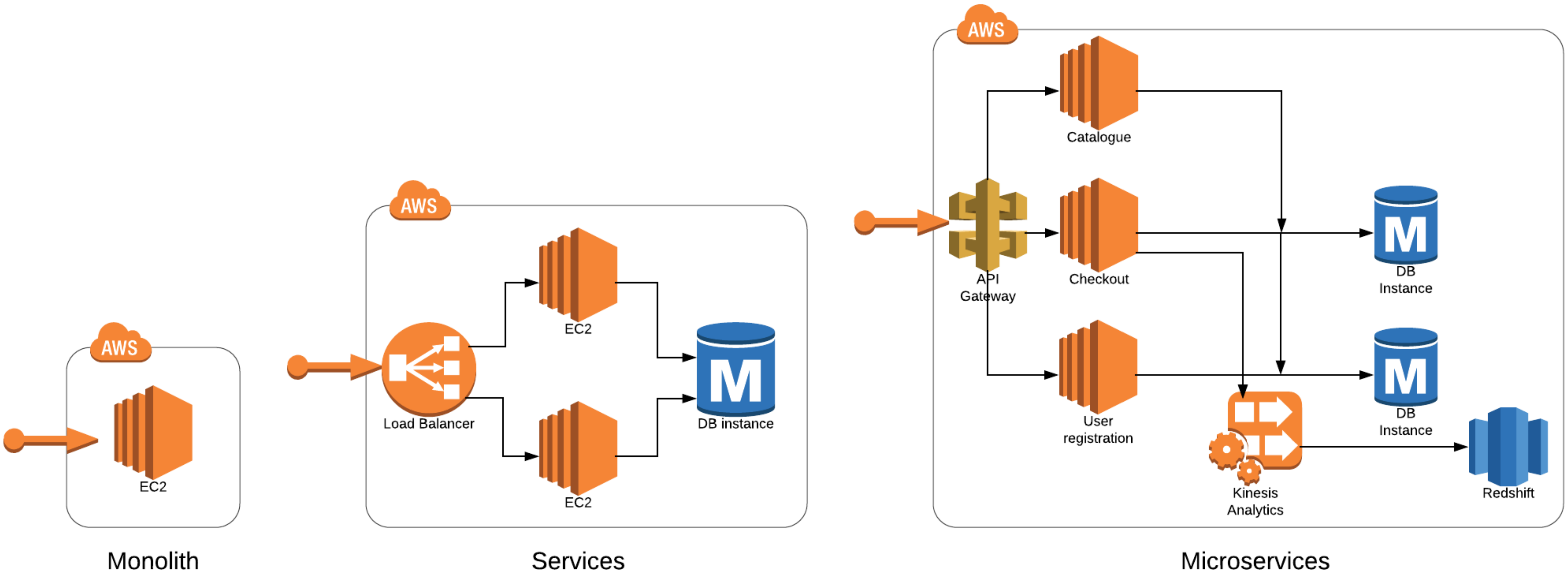
Visual-textual framework for serverless computation: a Luna Language approach

