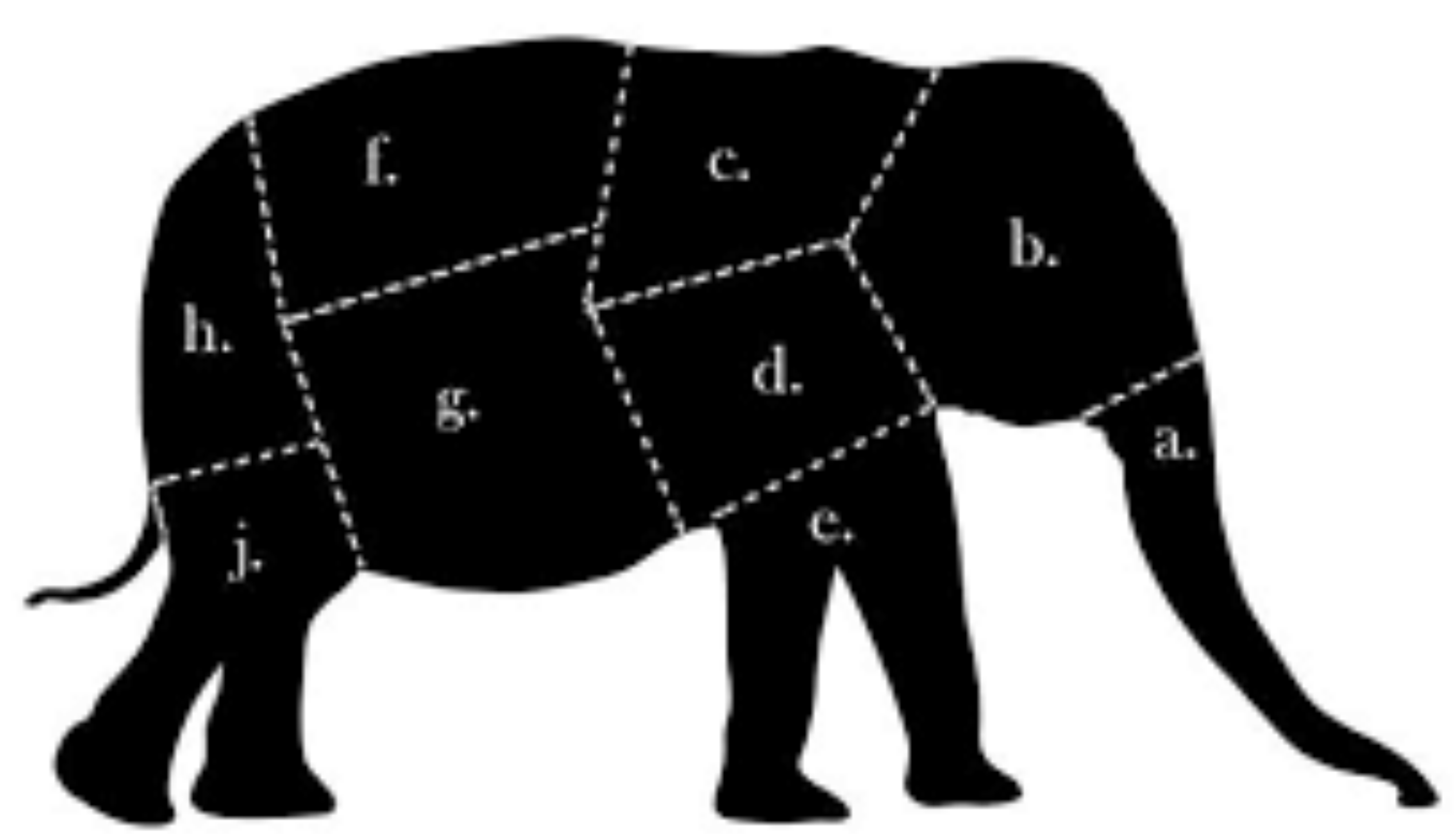
```
conn
|> route()
|> parse()
|> model()
|> view()
|> render()
```

ABSTRACTION & DSLS

HOW TO EAT THE ELEPHANT

ABSTRACTION & DSLS

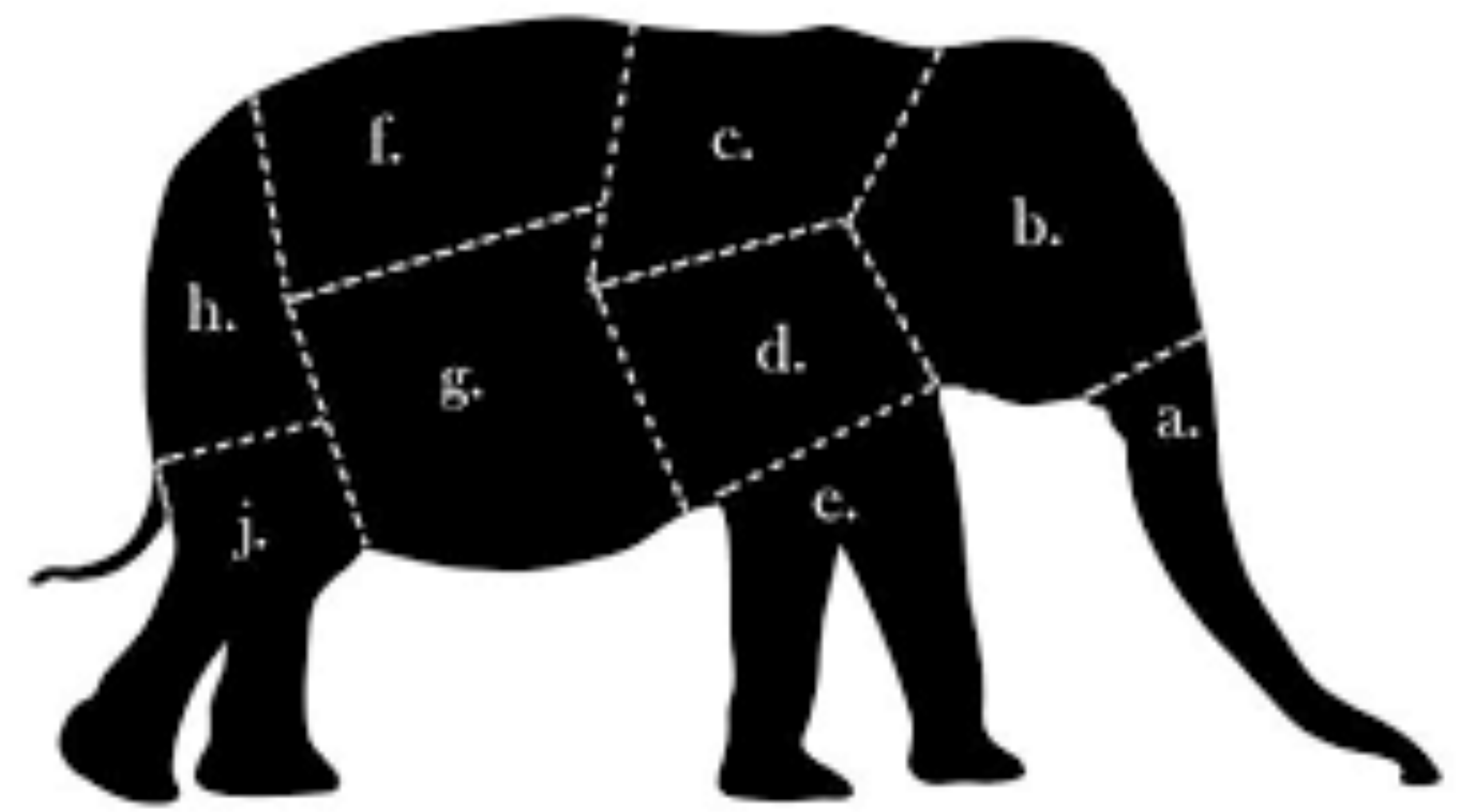
HOW TO EAT THE ELEPHANT



ABSTRACTION & DSLS

HOW TO EAT THE ELEPHANT

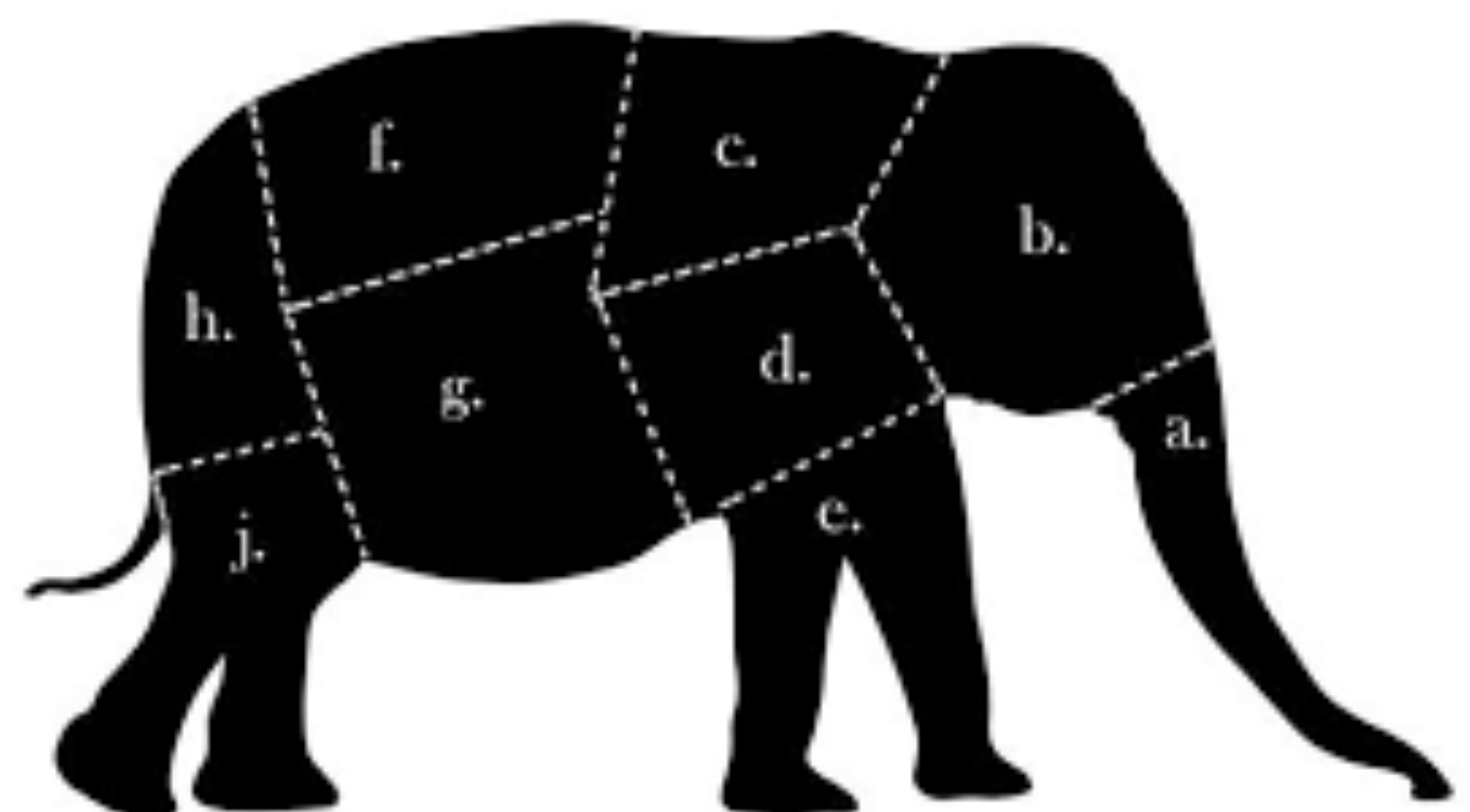
- By feature?



ABSTRACTION & DSLS

HOW TO EAT THE ELEPHANT

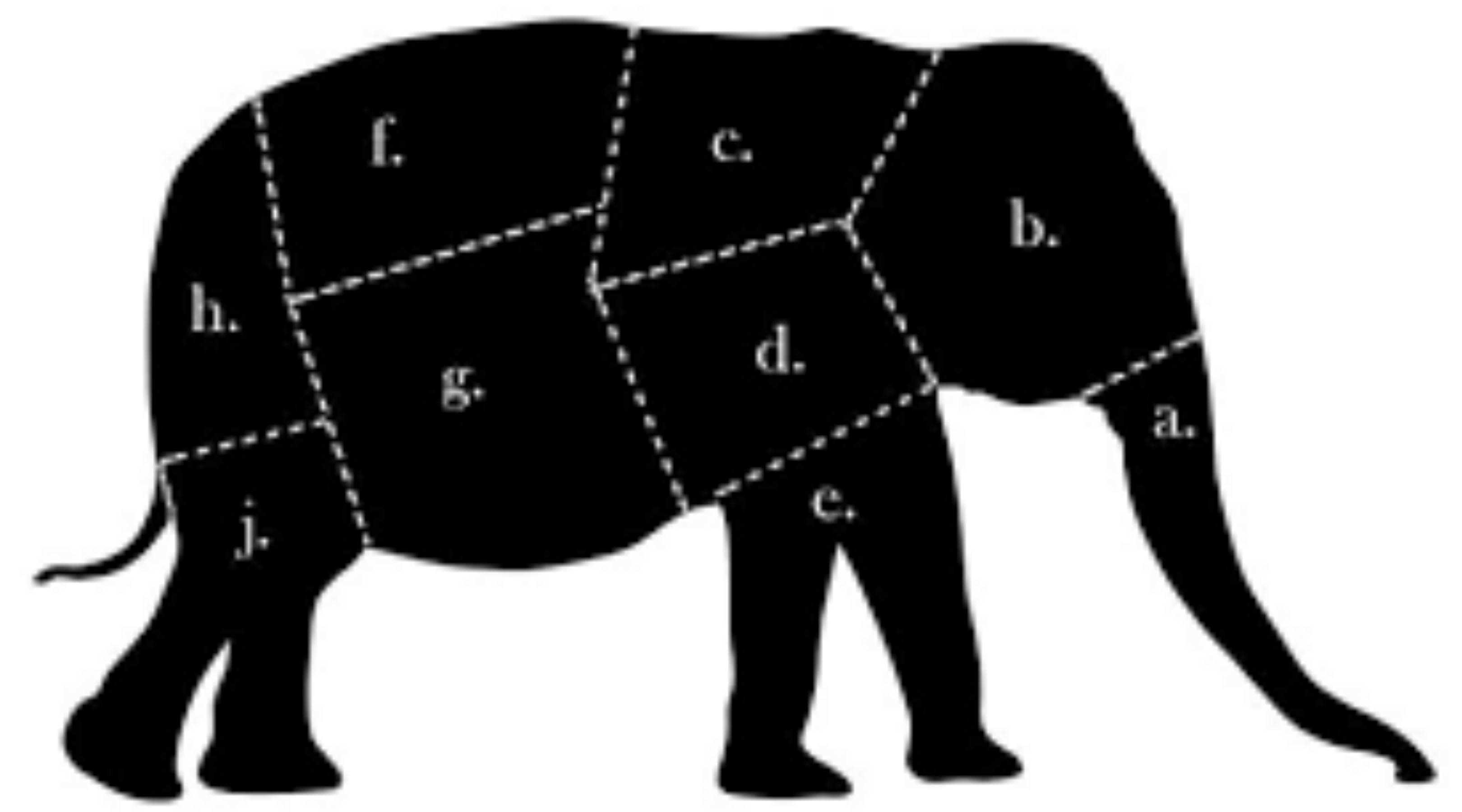
- By feature?
- By behaviour?



ABSTRACTION & DSLS

HOW TO EAT THE ELEPHANT

- By feature?
- By behaviour?
- By structure / properties!