Piotr Moczurad, Maciej Malawski

Workshop on Serverless Computing 2018
Zürich, 20.12.2018

The evolution of an architecture



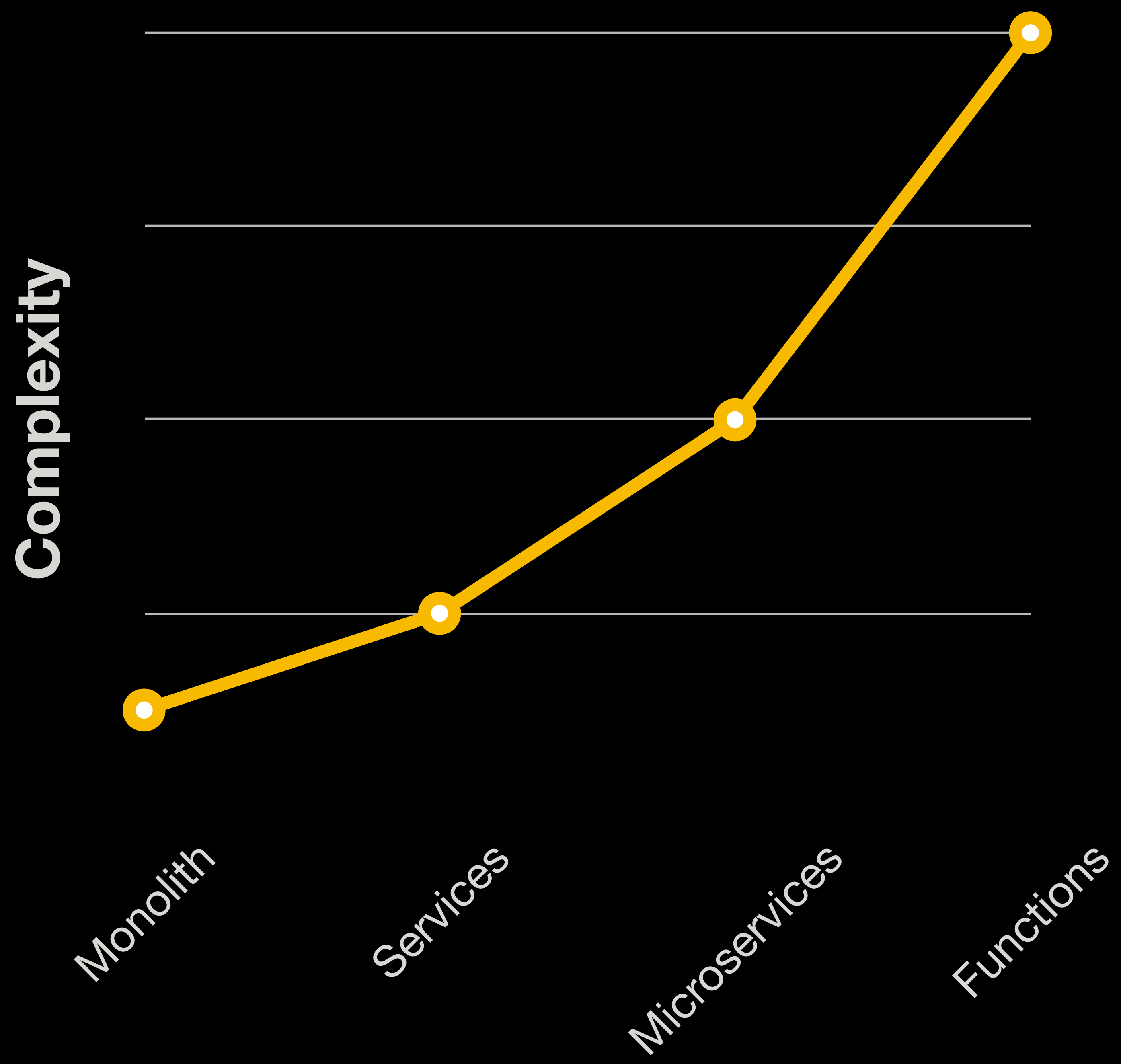
The next step: serverless

A function is a first-class unit that is billed and deployed separately.

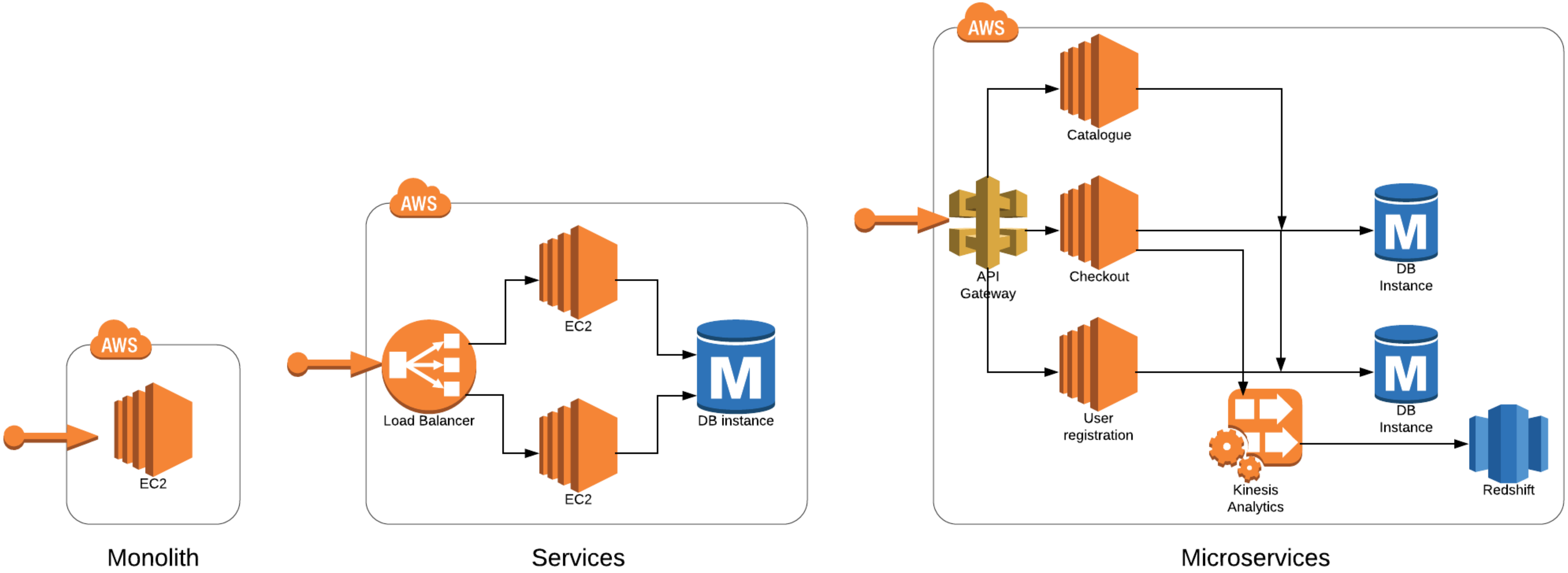
It seems to be the finest level of granularity we can achieve in cloud computing.

The trend

- Subdivide computations
- Separate responsibilities
- Reduce cost
- Increase performance
- Increase **scalability**
- Increase **complexity**…



Look again…



How do we go about visualising the serverless architecture?