ON STRUCTURE

ON STRUCTURE

STRUCTURE

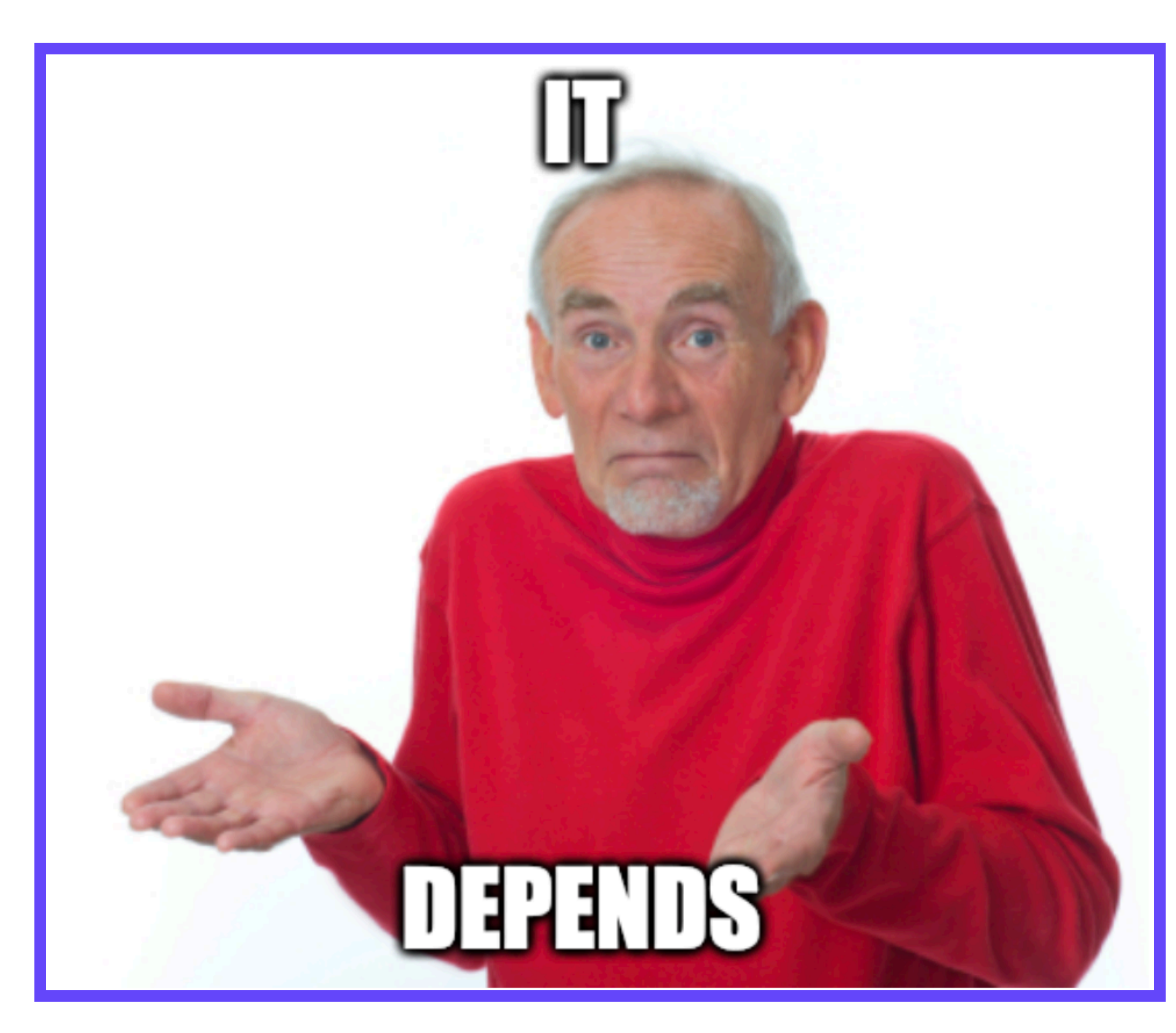
THE ONLY THREE RIGHT ANSWERS

THE ONLY THREE RIGHT ANSWERS

1

STRUCTURE

THE ONLY THREE RIGHT ANSWERS

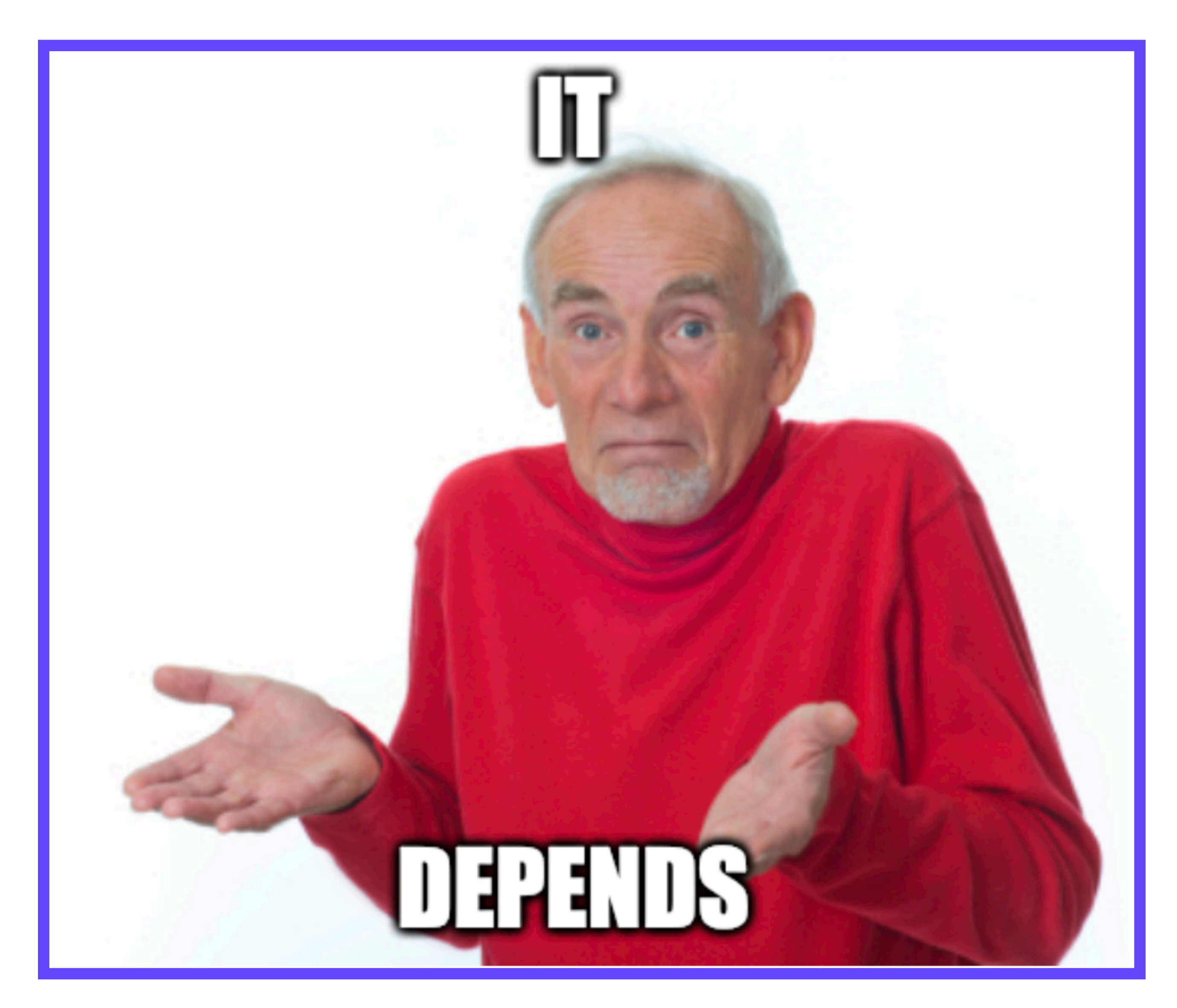


1

2

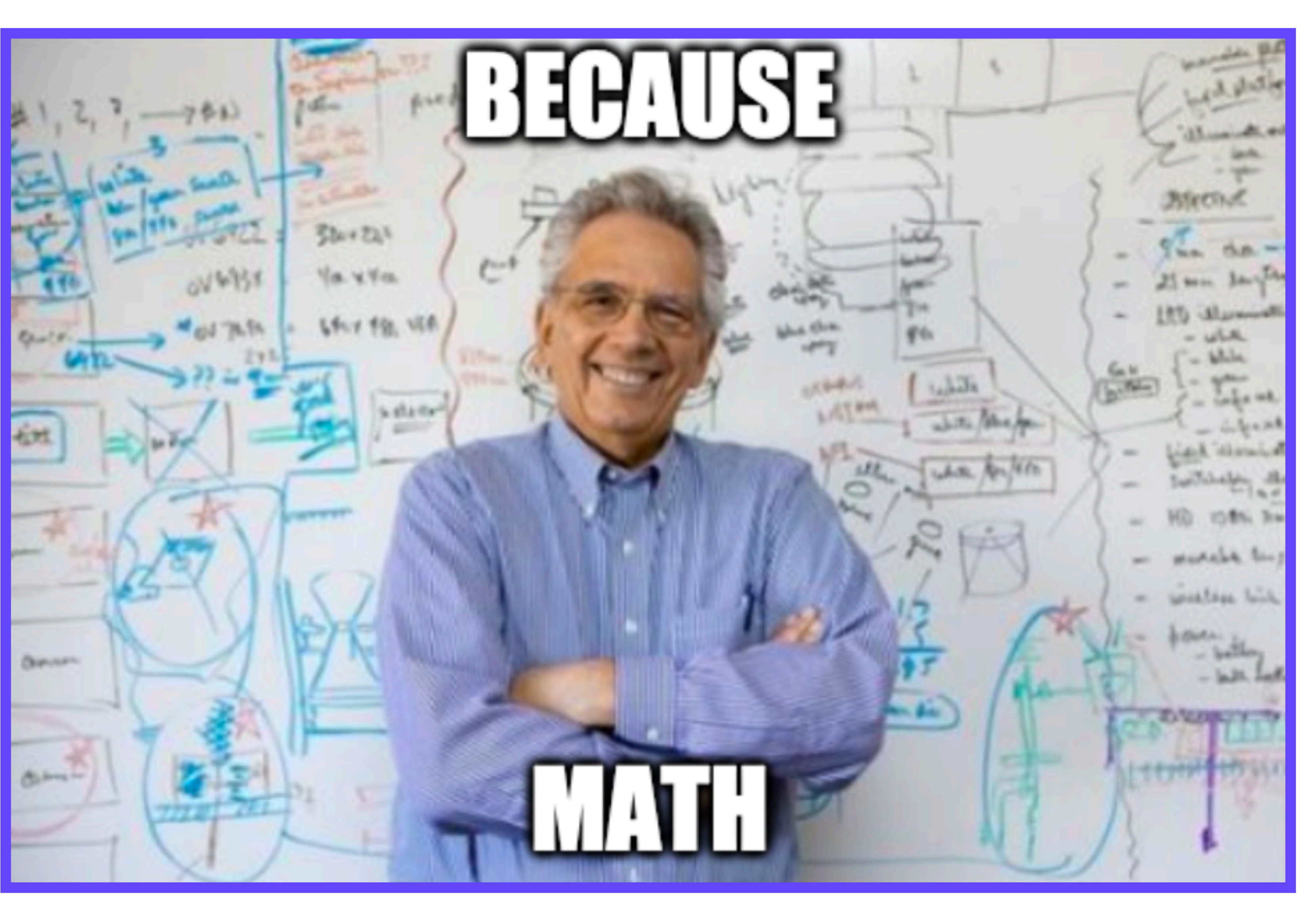
STRUCTURE

THE ONLY THREE RIGHT ANSWERS



1

2



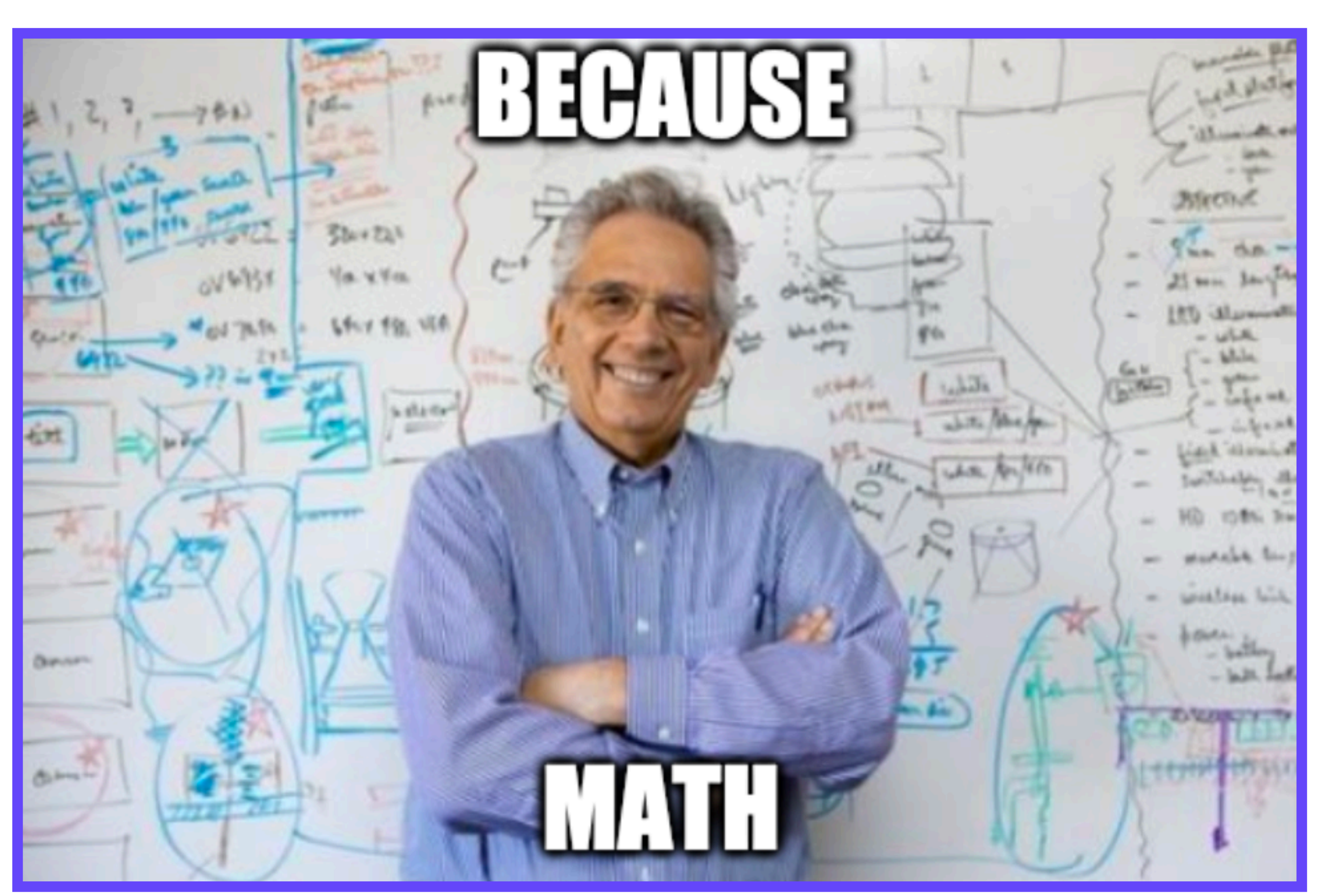
3

STRUCTURE

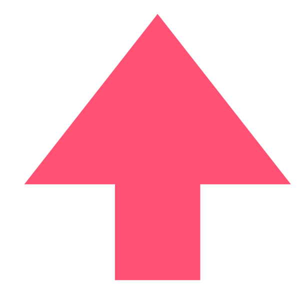
THE ONLY THREE RIGHT ANSWERS

1

2



3



STRUCTURE

ASSOCIATIVITY

STRUCTURE

ASSOCIATIVITY

- Not a data structure

STRUCTURE

ASSOCIATIVITY

- Not a data structure
- Not a function

STRUCTURE

ASSOCIATIVITY

- Not a data structure
- Not a function
- An interface & rules!

STRUCTURE

ASSOCIATIVITY

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

- Not a data structure
- Not a function
- An interface & rules!

STRUCTURE

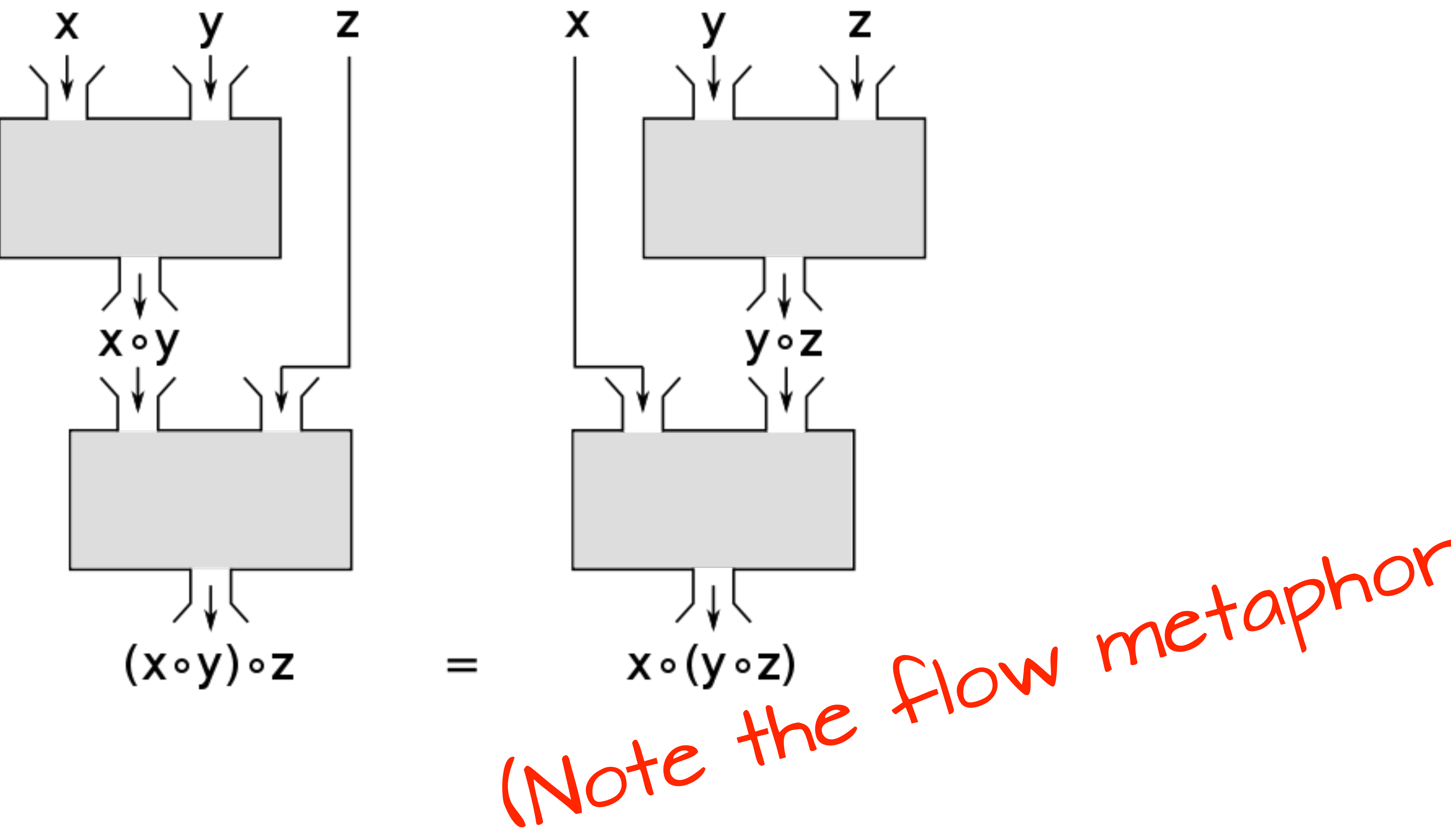
ASSOCIATIVITY

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

- Not a data structure
- Not a function
- An interface & rules!



STRUCTURE

A SEMIGROUP ON…

STRUCTURE

A SEMIGROUP ON…

```elixir
defprotocol Semigroup do
  def concat(a, b)
end

defimpl Semigroup, for: Integer do
  def concat(a, b), do: a + b
end

defimpl Semigroup, for: List do
  def concat(xs, ys), do: xs ++ ys
end
```

STRUCTURE

UNLAWFUL COUNTEREXAMPLE 🚨

```
 1.0 / (2.0  / 3.0) == 1.5
(1.0 /  2.0) / 3.0  == 0.1666...
```

STRUCTURE

HOW TO COMPOSE PROPERTIES

STRUCTURE

HOW TO COMPOSE PROPERTIES

- A structure of structures
  - Keep it in your brain
  - Enforce with `TypeClass`

LET’S DO SOMETHING WILD

LET'S DO SOMETHING WILD

POWER UP

POWER UP

EXPLICIT ASSUMPTIONS

POWER UP

EXPLICIT ASSUMPTIONS

- Parallel pipes

POWER UP

EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order

POWER UP

EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order

POWER UP

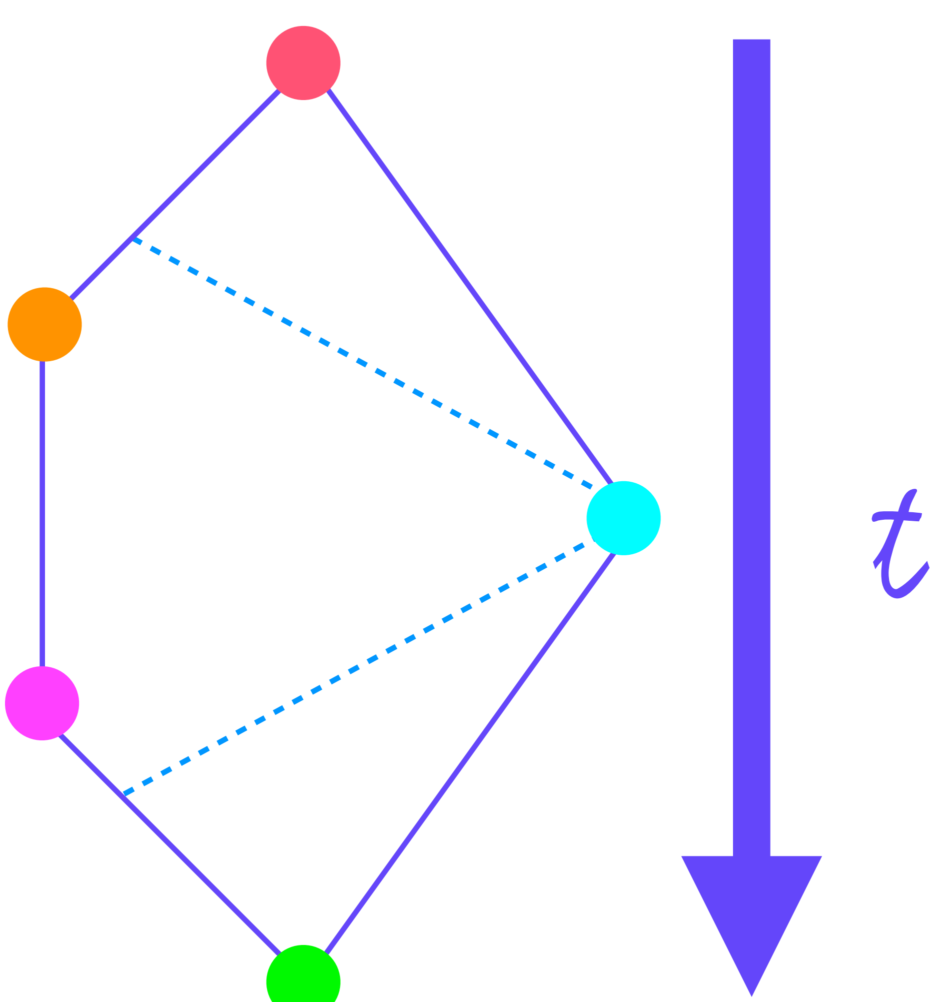
EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order