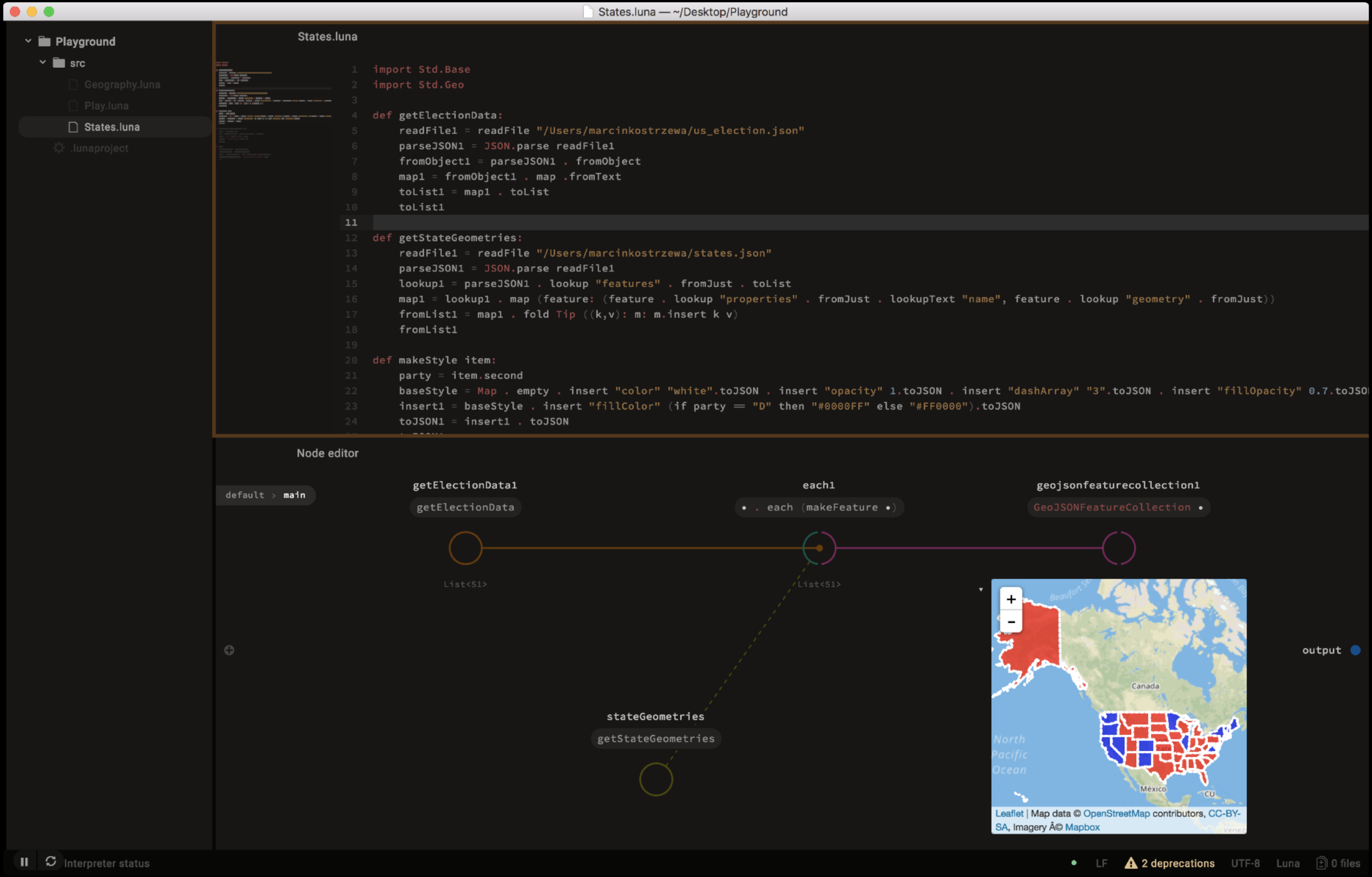
Visualisation

Ideally: **a graph.**

Node: a **function.**

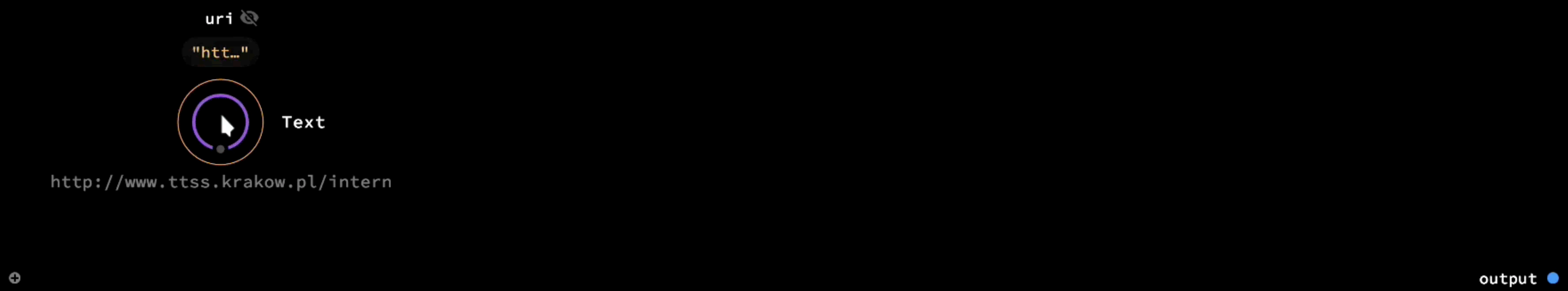
Edge: the **flow of data** between functions.

Visual-textual programming

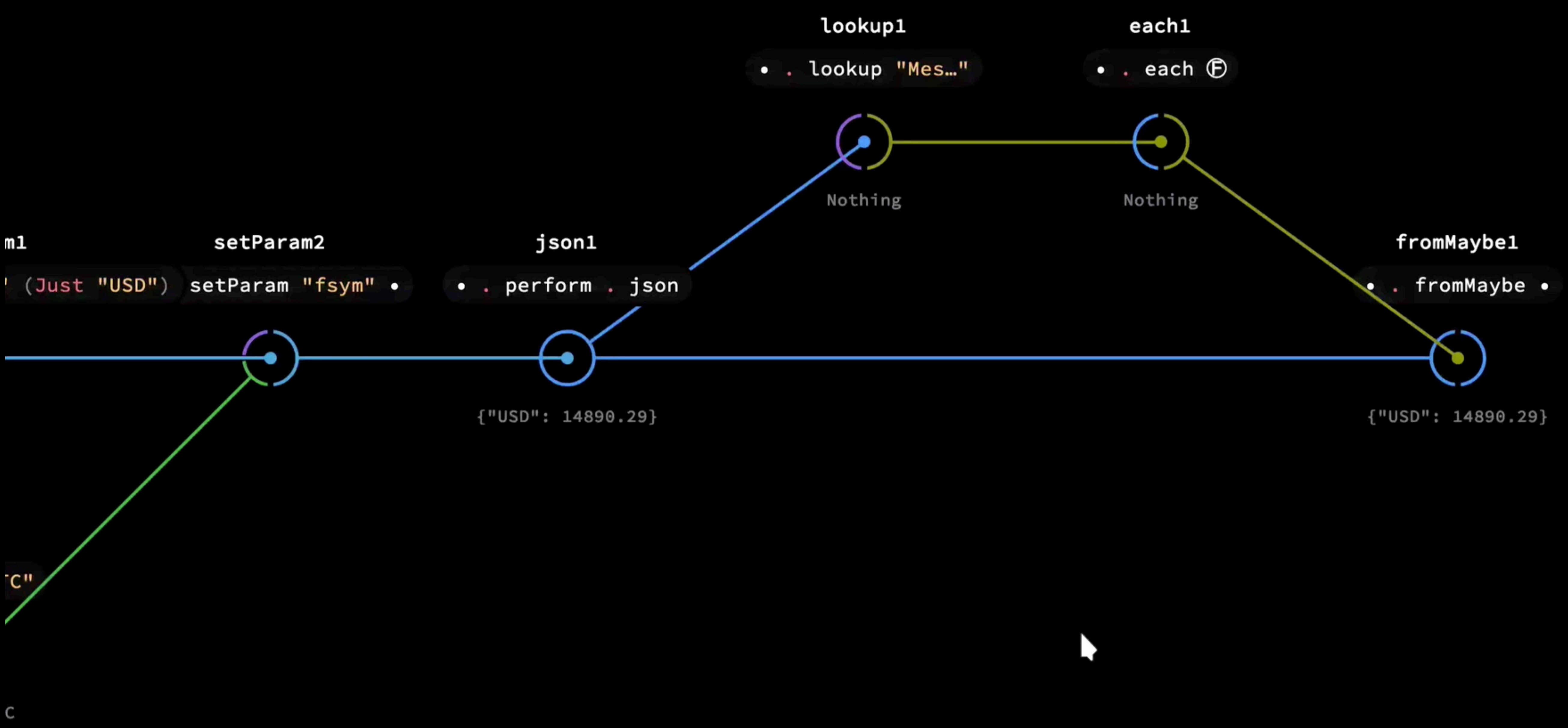


```
import Std.Base
import Std.Geo

def getElectionData:
    readFile1 = readFile "/Users/marcinkostrzewa/us_election.json"
    parseJSON1 = JSON.parse readFile1
    fromObject1 = parseJSON1 . fromObject
    map1 = fromObject1 . map .fromText
    toList1 = map1 . toList
    toList1

def getStateGeometries:
    readFile1 = readFile "/Users/marcinkostrzewa/states.json"
    parseJSON1 = JSON.parse readFile1
    lookup1 = parseJSON1 . lookup "features" . fromJust . toList
    map1 = lookup1 . map (feature: (feature . lookup "properties" . fromJust . lookupText "name", feature . lookup "geometry" . fromJust))
    fromList1 = map1 . fold Tip ((k,v): m: m.insert k v)
    fromList1

def makeStyle item:
    party = item.second
    baseStyle = Map . empty . insert "color" "white".toJSON . insert "opacity" 1.toJSON . insert "dashArray" "3".toJSON . insert "fillOpacity" 0.7.toJSO
    insert1 = baseStyle . insert "fillColor" (if party == "D" then "#0000FF" else "#FF0000").toJSON
    toJSON1 = insert1 . toJSON
```