POWER UP

EXPLICIT ASSUMPTIONS

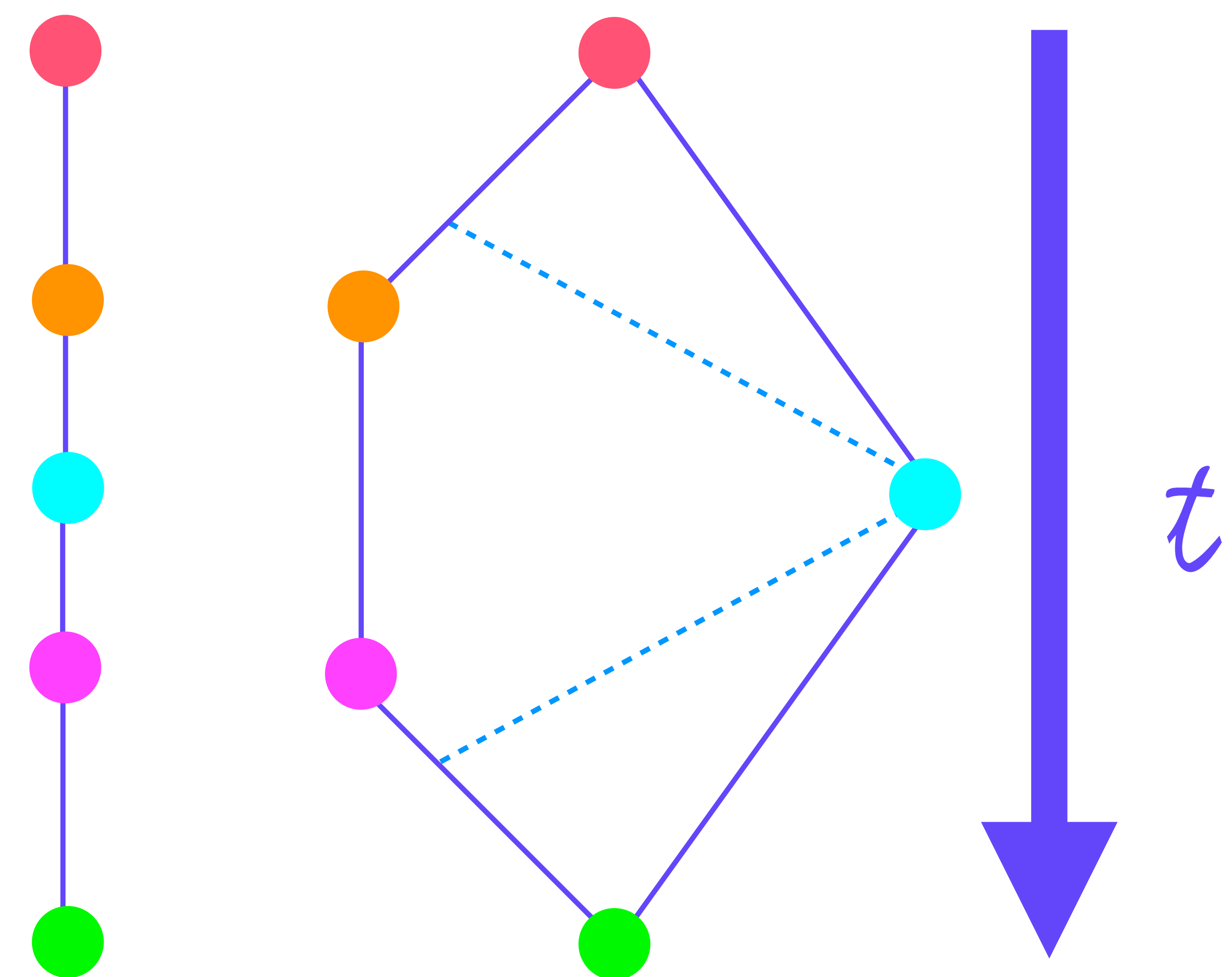
- Parallel pipes
- Concurrency = partial order



POWER UP

EXPLICIT ASSUMPTIONS

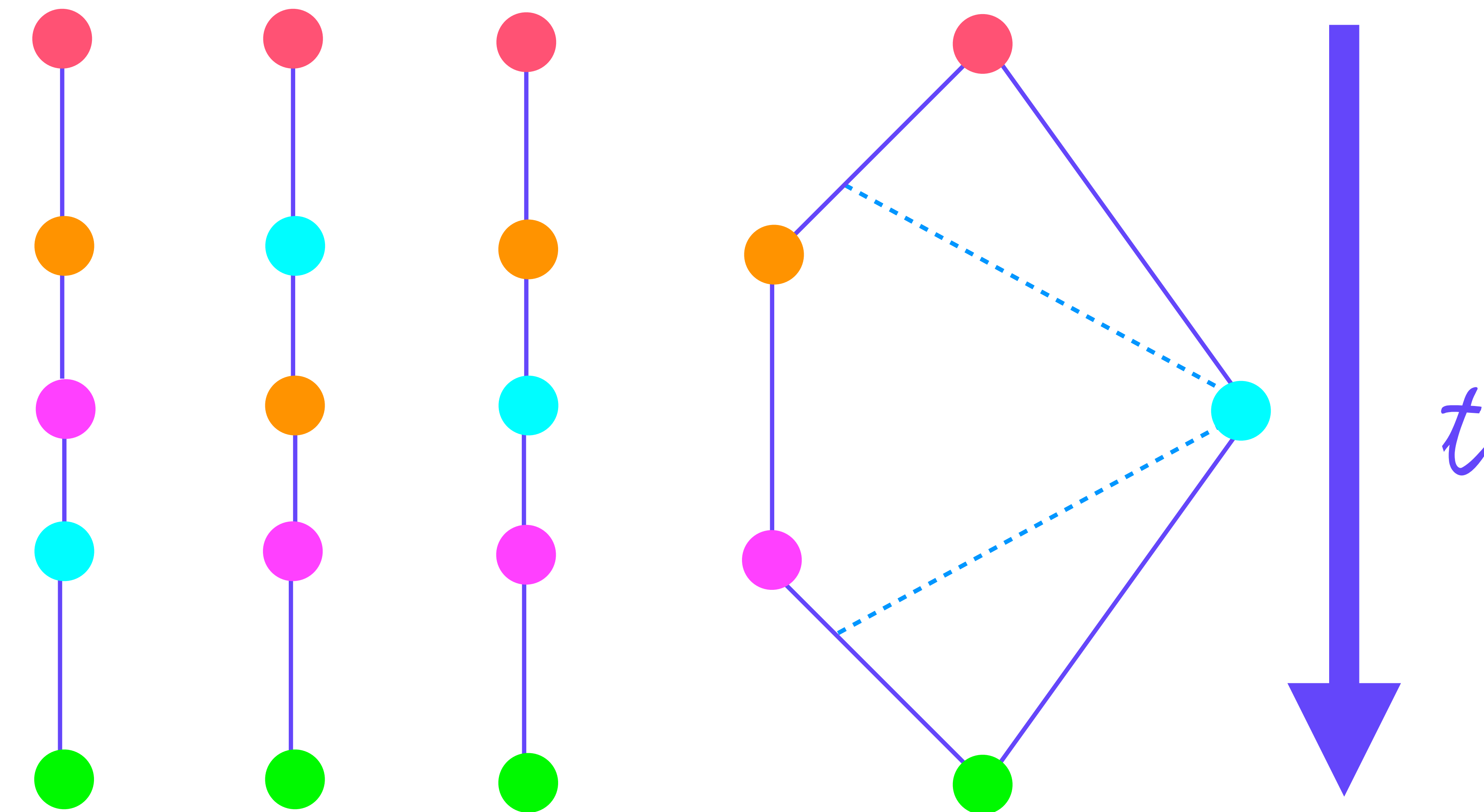
- Parallel pipes
- Concurrency = partial order



POWER UP

EXPLICIT ASSUMPTIONS

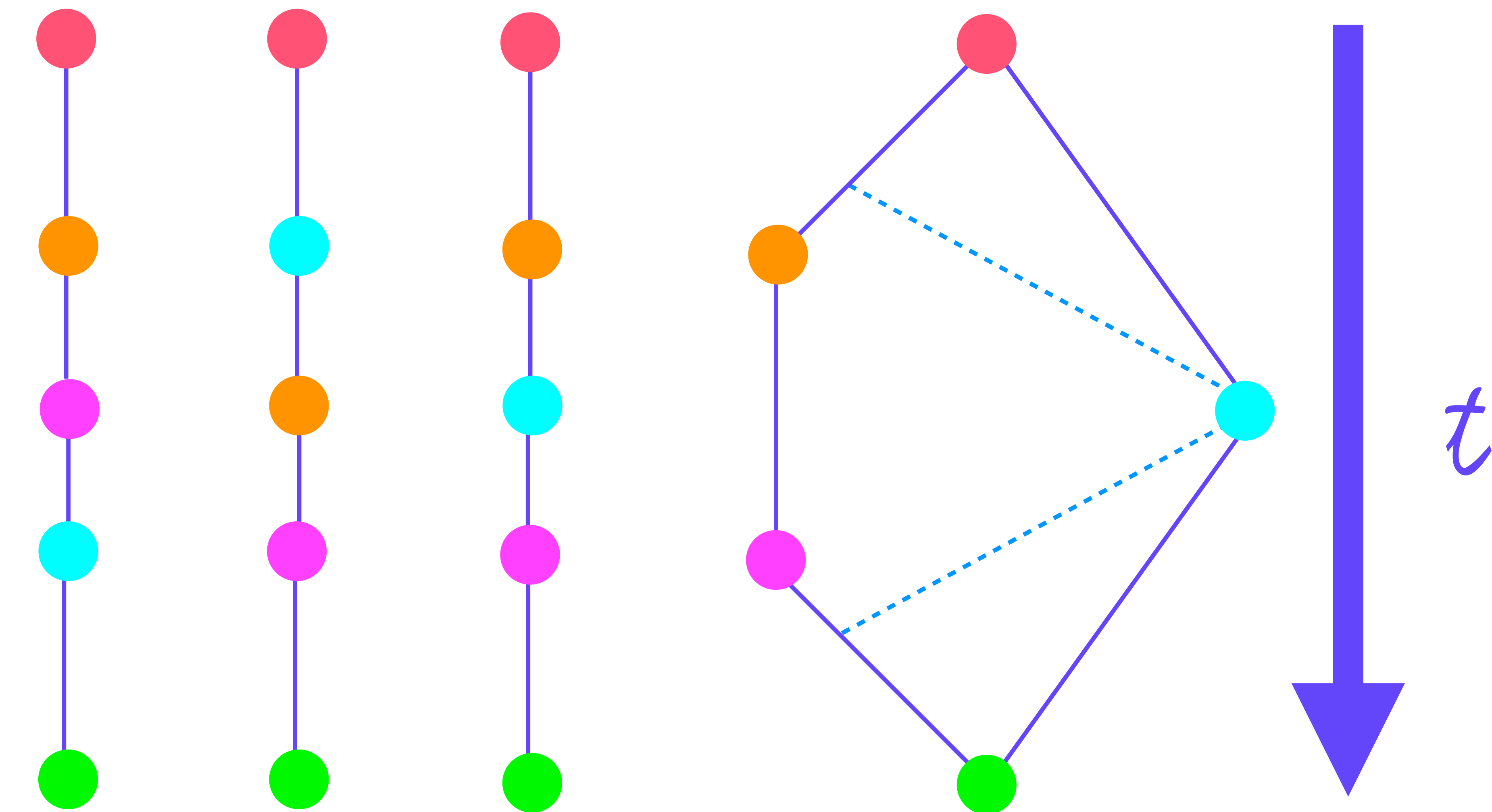
- Parallel pipes
- Concurrency = partial order



POWER UP

EXPLICIT ASSUMPTIONS

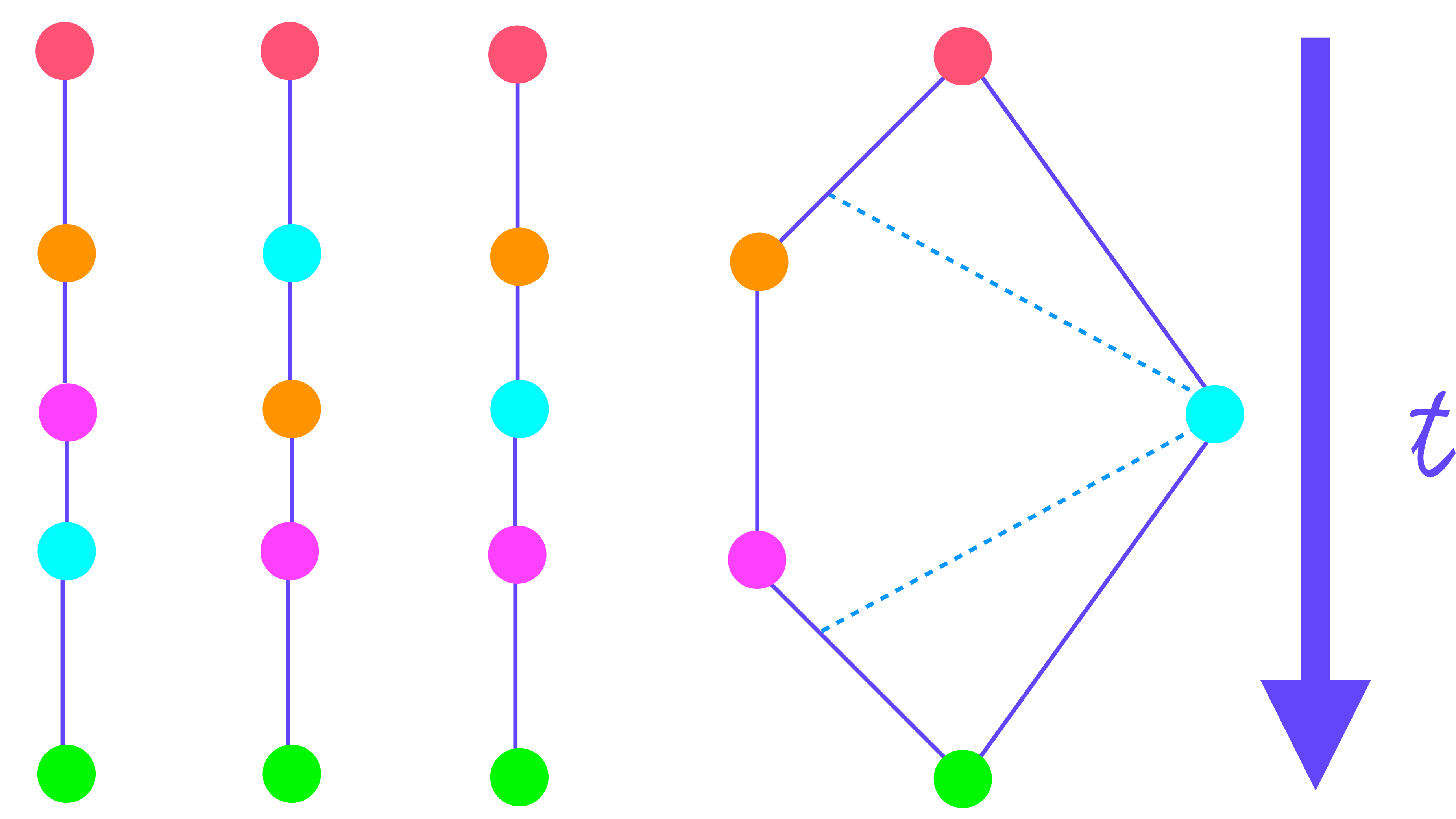
- Parallel pipes
- Concurrency = partial order
- Monotonic
  - All loops must be linearized



POWER UP

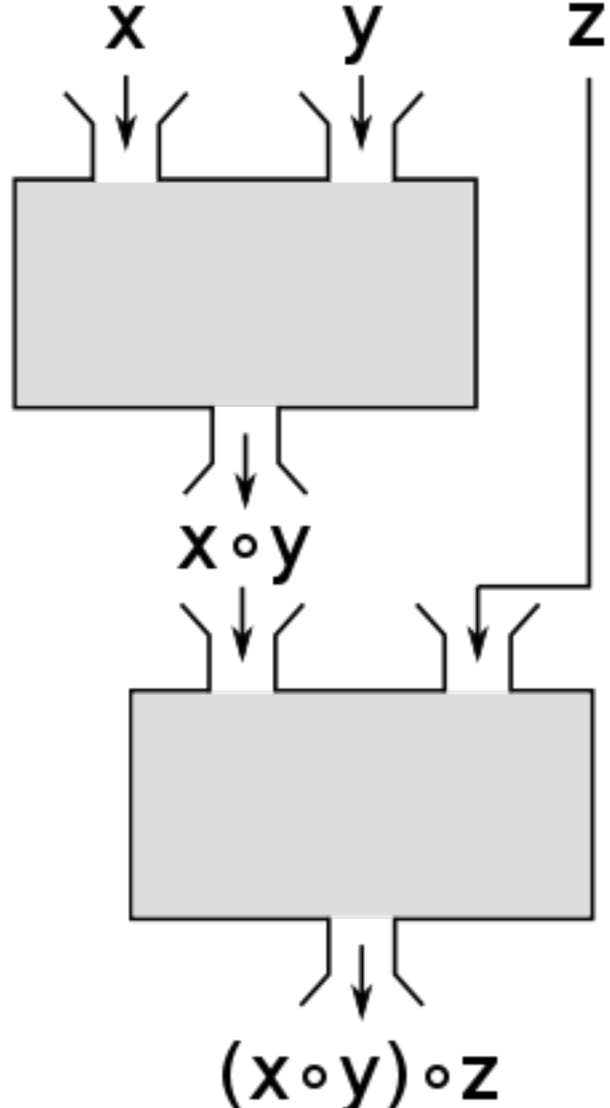
EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order
- Monotonic
  - All loops must be linearized
- Properties
  - Serial composition
  - Parallel composition
  - Explicit evaluation strategy