Data flow graph

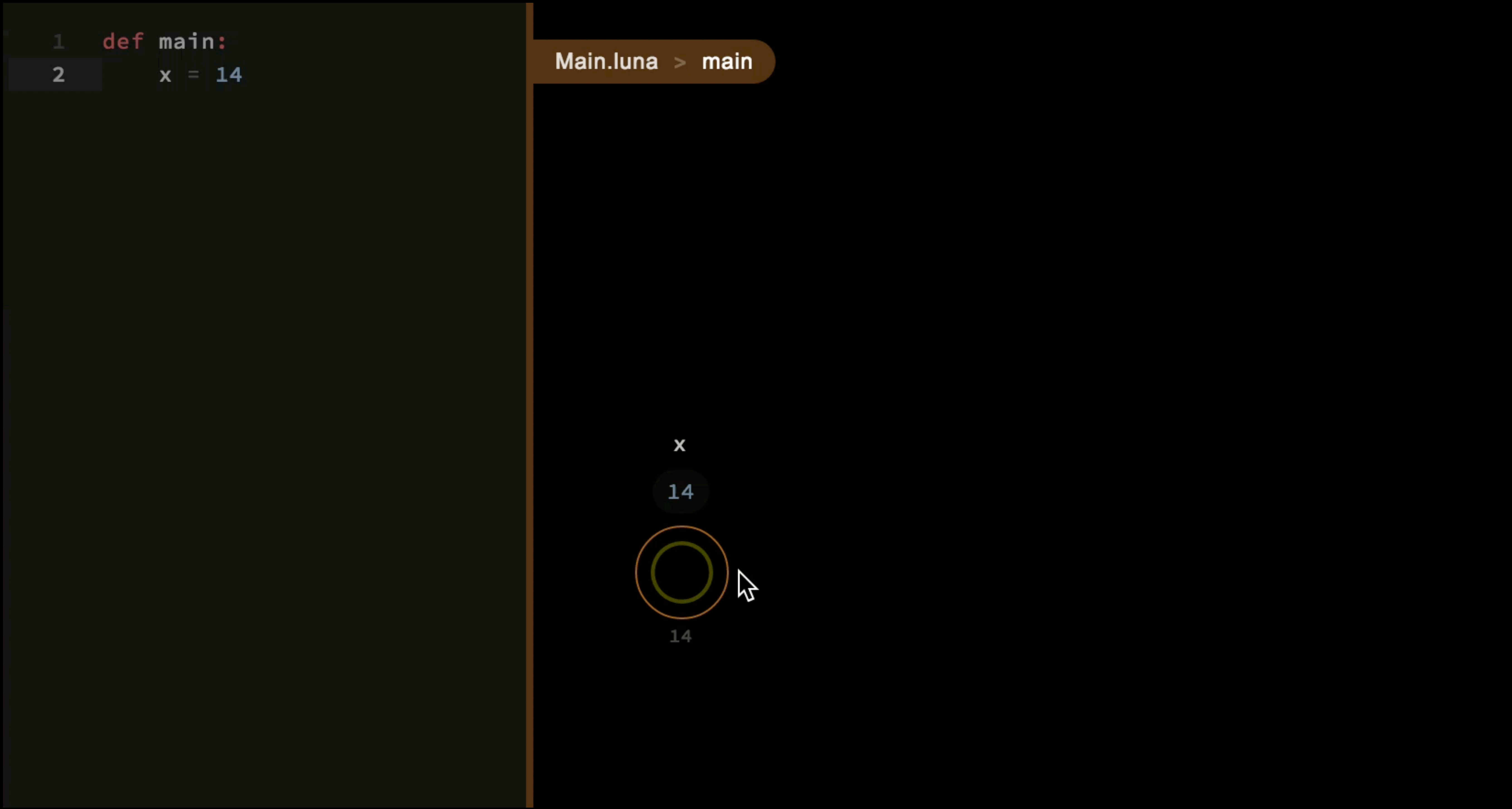


Adjustable levels of abstraction



```
lookup1
• . lookup "Mes…"
Nothing

each1
• . each Ⓕ
Nothing

m1
' (Just "USD")

setParam2
setParam "fsym" •

json1
• . perform . json
{"USD": 14890.29}

fromMaybe1
• . fromMaybe •
{"USD": 14890.29}

C"

C
```

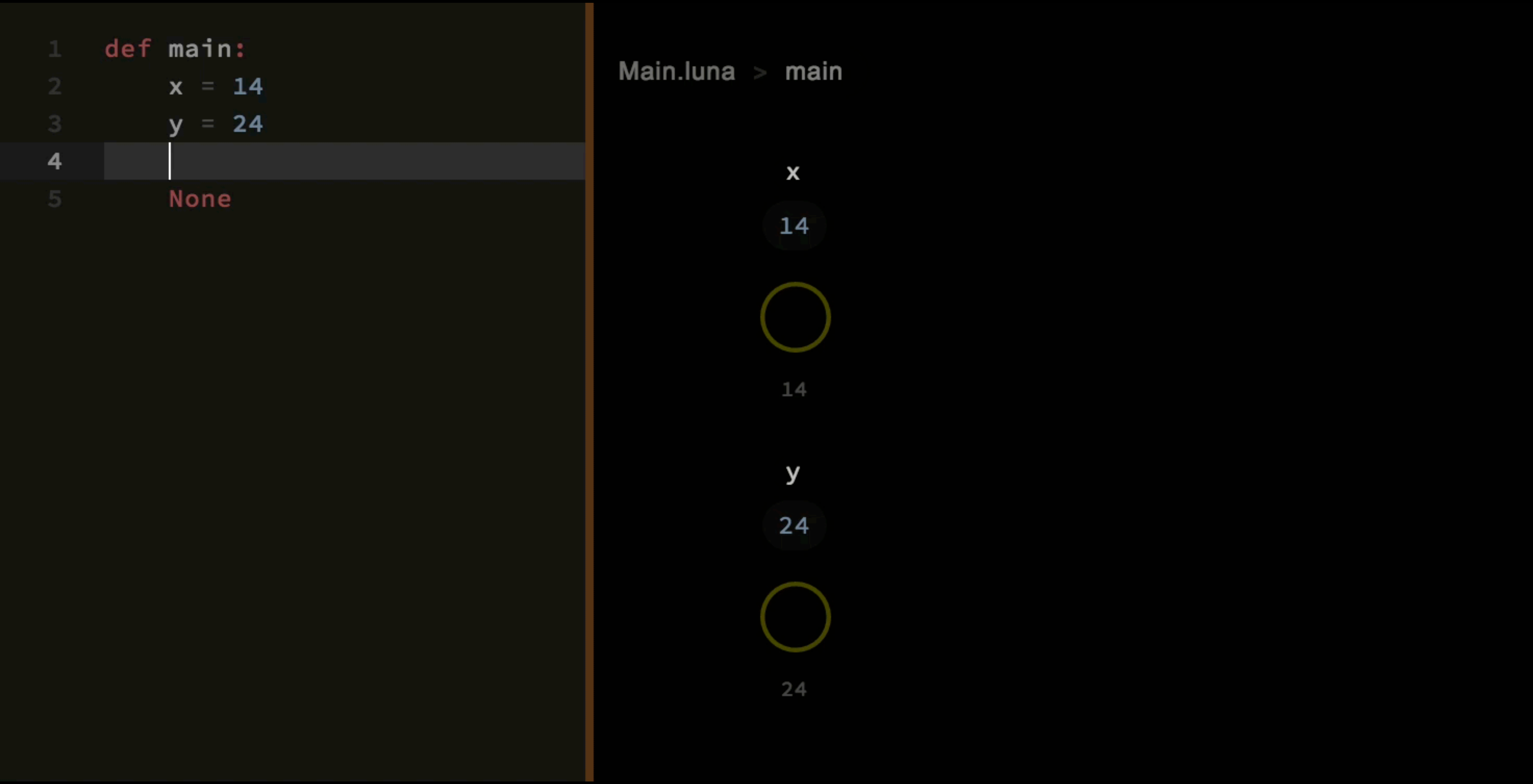
Code representation



```
def main:
    x = 14
```

Every node corresponds to one line of code

Code representation



```
def main:
    x = 14
    y = 24

    None
```

Everything you type is reflected by the graph

Benefits

- Clear pipeline in form of a readable graph
- Manageable complexity: correctness & productivity
- Communication backed by the compiler

Luna + Serverless

Ideal world

Functions executing remotely (e.g. on AWS Lambda) **indistinguishable** from local asynchronous functions.

Rationale

Local async function:

- Doesn’t return immediately
- Can fail (interrupted?)

Remote function:

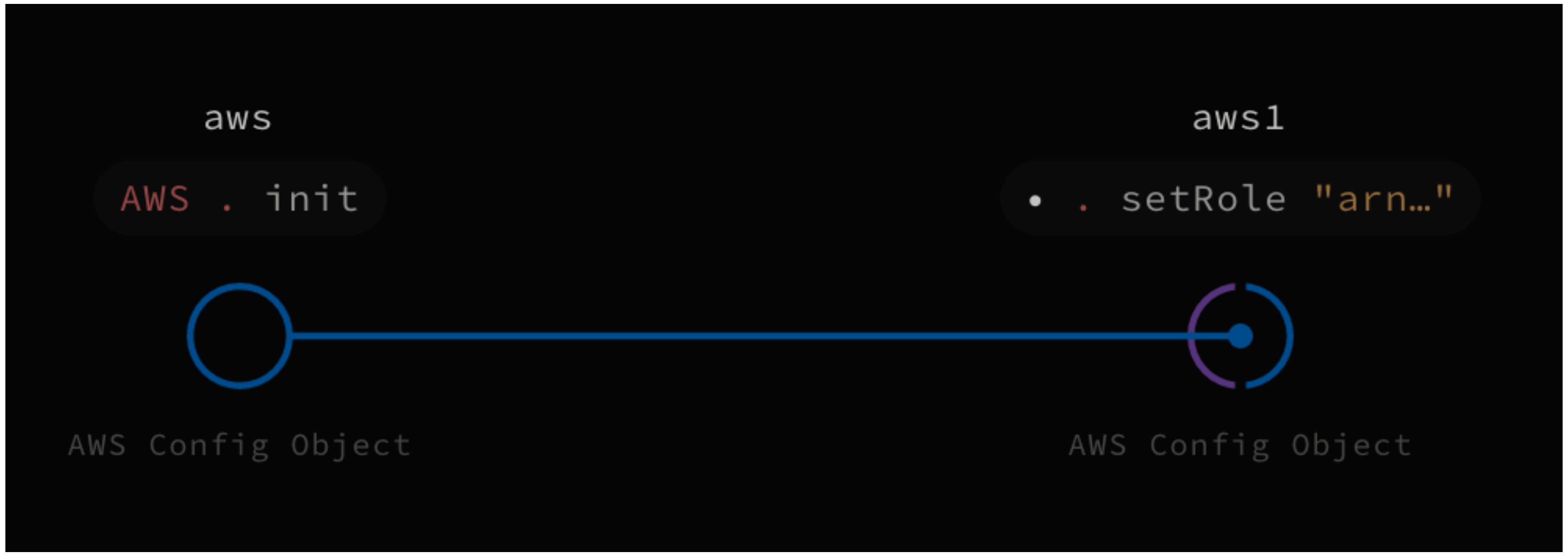
- Doesn’t return immediately
- Can fail (network failure?)

This can be captured by the same result type:

`Future[T]`

The Luna Serverless Framework

Initialization/configuration

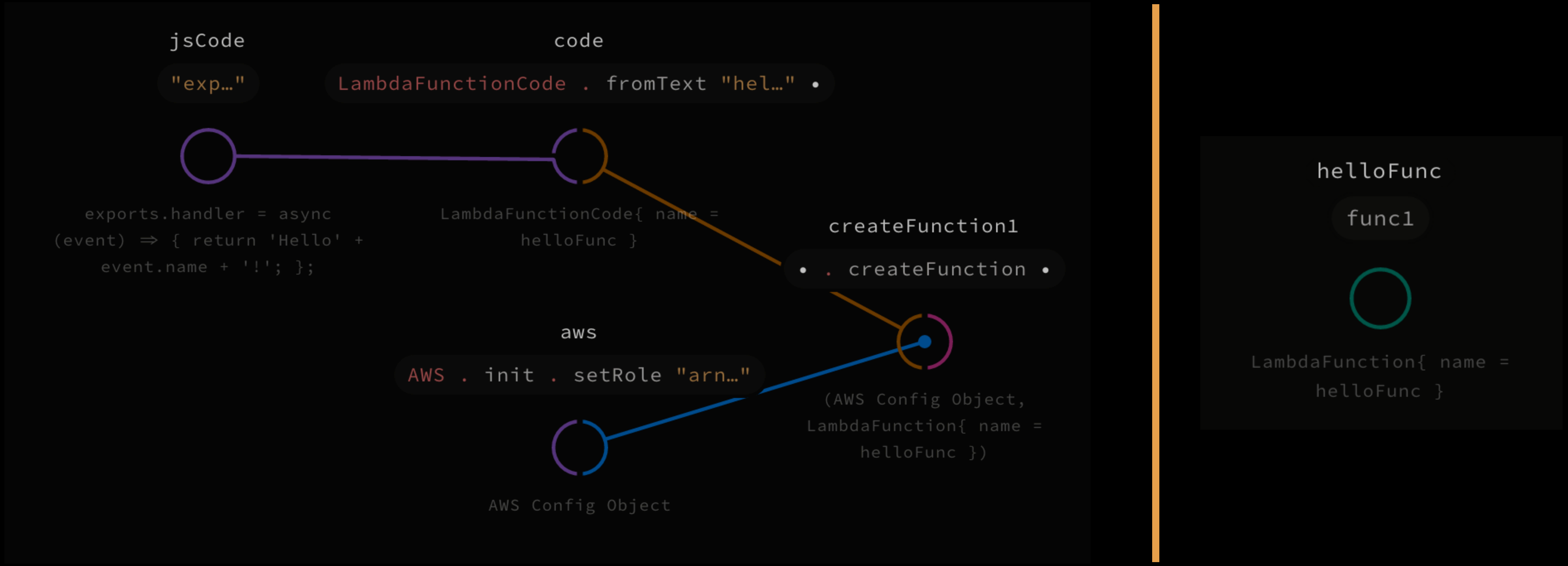


```
aws
AWS . init
AWS Config Object

aws1
• . setRole "arn…"
AWS Config Object
```

The **configuration** is abstracted away.
Sensible, overridable **defaults** are what makes the work efficient.

```
aws  = AWS.init
aws1 = aws.setRole “myRole…”
```

Defining a function



jsCode

"exp…"

exports.handler = async (event) ⇒ { return 'Hello' + event.name + '!'; };

code

LambdaFunctionCode . fromText "hel…" •

LambdaFunctionCode{ name = helloFunc }

createFunction1

• . createFunction •

(AWS Config Object, LambdaFunction{ name = helloFunc })

aws

AWS . init . setRole "arn…"