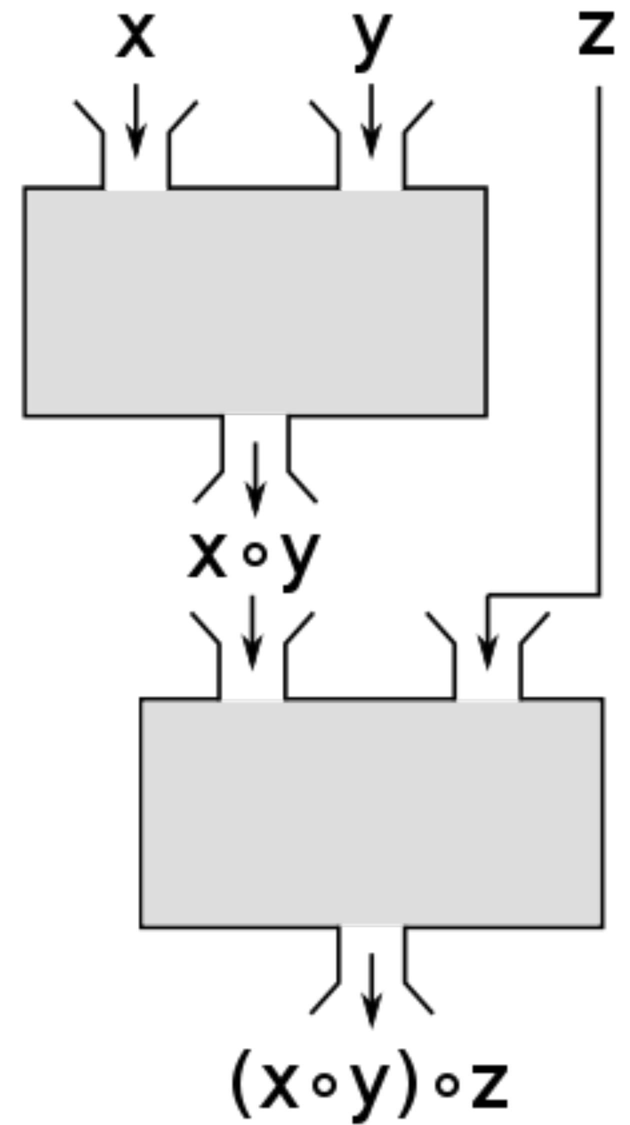
POWER UP

PIPES++



POWER UP

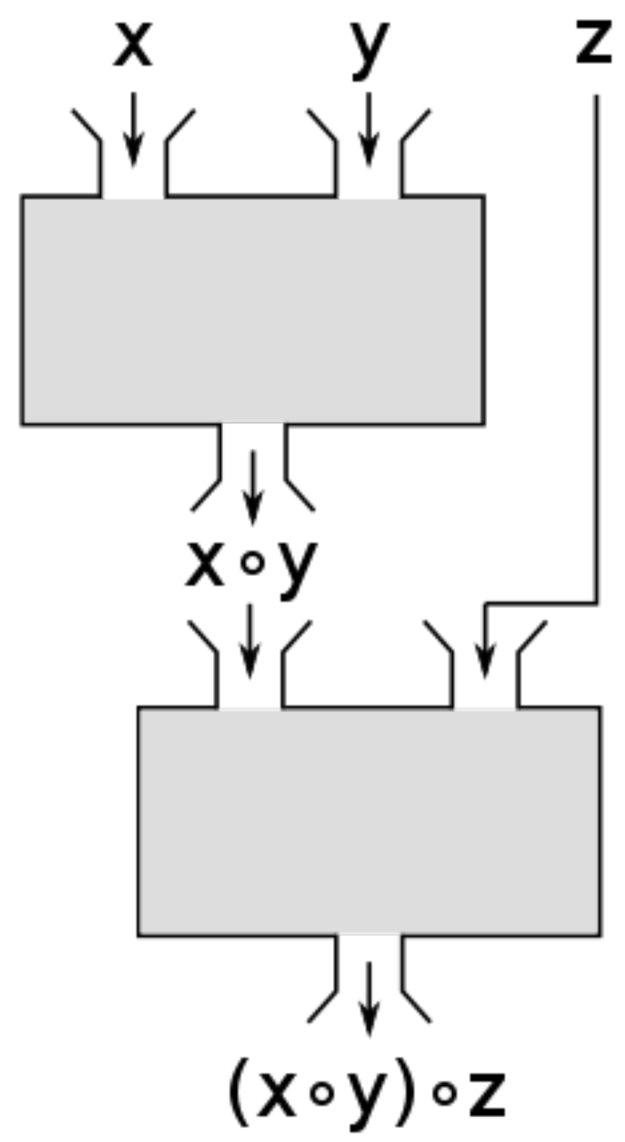
PIPES++




```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end)
  |                               <~> fanout(&inspect/1, fn z -> z end)
  |                               <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
  | <~> unsplit(&String.at(&2, round(&1)) end)
```

POWER UP

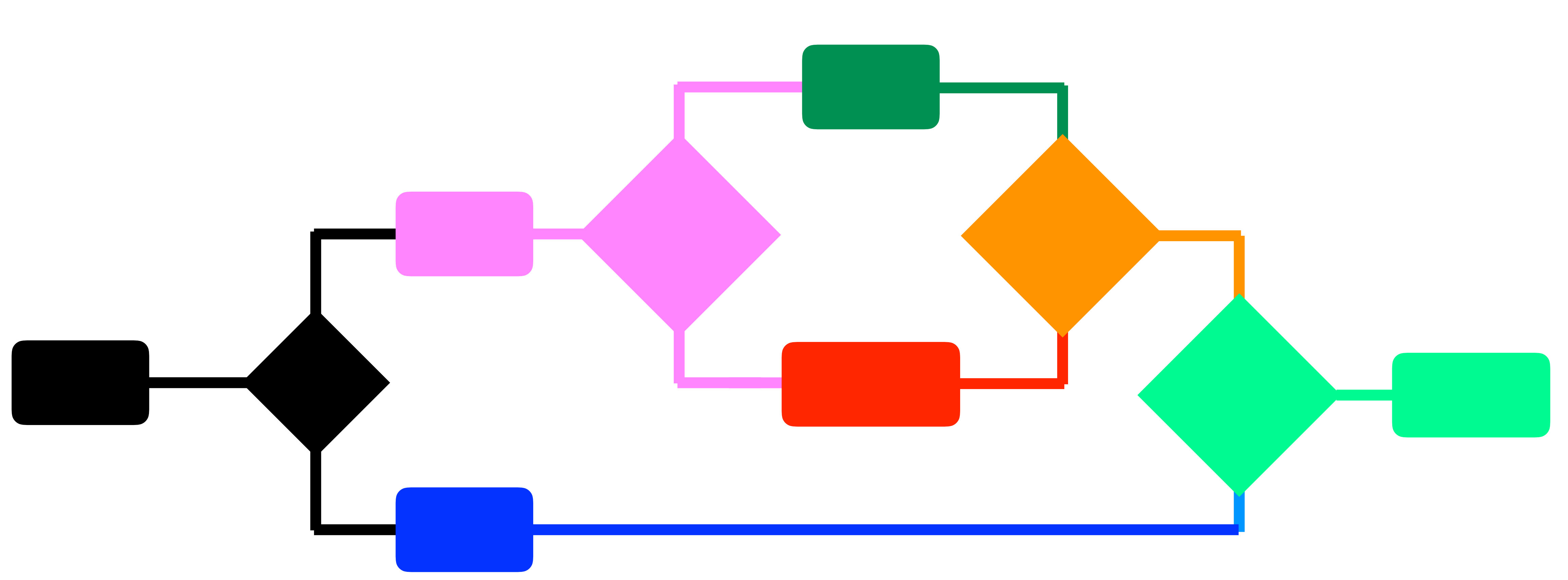
PIPES++

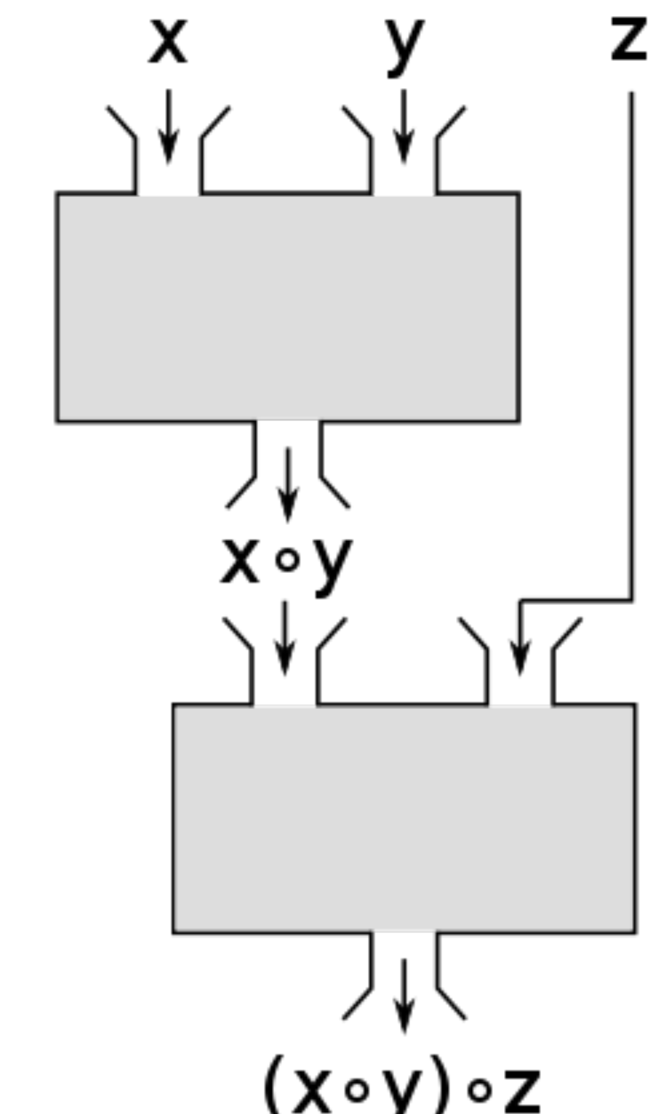



```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
  |                           <~> fanout(&inspect/1, fn z -> z end)
  |                           <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
  | <~> unsplit(&String.at(&2, round(&1)) end)
```

POWER UP

PIPES++






```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
                             <~> fanout(&inspect/1, fn z -> z end)
                             <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
    <~> unsplit(&String.at(&2, round(&1)) end)
```

POWER UP

CLEANUP

```
split do
  fn x -> x / 5 end

  fn y -> y + 1 end |> into(split do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(&String.at(&2, round(&1)))
```

POWER UP

CLEANUP

```
split do
  fn x -> x / 5 end

  fn y -> y + 1 end |> into(split do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(&String.at(&2, round(&1)))
```

POWER UP

CARRIER DATA

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

POWER UP

CARRIER DATA

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

POWER UP

CARRIER DATA

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

POWER UP

CARRIER DATA

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

POWER UP

CARRIER DATA

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

POWER UP

PROTOCOL

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

POWER UP

SIMPLE CASE

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

POWER UP

SIMPLE CASE

```elixir
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```elixir
defimpl Dataflow, for: Any do
  def split(input, path_a, path_b) do
    {path_a(input), path_b(input)}
  end

  def unsplit({a, b}, by: combine), do: combine(a, b)
end
```

POWER UP

SIMPLE CASE

```
split 45 do
  fn x -> x / 5 end

  fn y -> y + 1 end
  |> split(do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(by: fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(by: &String.at(&2, round(&1)))
```

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defimpl Dataflow, for: Any do
  def split(input, path_a, path_b) do
    {path_a(input), path_b(input)}
  end

  def unsplit({a, b}, by: combine), do: combine(a, b)
end
```

POWER UP

ASYNC

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

POWER UP

ASYNC

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defmodule Async do
  defstruct :value

  def asyncify(input) do
    %Async{value: input}
  end

  def syncify(%{value: value}), do: value
end
```

POWER UP

ASYNC

```elixir
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```elixir
defmodule Async do
  defstruct :value

  def asyncify(input) do
    %Async{value: input}
  end

  def syncify(%{value: value}), do: value
end
```

```elixir
defimpl Dataflow, for: Async do
  def split(%{value: input}, path_a, path_b) do
    %Async{
      value: {
        Task.async(path_a(input)),
        Task.async(path_b(input))
      }
    }
  end

  def unsplit({a, b}, by: combine) do
    %Async{value: combine(Task.await(a), Task.await(b))}
  end
end
```

POWER UP

ASYNC

```
45
|> asyncify()
|> split do
  fn x -> x / 5 end

  fn y -> y + 1 end |> split(do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(&String.at(&2, round(&1)))
|> syncify()
```

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defmodule Async do
  defstruct :value

  def asyncify(input) do
    %Async{value: input}
  end

  def syncify(%{value: value}), do: value
end
```

```
defimpl Dataflow, for: Async do
  def split(%{value: input}, path_a, path_b) do
    %Async{
      value: {
        Task.async(path_a(input)),
        Task.async(path_b(input))
      }
    }
  end

  def unsplit({a, b}, by: combine) do
    %Async{value: combine(Task.await(a), Task.await(b))}
  end
end
```

POWER UP

UPSHOT

POWER UP

UPSHOT

- Higher *semantic density* (focused on meaning not mechanics)

POWER UP

UPSHOT

- Higher *semantic density* (focused on meaning not mechanics)
- Declarative, **configurable** data flow 🤯

POWER UP

UPSHOT

- Higher *semantic density* (focused on meaning not mechanics)
- Declarative, **configurable** data flow 🤯
- Extremely extensible
  - `defimpl Dataflow, for: %Stream{}`
  - `defimpl Dataflow, for: %Distributed{}`
  - `defimpl Dataflow, for: %Broadway{}`

POWER UP

UPSHOT

- Higher *semantic density* (focused on meaning not mechanics)
- Declarative, **configurable** data flow 🤯
- Extremely extensible
  - `defimpl Dataflow, for: %Stream{}`
  - `defimpl Dataflow, for: %Distributed{}`
  - `defimpl Dataflow, for: %Broadway{}`
- Model-testable

UPSHOT

- Higher *semantic density* (focused on meaning not mechanics)
- Declarative, **configurable** data flow 🤯
- Extremely extensible
  - `defimpl Dataflow, for: %Stream{}`
  - `defimpl Dataflow, for: %Distributed{}`
  - `defimpl Dataflow, for: %Broadway{}`
- Model-testable
- Composable with other pipes and *change evaluation strategies*

A CALL FOR LIBRARIES

A CALL FOR LIBRARIES

📣

A CALL FOR LIBRARIES

SUMMARY

SUMMARY

- Can plug into / extend

A CALL FOR LIBRARIES

SUMMARY

- Can plug into / extend
- Single-threaded context

A CALL FOR LIBRARIES

SUMMARY

- Can plug into / extend
- Single-threaded context
- Distributed context

A CALL FOR LIBRARIES

SUMMARY

- Can plug into / extend
- Single-threaded context
- Distributed context
- Dynamic hybrid contexts

A CALL FOR LIBRARIES

EXTEND RAILROAD PROGRAMMING

```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```

A CALL FOR LIBRARIES

EXTEND RAILROAD PROGRAMMING

```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```

Happy Path (Continue)

Error Case (Skip)

No Effect (Afterwards)

A CALL FOR LIBRARIES

EXTEND RAILROAD PROGRAMMING

```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```

Happy Path (Continue)

Error Case (Skip)

No Effect (Afterwards)

A CALL FOR LIBRARIES

EXTEND RAILROAD PROGRAMMING

```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```

Happy Path (Continue)

Error Case (Skip)

No Effect (Afterwards)

A CALL FOR LIBRARIES

SURPRISING NUMBER OF FACTORS

```
def foo(val) do
  Task.async(fn ->
    IO.inspect(val)

    bar = fn (inner_val) ->
      Task.async(fn ->
        IO.inspect(inner_val)
      end)
    end

    bar(val + 2)
  end)
end

foo(42)
```

A CALL FOR LIBRARIES

SURPRISING NUMBER OF FACTORS

Log

Program

```
def foo(val) do
  Task.async(fn ->
    IO.inspect(val)

    bar = fn (inner_val) ->
      Task.async(fn ->
        IO.inspect(inner_val)
      end)
    end

    bar(val + 2)
  end)
end

foo(42)
```

A CALL FOR LIBRARIES

SURPRISING NUMBER OF FACTORS

```
def foo(val) do
  Task.async(fn ->
    IO.inspect(val)

    bar = fn (inner_val) ->
      Task.async(fn ->
        IO.inspect(inner_val)
      end)
    end