AWS Config Object

helloFunc

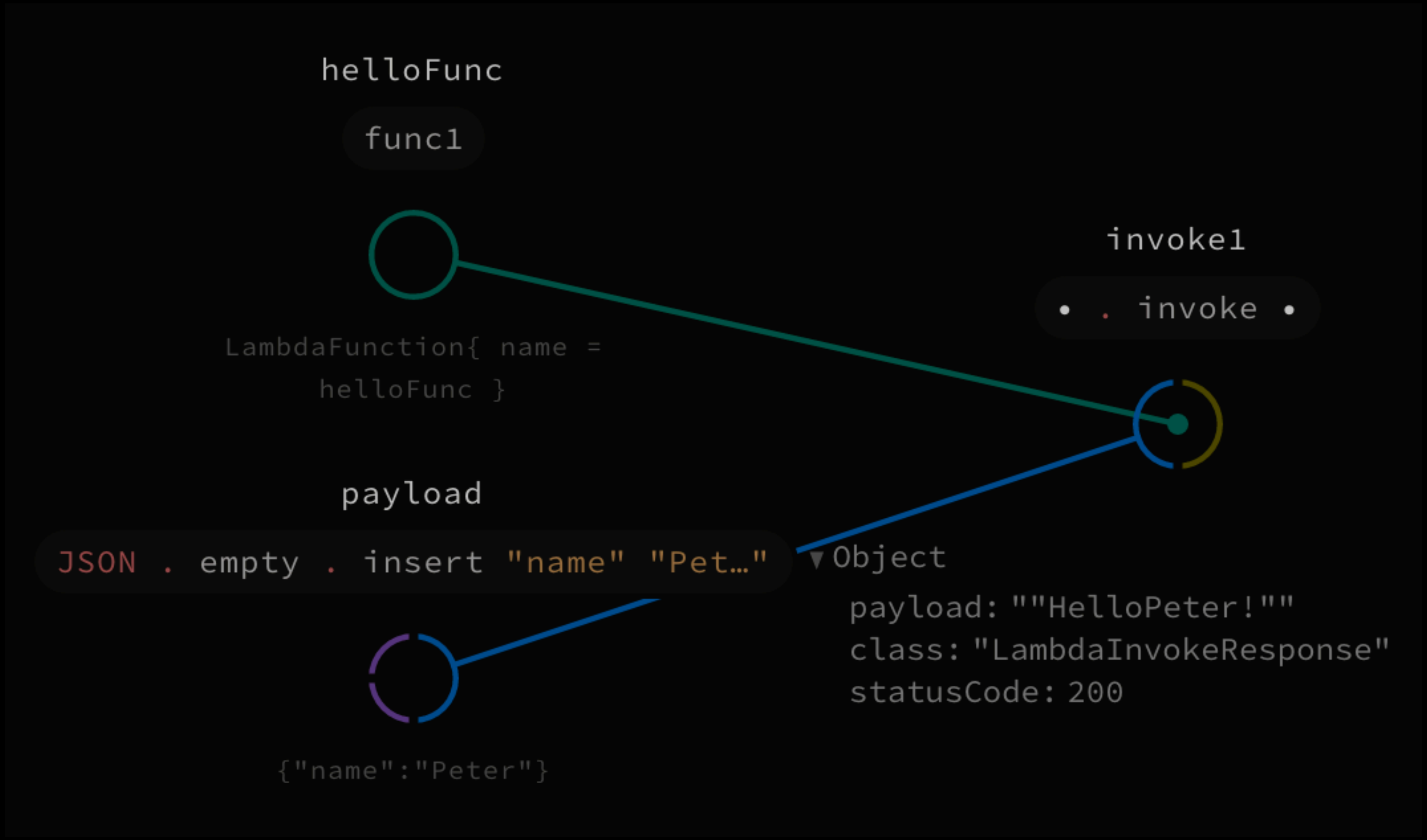
func1

LambdaFunction{ name = helloFunc }

PoC stage: take the function code **in JS** and create a Lambda function.

```
jsCode = “exports.handler = async (event) => {return ‘Hello’ + event.name + ‘!’; };”
code = LambdaFunctionCode.fromText “helloFunc” jsCode
createFunction1 = aws.createFunction code
```

Invoking a function


helloFunc
func1
LambdaFunction{ name = helloFunc }
invoke1
• . invoke •
payload
JSON . empty . insert "name" "Pet…"
{"name":"Peter"}

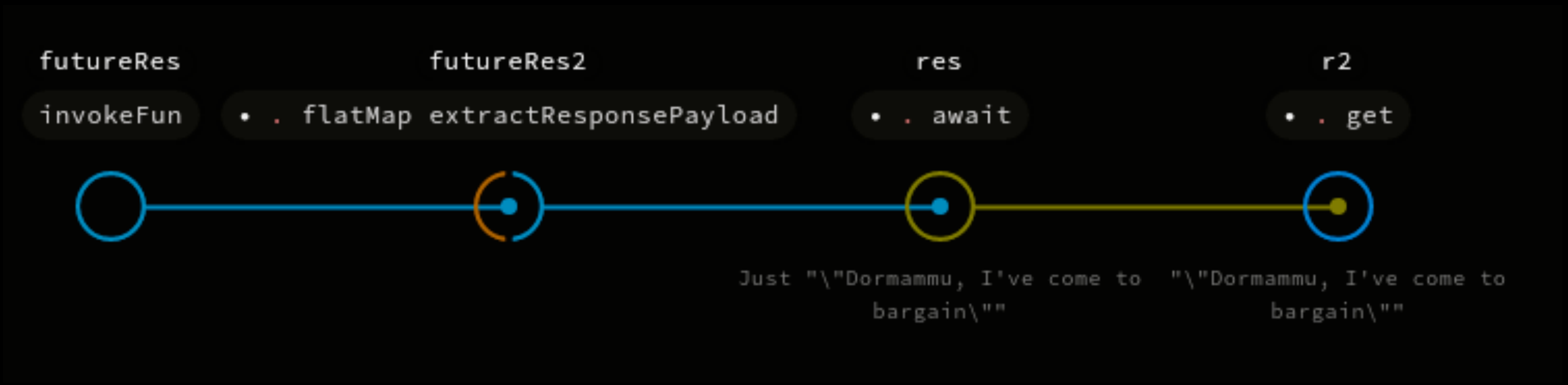
```
▾Object
  payload: ""HelloPeter!""
  class: "LambdaInvokeResponse"
  statusCode: 200
```