    bar(val + 2)
  end)
end

foo(42)
```

Log

SUMMARY

SUMMARY

KEEP IN MIND

SUMMARY

KEEP IN MIND

1. Protocols-for-DDD (P4D3)
2. Add a semantic layer
3. How do you locally test your distributed system? Look at the properties!
4. Under which conditions does your code work? What are your assumptions?
5. Prop testing is useful for structured abstractions
6. You should be able to code half-asleep

https://fission.codes
https://talk.fission.codes
https://tools.fission.codes

BEDANKT, AMSTERDAM

brooklyn@fission.codes
github.com/expede
@expede

REWRITING A PHOENIX APP

REWRITING A PHOENIX APP
🕚 ANTI-PATTERNS & BUYING TIME 🔥

REWRITING A PHOENIX APP

THE PROBLEM

REWRITING A PHOENIX APP

THE PROBLEM

- Inherited project
    - First go at Phoenix
    - “We don’t believe in tests”

THE PROBLEM

- Inherited project
  - First go at Phoenix
  - “We don’t believe in tests”
- Very brittle, many bugs 😱

REWRITING A PHOENIX APP

THE PROBLEM

- Inherited project
    - First go at Phoenix
    - “We don’t believe in tests”
- Very brittle, many bugs 😱
- Now tight deadline — no time to do a total rewrite
    - …or was there? 😏

REWRITING A PHOENIX APP

STRATEGY: GENERALIZATION!

REWRITING A PHOENIX APP

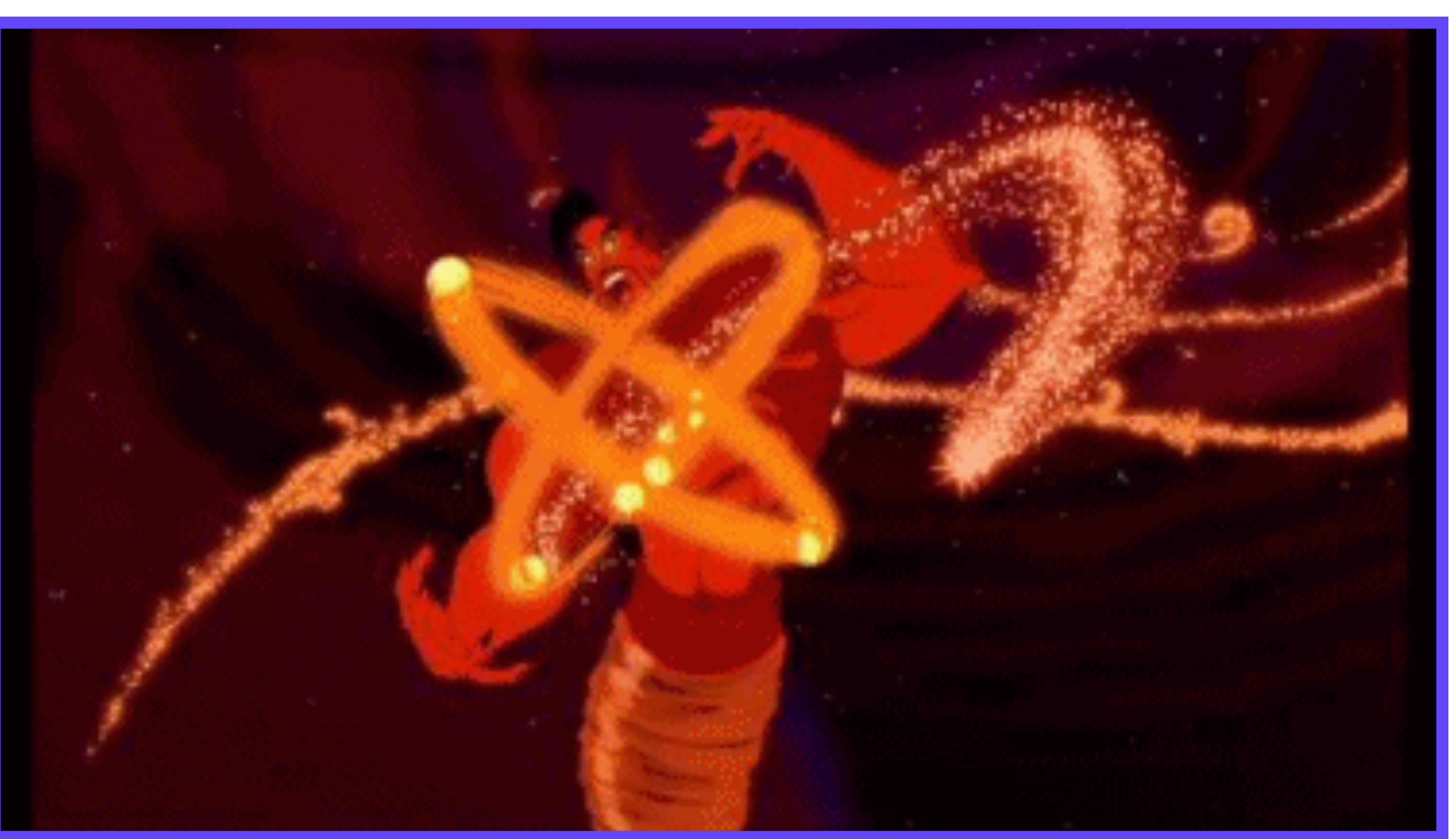
STRATEGY: GENERALIZATION!

- Deescalate
- Running code
- Get to “net new” features
- Two things that people don’t generally associate with abstraction!

REWRITING A PHOENIX APP
⚠️ A WORD OF WARNING ⚠️

A WORD OF WARNING

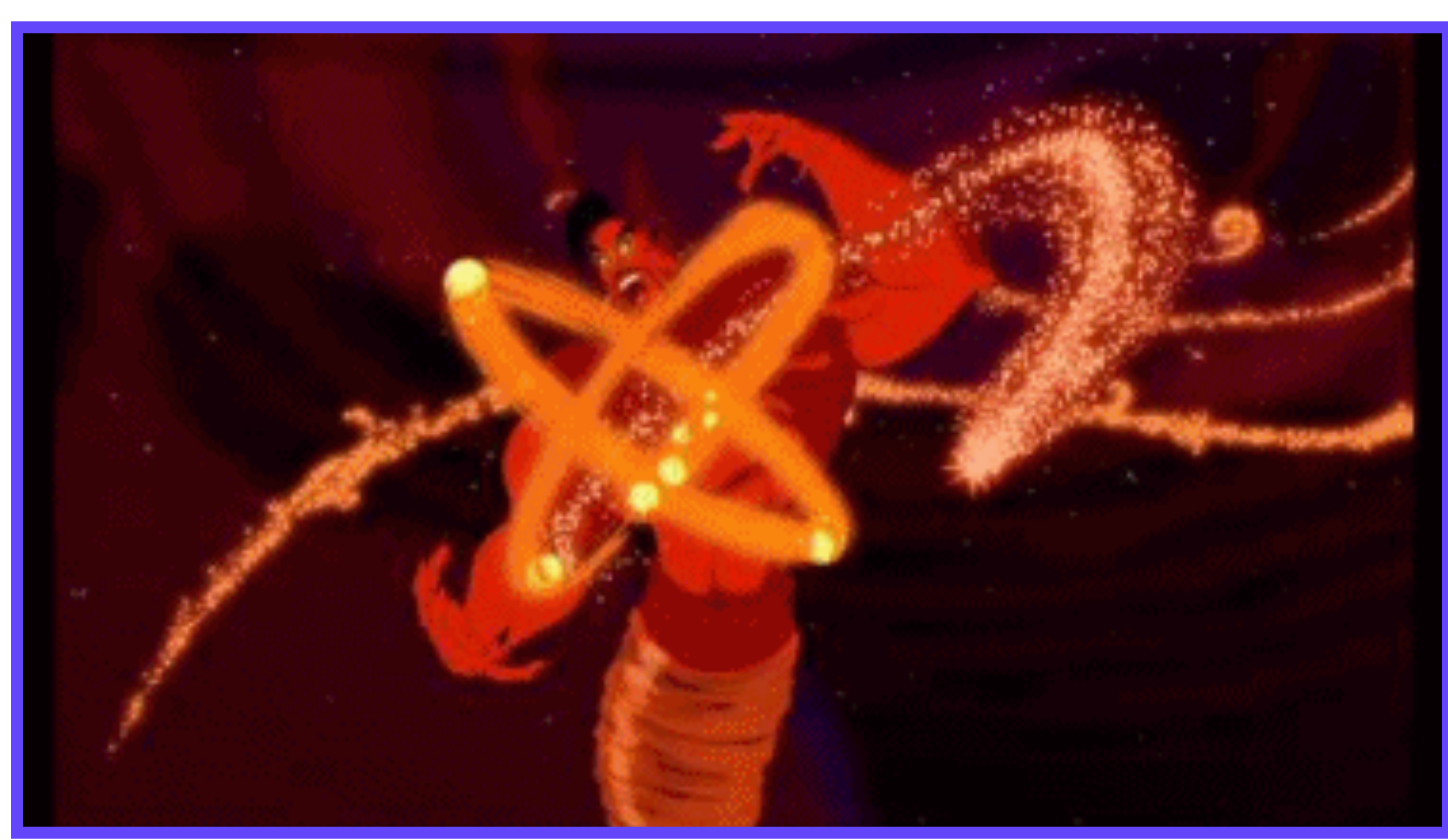
A WORD OF WARNING



How you see yourself

REWRITING A PHOENIX APP

A WORD OF WARNING ⚠️



How you see yourself

How others see you

REWRITING A PHOENIX APP

THE (TEMPORARY) SOLUTION

REWRITING A PHOENIX APP

THE (TEMPORARY) SOLUTION

```
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
          {:ok, user} ->
            conn
            |> put_flash(:info, "Updated user")
            |> redirect(to: user_path(conn, :show, user))

          {:error, %Ecto.Changeset{} = changeset} ->
            render(conn, "edit.html")
        end
    end
  end
end
```

REWRITING A PHOENIX APP

THE (TEMPORARY) SOLUTION

```elixir
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
          {:ok, user} ->
            conn
            |> put_flash(:info, "Updated user")
            |> redirect(to: user_path(conn, :show, user))

          {:error, %Ecto.Changeset{} = changeset} ->
            render(conn, "edit.html")
      end
    end
  end
end
```

REWRITING A PHOENIX APP

THE (TEMPORARY) SOLUTION

```
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
           {:ok, user} ->
             conn
             |> put_flash(:info, "Updated user")
             |> redirect(to: user_path(conn, :show, user))

           {:error, %Ecto.Changeset{} = changeset} ->
             render(conn, "edit.html")
         end
    end
  end
end
```

REWRITING A PHOENIX APP

THE (TEMPORARY) SOLUTION

```
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
           {:ok, user} ->
             conn
             |> put_flash(:info, "Updated user")
             |> redirect(to: user_path(conn, :show, user))

           {:error, %Ecto.Changeset{} = changeset} ->
             render(conn, "edit.html")
         end
    end
  end
end
```

- Generate all of MVVC
- Compile-time functions
  - Incl. validation
- Spend 90% time on macro
  - Write
  - Test!

REWRITING A PHOENIX APP

TRADEOFFS

```
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
           {:ok, user} ->
             conn
             |> put_flash(:info, "Updated user")
             |> redirect(to: user_path(conn, :show, user))

           {:error, %Ecto.Changeset{} = changeset} ->
             render(conn, "edit.html")
         end
    end
  end
end
```

REWRITING A PHOENIX APP

TRADEOFFS

```elixir
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
           {:ok, user} ->
             conn
             |> put_flash(:info, "Updated user")
             |> redirect(to: user_path(conn, :show, user))

           {:error, %Ecto.Changeset{} = changeset} ->
             render(conn, "edit.html")
         end
    end
  end
end
```

- Implement *very fast*
- Flexible
- Makes a lot of assumptions
- Macro magic
  - Difficult to read, extend, &c
  - Low-level implementation
    - The macro swamp

REWRITING A PHOENIX APP

TRADEOFFS

```elixir
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
           {:ok, user} ->
             conn
             |> put_flash(:info, "Updated user")
             |> redirect(to: user_path(conn, :show, user))

           {:error, %Ecto.Changeset{} = changeset} ->
             render(conn, "edit.html")
         end
    end
  end
end
```

REWRITING A PHOENIX APP

TRADEOFFS

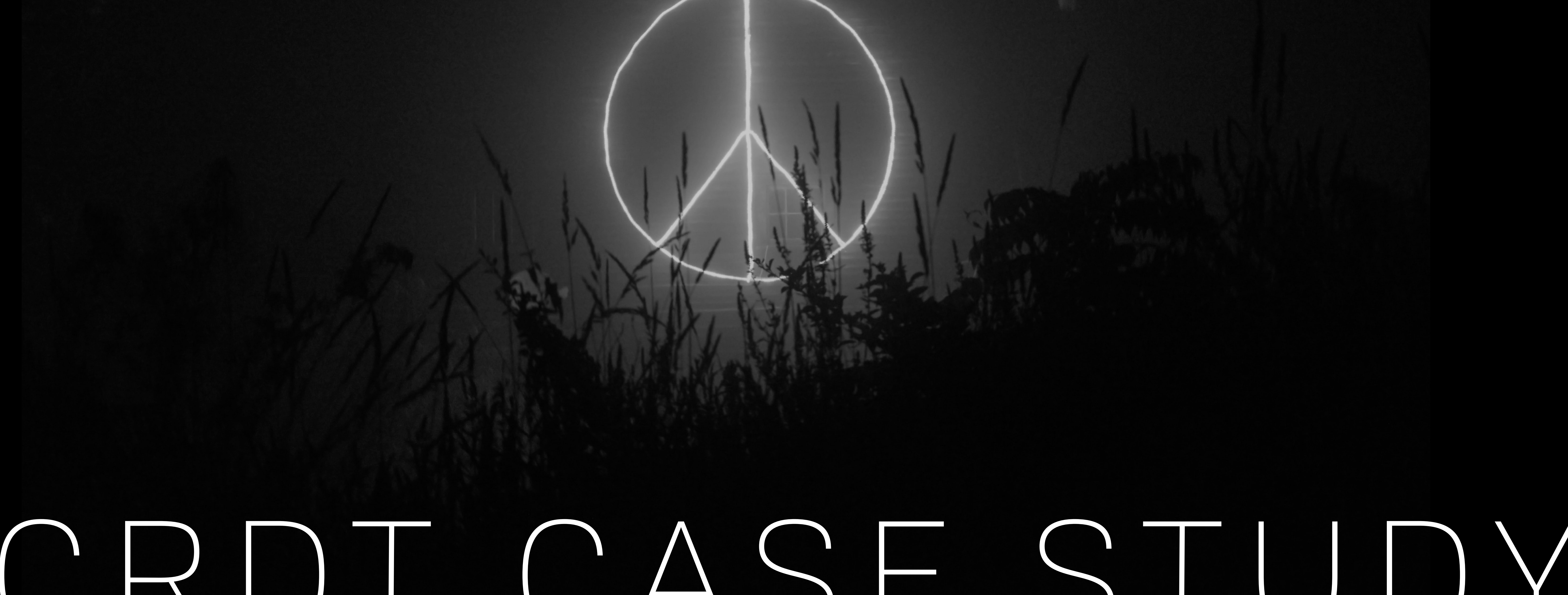
```
defmodule WebApp.Vertical.User do
  use Vertical.Slice

  defvertical :users, excluding: [:delete] do
    def update(conn, _params) do
      conn
      |> User.changeset(room_params)
      |> User.Repo.update()
      |> render("update.html")
      |> case do
           {:ok, user} ->
             conn
             |> put_flash(:info, "Updated user")
             |> redirect(to: user_path(conn, :show, user))

           {:error, %Ecto.Changeset{} = changeset} ->
             render(conn, "edit.html")
         end
    end
  end
end
```

- Straightforward: MVVC!
- Encode patterns directly in code 🎉

CRDT CASE STUDY



CRDT CASE STUDY

⏲️ ELEGANT AND USEFUL ☮️

CRDT CASE STUDY

SCENARIO

CRDT CASE STUDY

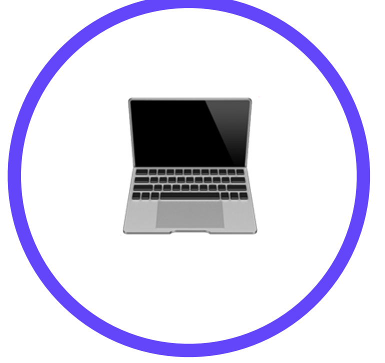
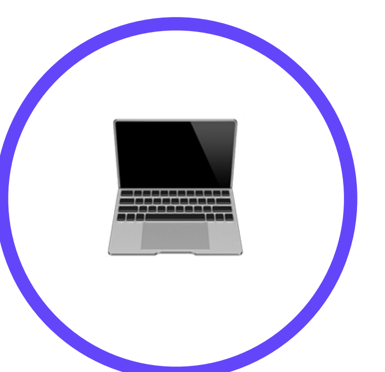
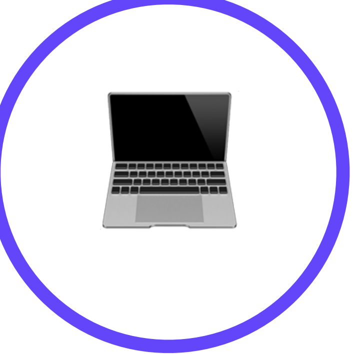
SCENARIO

- Distributed / uncontrolled topology

CRDT CASE STUDY

SCENARIO

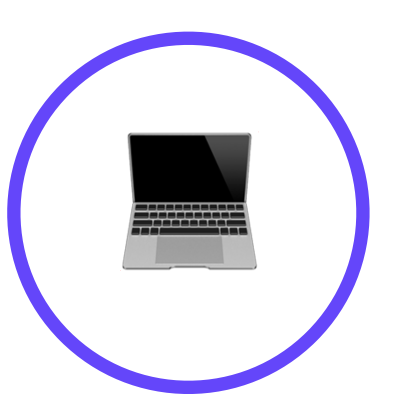
- Distributed / uncontrolled topology

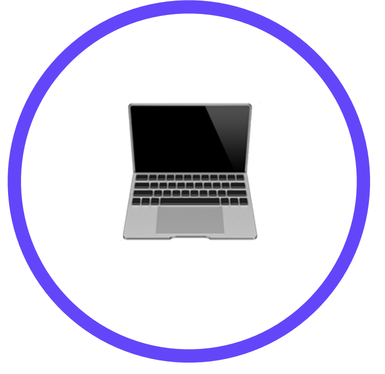
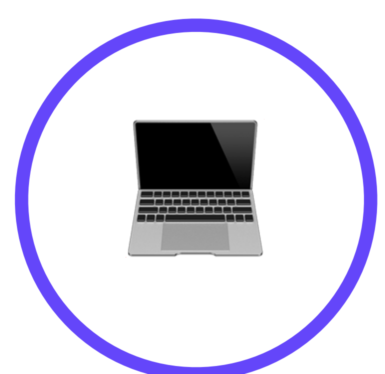
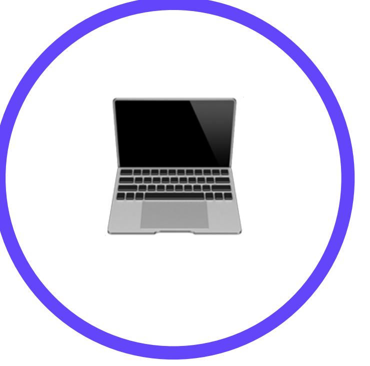




CRDT CASE STUDY

SCENARIO

- Distributed / uncontrolled topology
- Eventual consistency

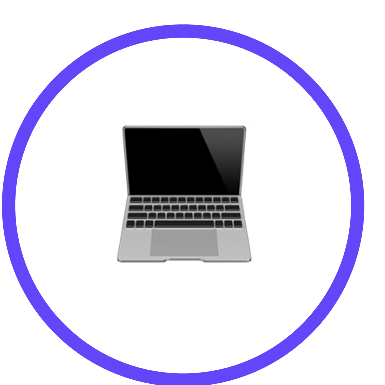
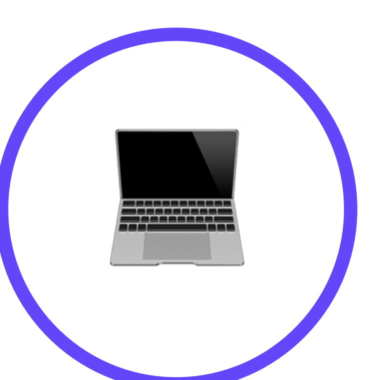
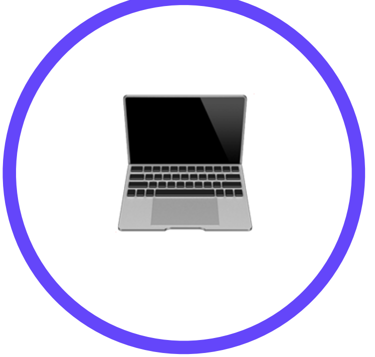
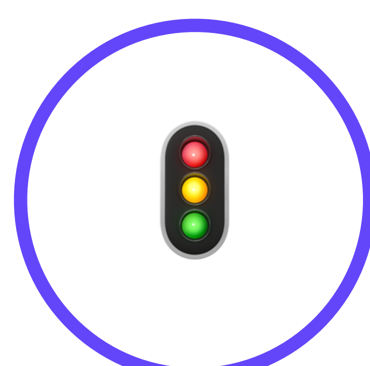
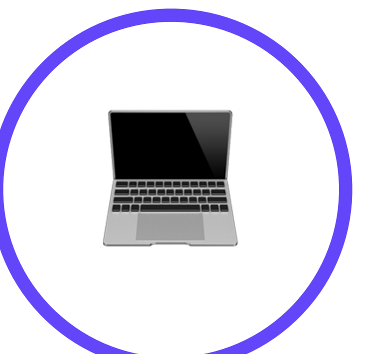
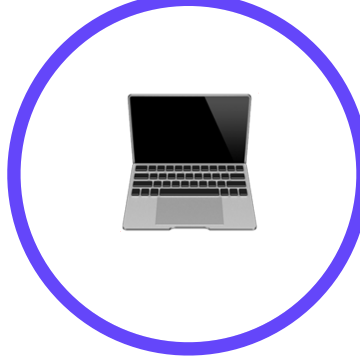


CRDT CASE STUDY

SCENARIO

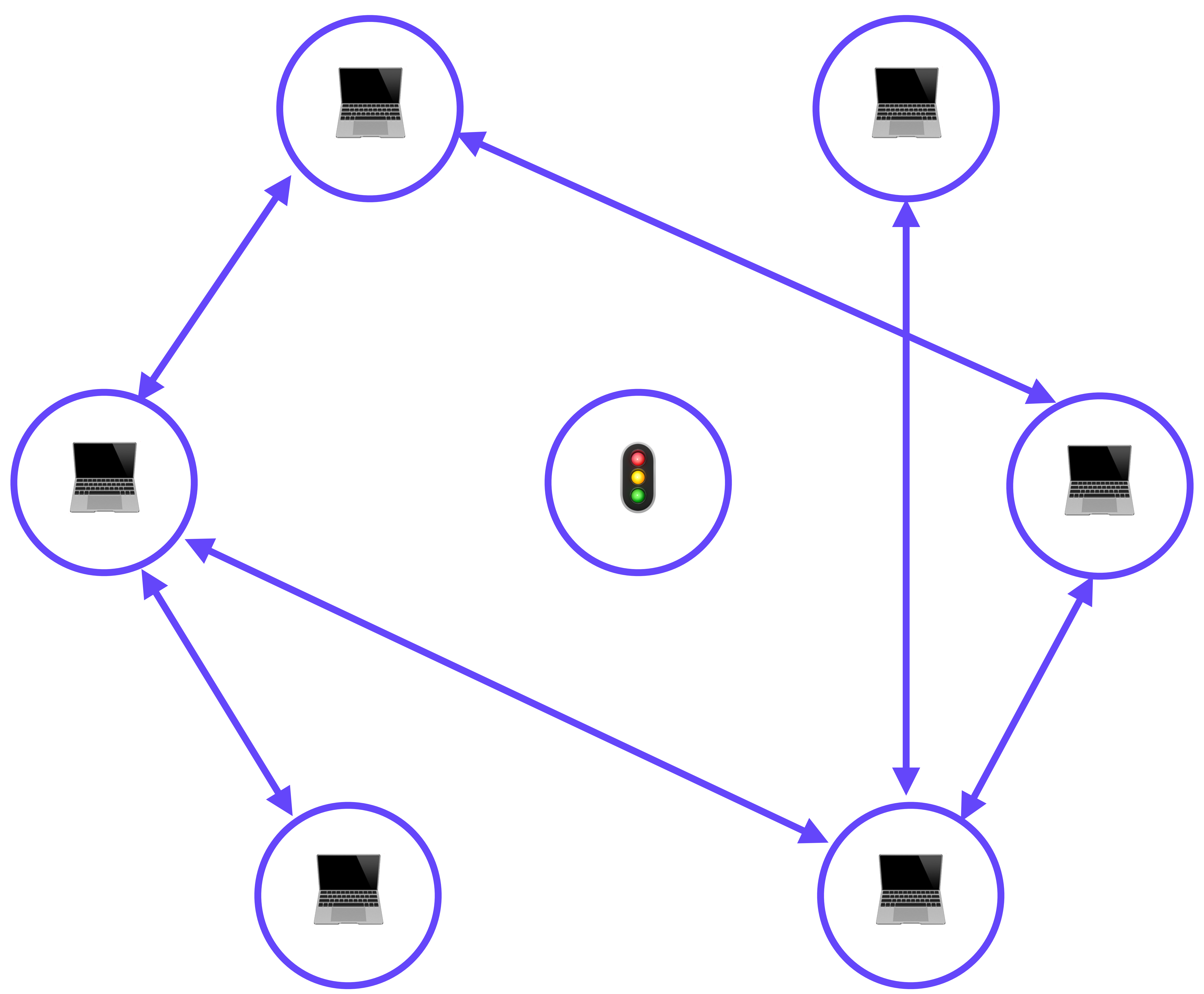
- Distributed / uncontrolled topology
- Eventual consistency

CRDT CASE STUDY

SCENARIO

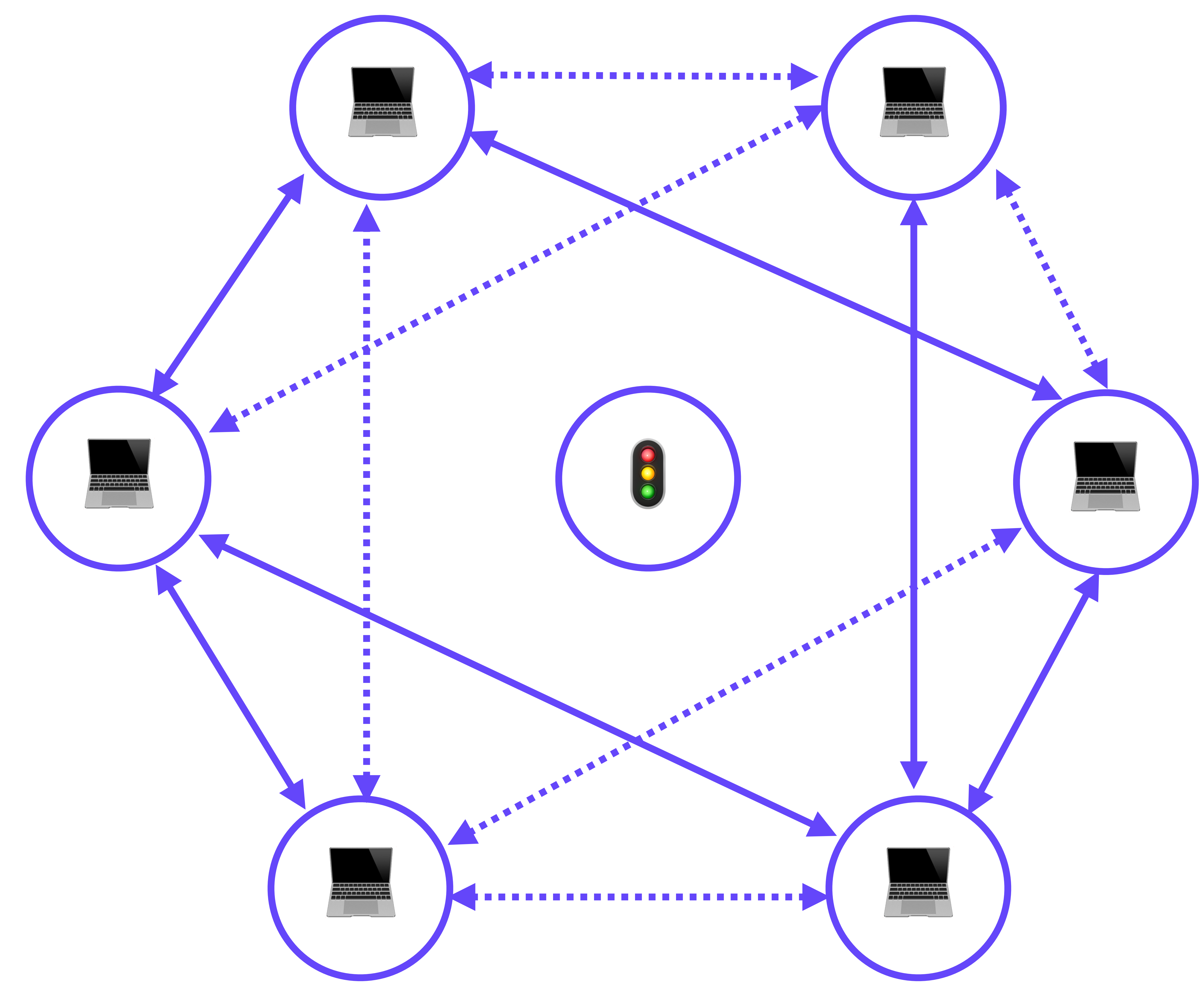
- Distributed / uncontrolled topology
- Eventual consistency



CRDT CASE STUDY

SCENARIO

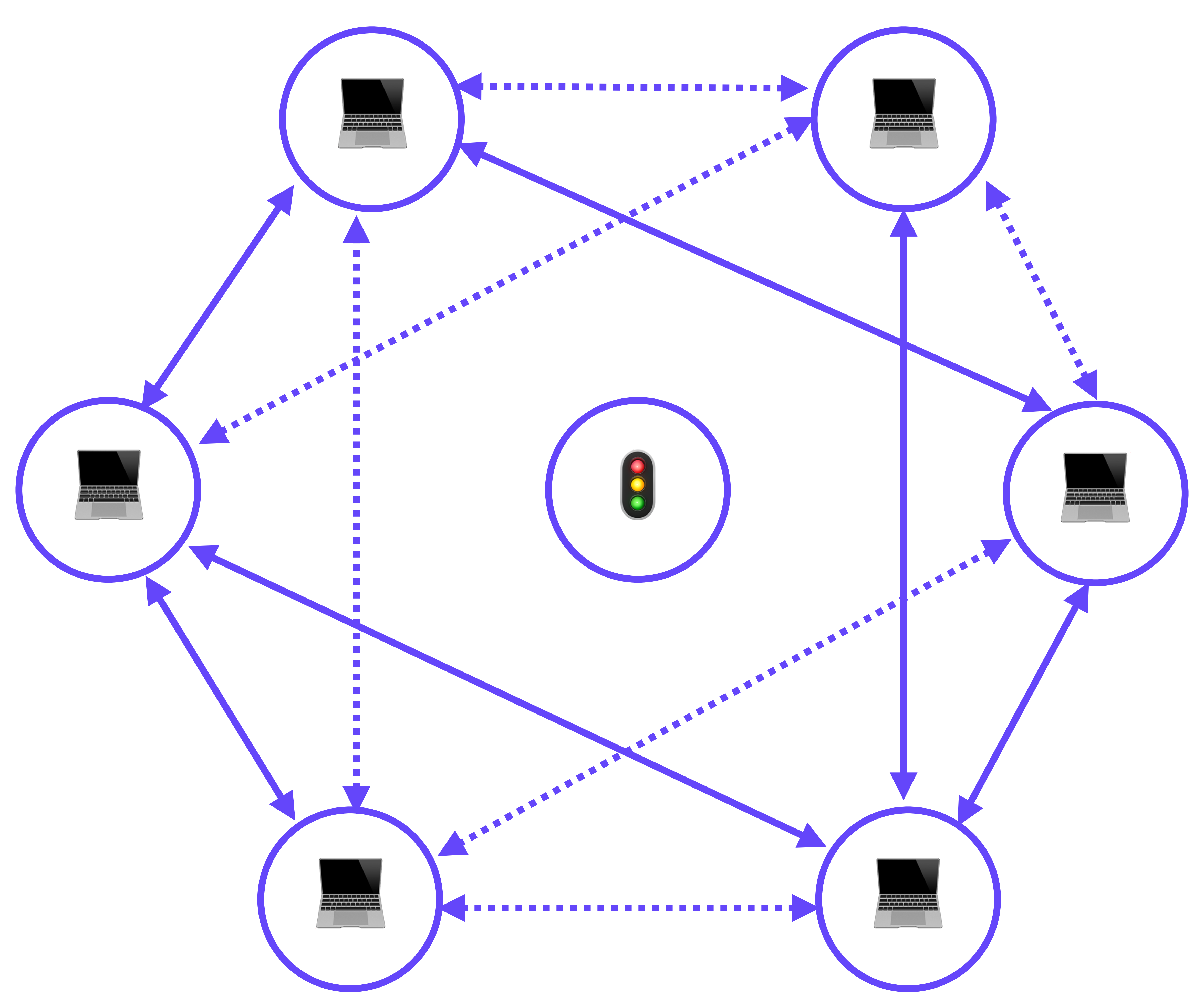
- Distributed / uncontrolled topology
- Eventual consistency



CRDT CASE STUDY

SCENARIO

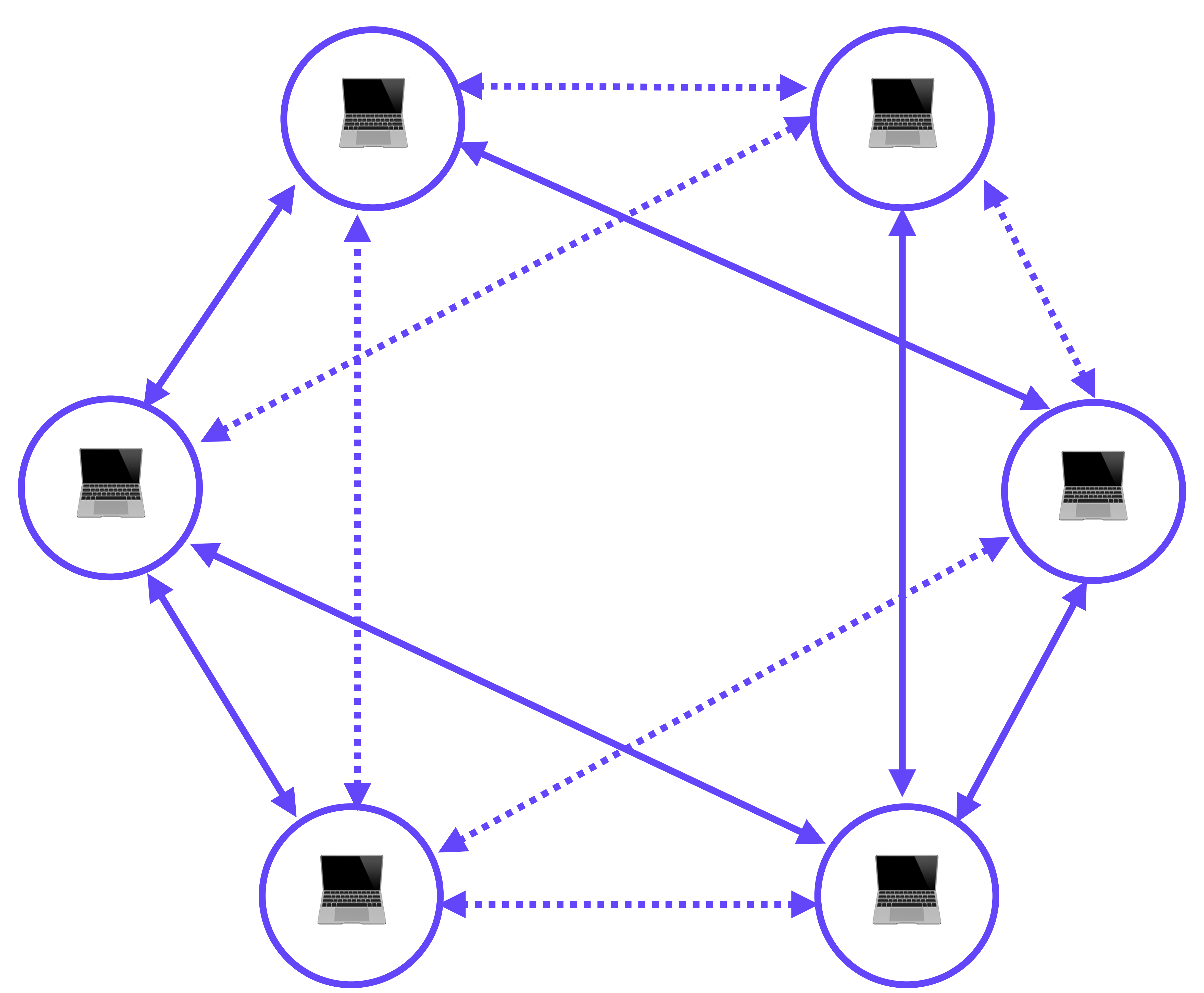
- Distributed / uncontrolled topology
- Eventual consistency
- High resilience
  - Infinite(ish) time tolerance
  - Decentralized (e.g. via gossip)
  - Self-healing



CRDT CASE STUDY

SCENARIO

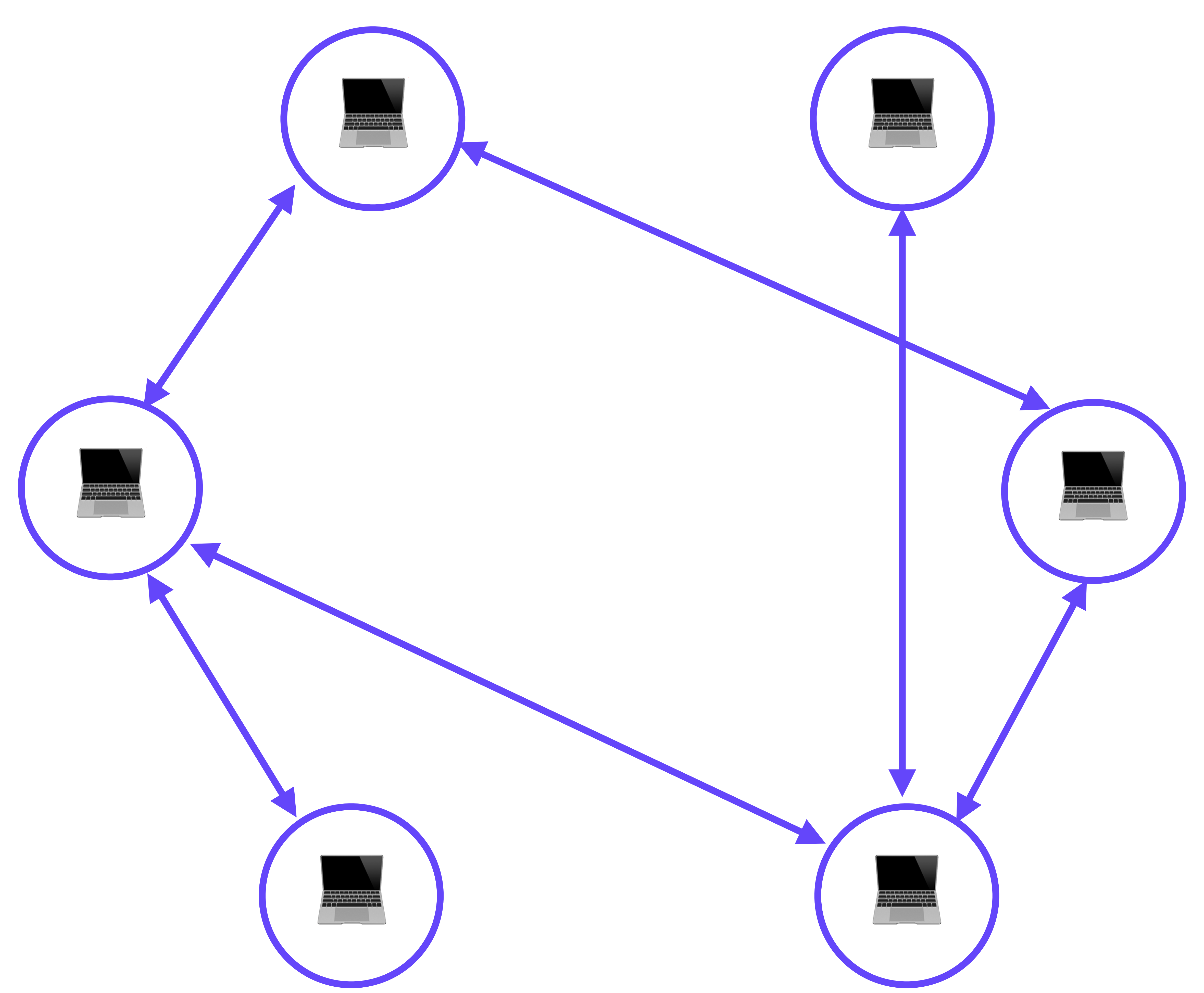
- Distributed / uncontrolled topology
- Eventual consistency
- High resilience
  - Infinite(ish) time tolerance
  - Decentralized (e.g. via gossip)
  - Self-healing



CRDT CASE STUDY

SCENARIO

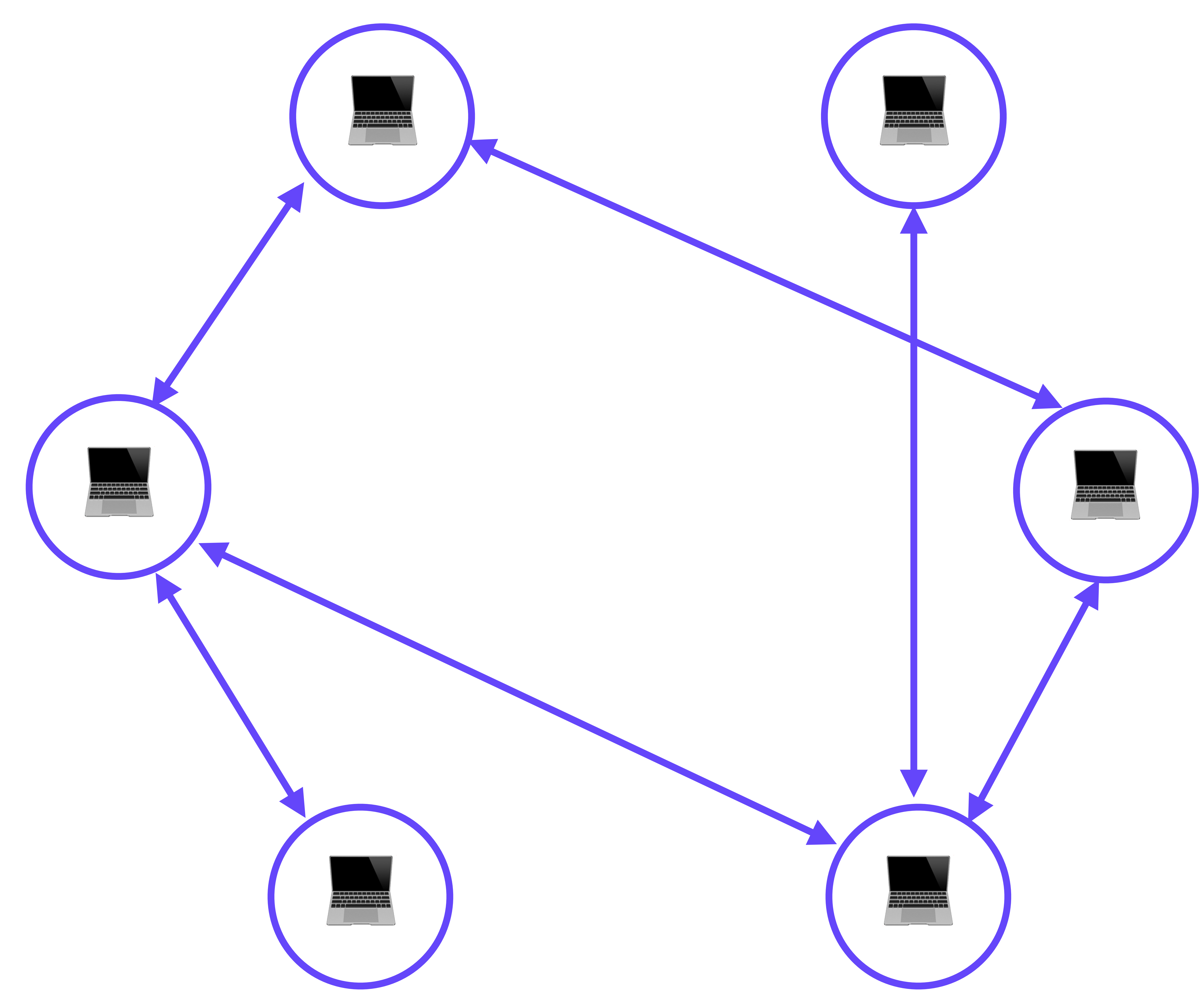
- Distributed / uncontrolled topology
- Eventual consistency
- High resilience
  - Infinite(ish) time tolerance
  - Decentralized (e.g. via gossip)
  - Self-healing



CRDT CASE STUDY

SCENARIO

- Distributed / uncontrolled topology
- Eventual consistency
- High resilience
  - Infinite(ish) time tolerance
  - Decentralized (e.g. via gossip)
  - Self-healing
- Two variants
  - State-based
  - Operation-based ✅



CRDT CASE STUDY

BASIC: POSITIVE/NEGATIVE COUNTER

CRDT CASE STUDY

BASIC: POSITIVE/NEGATIVE COUNTER

- Two sets
  - Increments
  - Decrements
- Operations
  - Internal
    - Increment
    - Decrement
    - Compare
  - External
    - Merge
    - Query

CRDT CASE STUDY

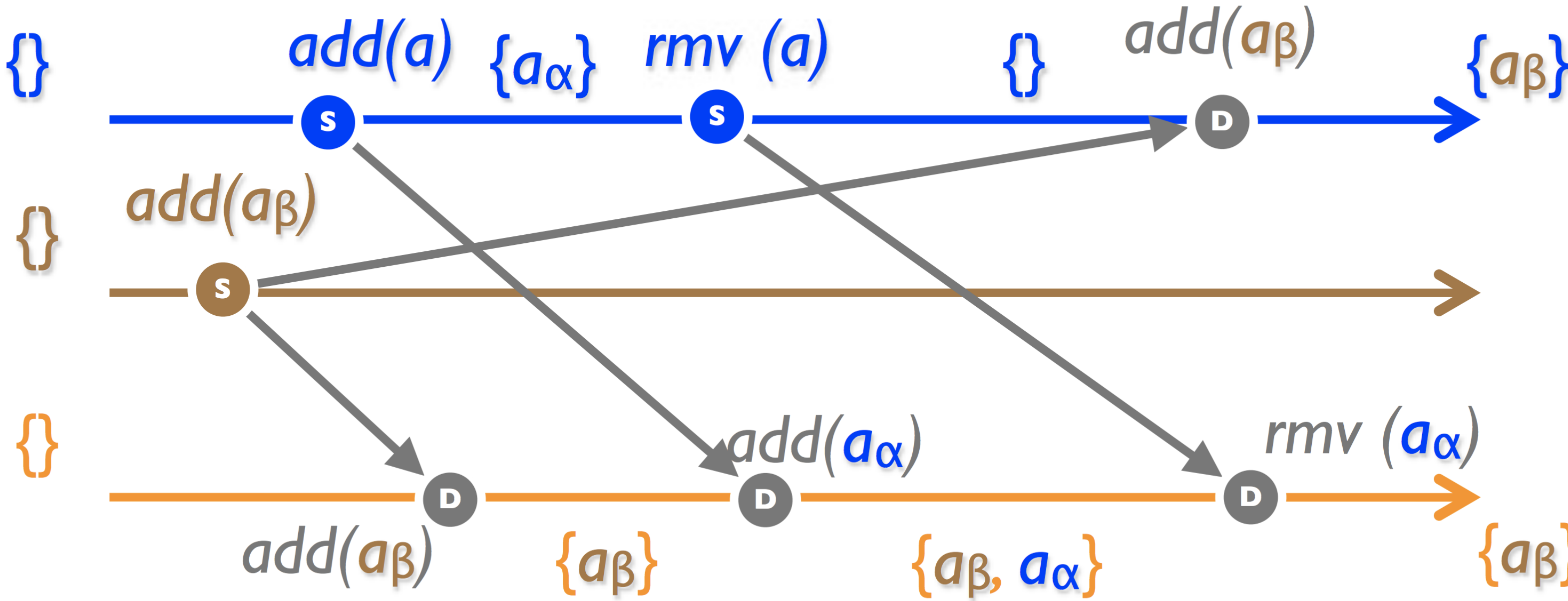
BASIC: POSITIVE/NEGATIVE COUNTER

- Two sets
  - Increments
  - Decrements
- Operations
  - Internal
    - Increment
    - Decrement
    - Compare
  - External
    - Merge
    - Query

- Challenges
  - Negative counts
  - Double deletes
- Solutions
  - Only delete existing items
  - Commutativity

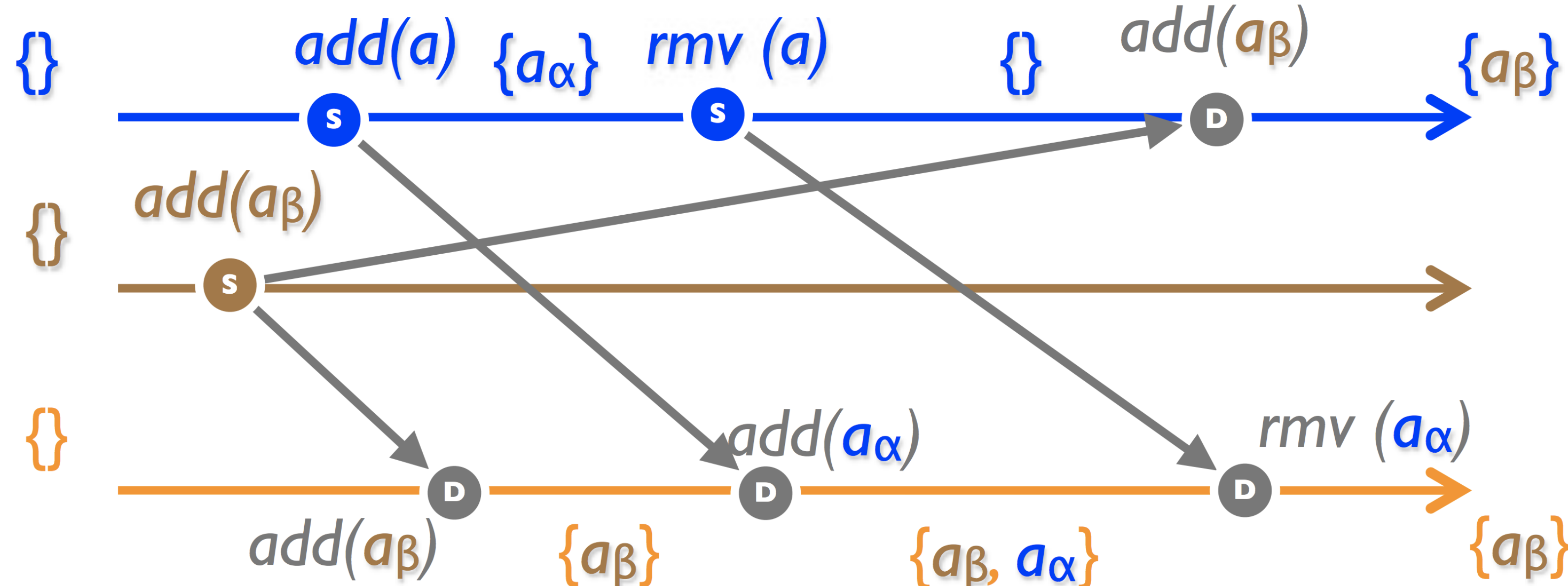
CRDT CASE STUDY

OBSERVE/REMOVE SET



CRDT CASE STUDY

OBSERVE/REMOVE SET

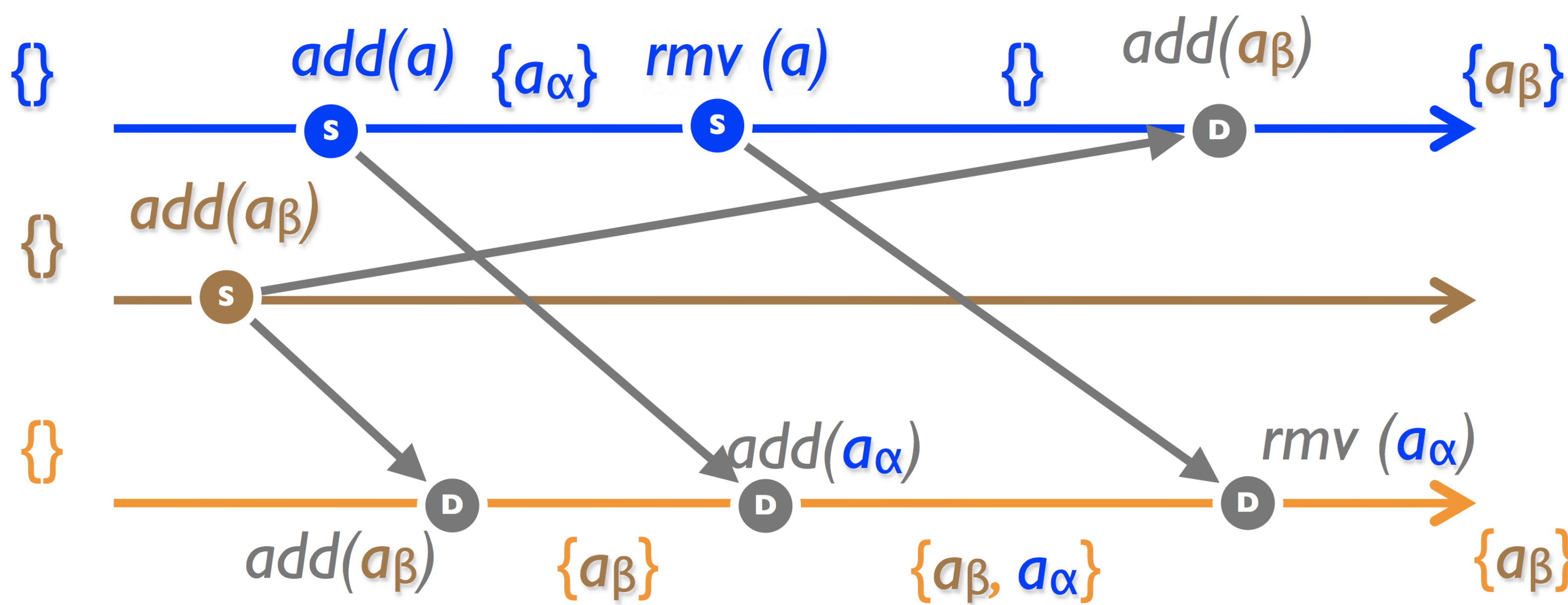


- Two sub-sets
  - Add
  - Remove
- Operations
  - Internal
    - Add
    - Remove
    - Compare
    - Tag
  - External
    - Merge — semigroup
    - Query

CRDT CASE STUDY

OBSERVE/REMOVE SET

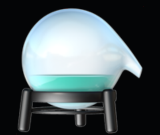
- Two sub-sets
  - Add
  - Remove
- Operations
  - Internal
    - Add
    - Remove
    - Compare
    - Tag
  - External
    - Merge — semigroup
    - Query



- Challenges
  - Add-after-remov
  - Unique tags vs re-add
  - Can only delete what you know about
    - But deleting an unknown element is the same as a no-op, so great!
    - Remove has priority over add
  - Wall clock timestamps are unreliable
    - Need monotonically increasing structure
- Solutions
  - Only delete existing items
  - Commutativity
- Protocols
  - Collectable
  - Enumerable
  - Semigroup (merge) -> Commutative (CRDT)
    - “Meet semilattice”

WHAT EVEN IS ELIXIR?

WHAT EVEN IS ELIXIR?



WHAT EVEN IS ELIXIR?

SHOW OF HANDS ✋

Is Elixir a functional language?

WHAT EVEN IS ELIXIR?

STRUCTURAL INTEGRITY

WHAT EVEN IS ELIXIR?

STRUCTURAL INTEGRITY

☝️ Production Elixir can actually be pretty unstructured

WHAT EVEN IS ELIXIR?

STRUCTURAL INTEGRITY

☝️ Production Elixir can actually be pretty unstructured

- Imperative
  - Lots of ambient state
  - Spread across processes
  - Ordering & timing bugs

WHAT EVEN IS ELIXIR?

STRUCTURAL INTEGRITY

☝️ Production Elixir can actually be pretty unstructured

- Imperative
  - Lots of ambient state
  - Spread across processes
  - Ordering & timing bugs
- Functional
  - Well-defined structured functions (`map`, `filter`, `reduce`)
  - Immutable references

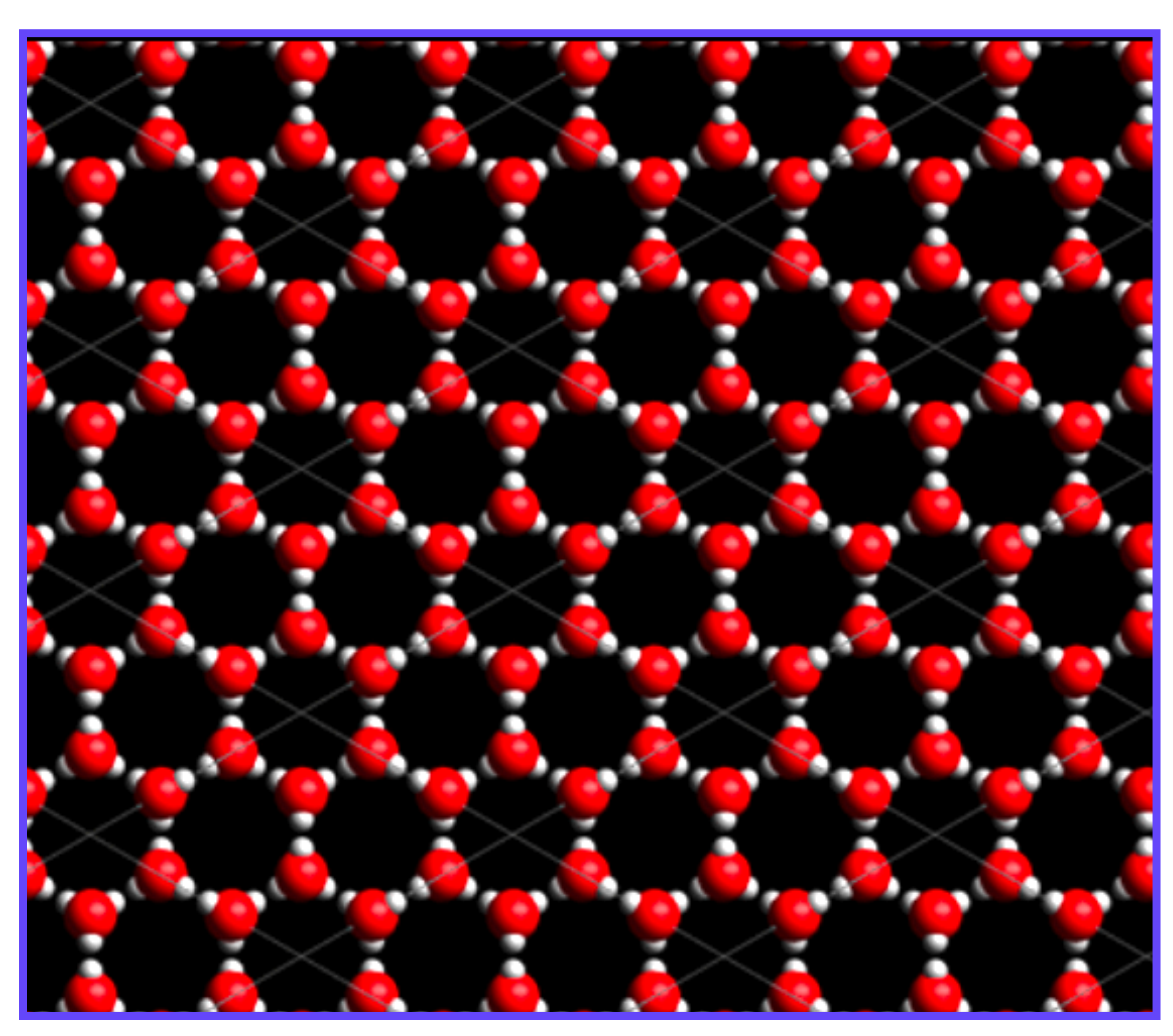
WHY NOT FULLY STRUCTURED?

WHY NOT FULLY STRUCTURED?

- Naive structure can be rigid!
- Composition
- Orthogonality

WHY NOT FULLY STRUCTURED?

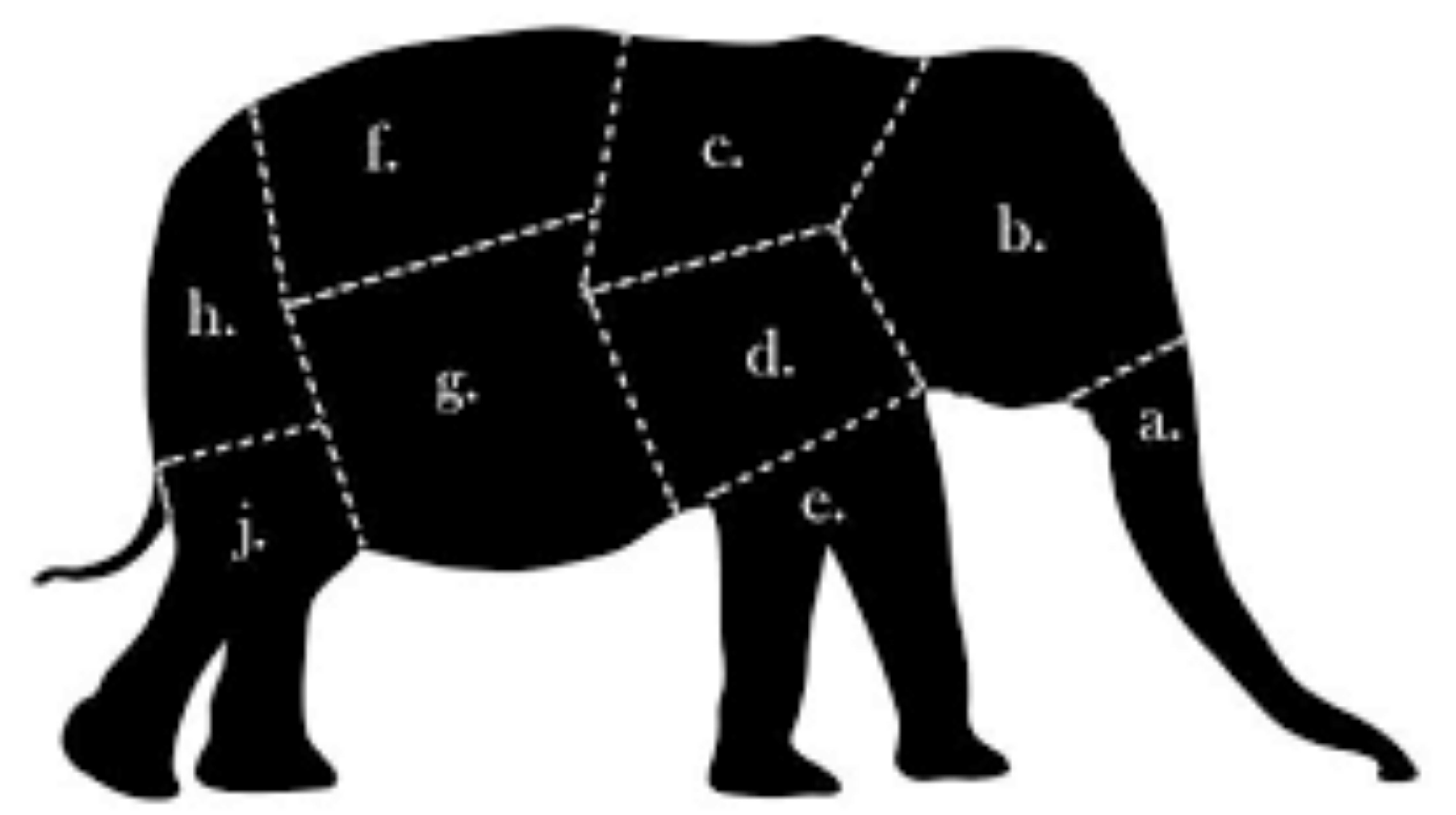
- Naive structure can be rigid!
- Composition
- Orthogonality



Credit: psihedelisto

WHAT EVEN IS ELIXIR?

HOW TO EAT THE STRUCTURAL ELEPHANT



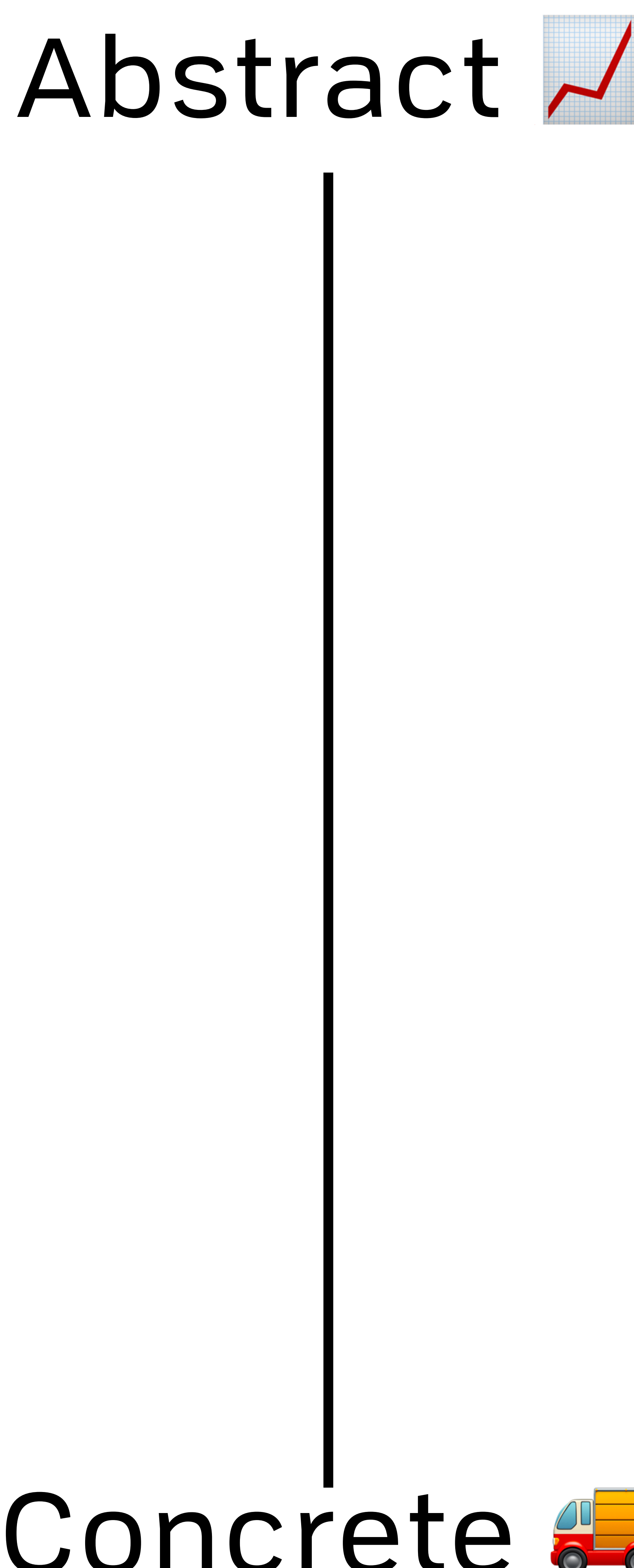
POWER UP

Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. **Data structures, not algorithms, are central to programming.**

~ROB PIKE, 5 RULES OF PROGRAMMING

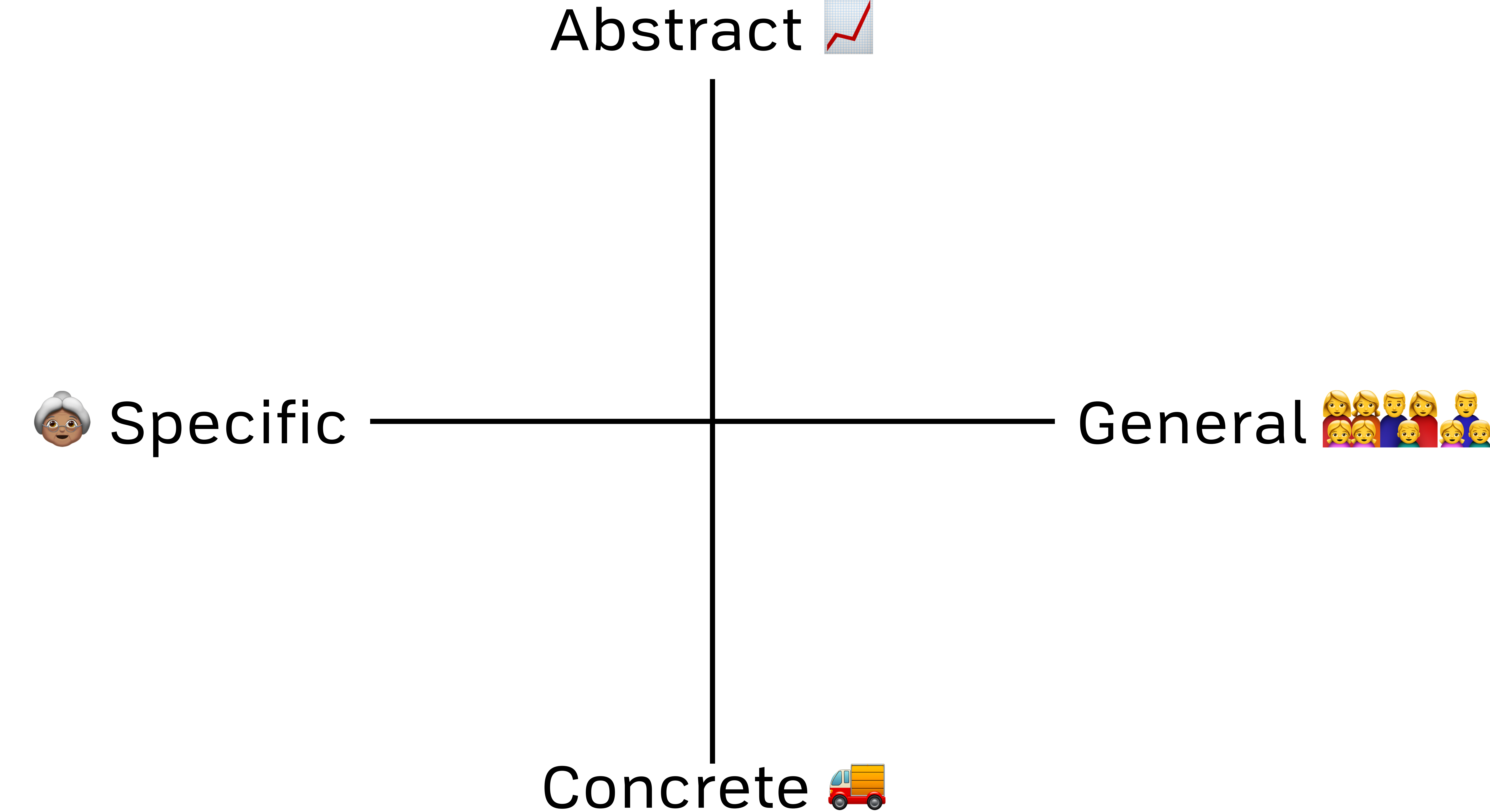
DEFINITIONS

DEFINITIONS



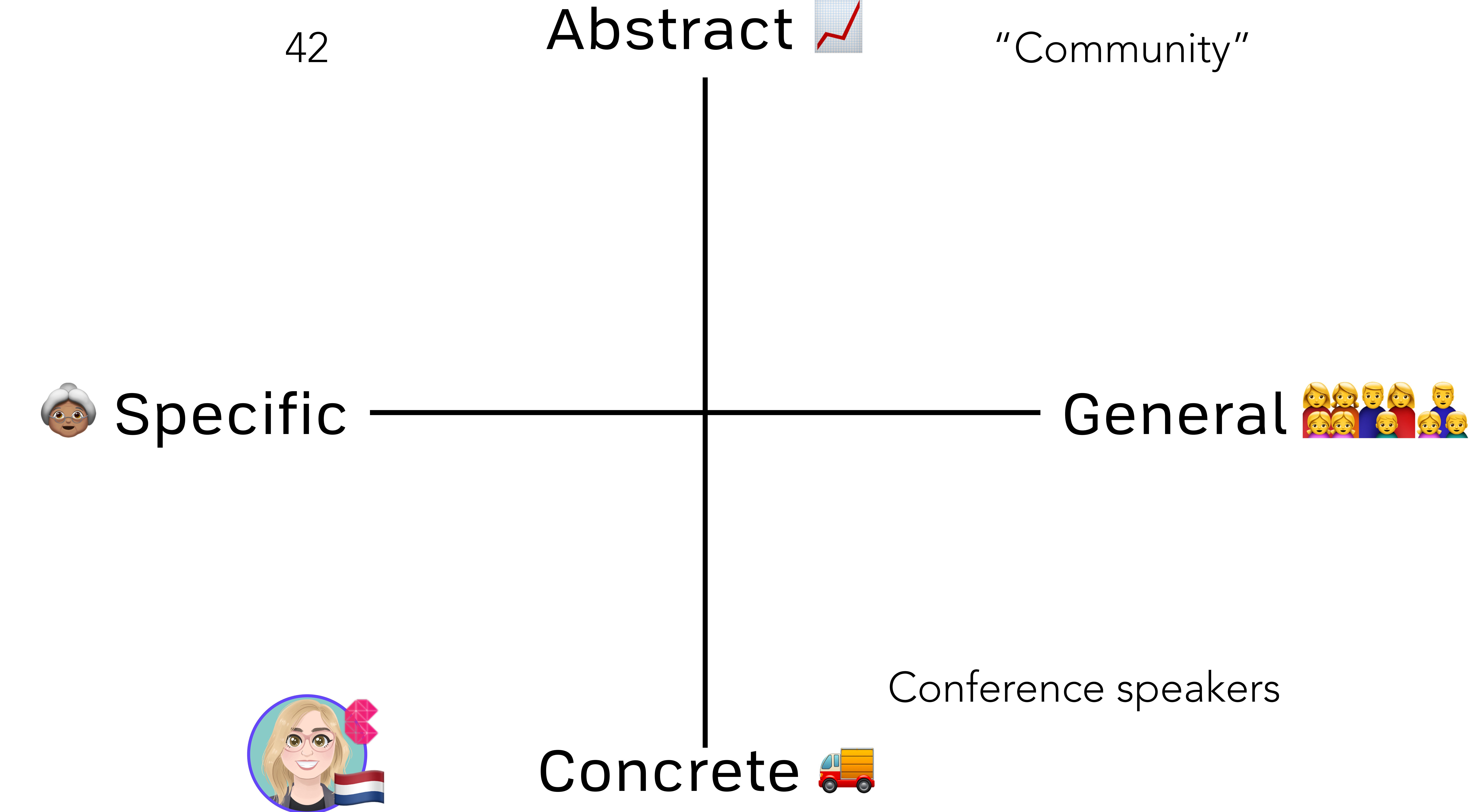
ABSTRACTION & DSLS

DEFINITIONS



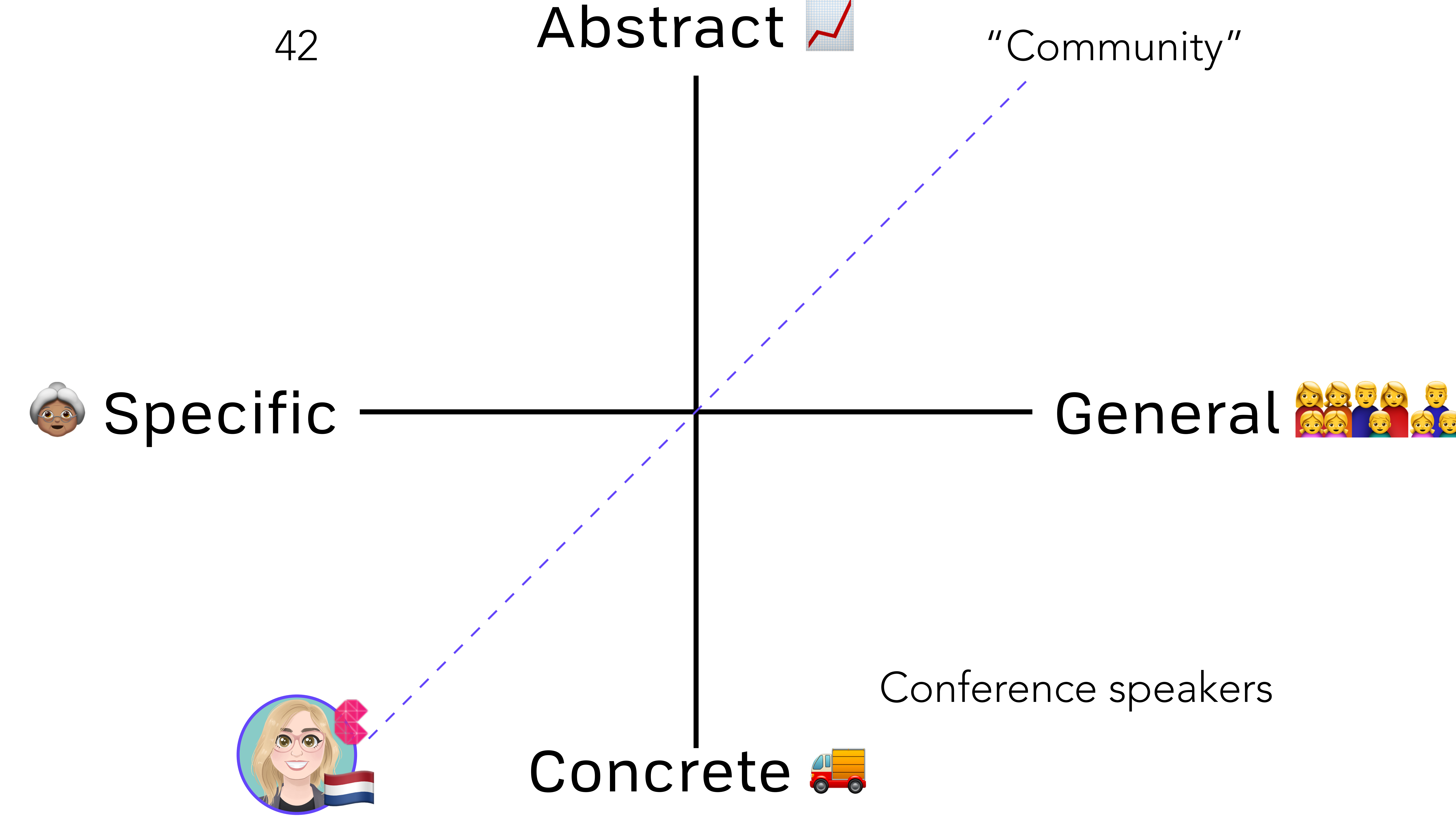
ABSTRACTION & DSLS

DEFINITIONS



ABSTRACTION & DSLS

DEFINITIONS



ABSTRACTION & DSLS

DEFINITIONS