Call a remote function similarly to a regular method.
Two flavours:
sync :: `Payload -> Result`
async :: `Payload -> Future Result`

```
payload = JSON.empty . Insert “name” “Peter”
helloFunc.invoke payload
```

Function return value

The functional and async way:



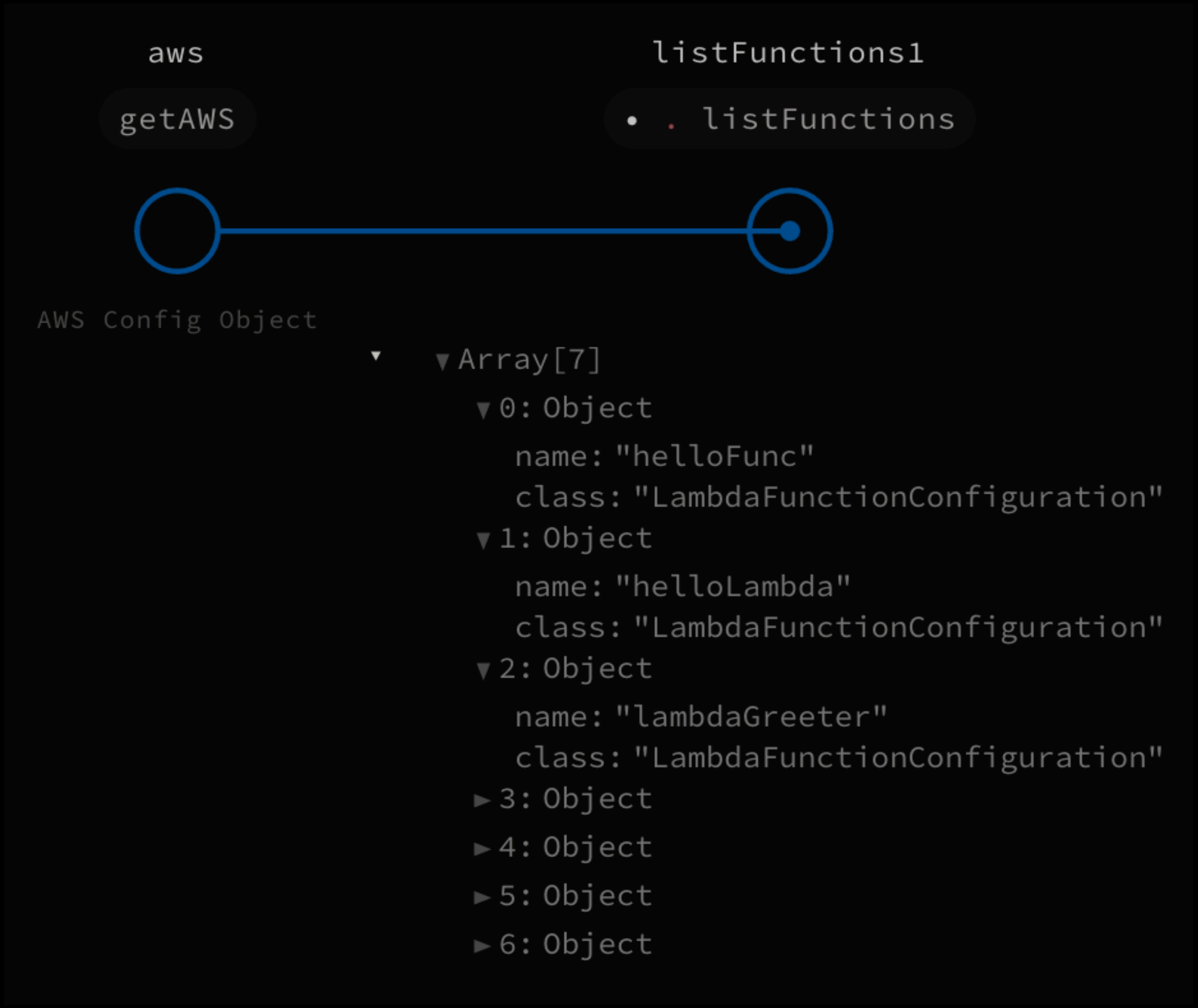
```
futureRes      futureRes2                                res                                      r2
invokeFun      • . flatMap extractResponsePayload        • . await                                • . get

                                                         Just "\"Dormammu, I've come to           "\"Dormammu, I've come to
                                                         bargain\""                               bargain\""
```

We can chain operations on values that are not-quite-there-yet.
(Thank you, **monads!**)

```
futureRes = invokeFun . flatMap extractRP . await . get
```

Utilities

Caching and lookup of remote functions



```
▼Array[7]
  ▼0: Object
     name: "helloFunc"
     class: "LambdaFunctionConfiguration"
  ▼1: Object
     name: "helloLambda"
     class: "LambdaFunctionConfiguration"
  ▼2: Object
     name: "lambdaGreeter"
     class: "LambdaFunctionConfiguration"
  ►3: Object
  ►4: Object
  ►5: Object
  ►6: Object
```

Creating the function many times is the thing to avoid.

Under the hood

- Part of the Luna Language Standard Library
- Crucial parts written in Haskell, API wrapper in Luna
- Leverages the Amazonka library [1] and its Amazonka Lambda extension [2]

[1] http://hackage.haskell.org/package/amazonka
[2] http://hackage.haskell.org/package/amazonka-lambda

Performance

- Performance was not a design goal! (Programmer productivity was)

- Incidentally, the performance is comparable to Haskell and Node.js:

Sync	Luna	Haskell	Node.js
Mean [s]	**35.12**	37.17	37.94
StdDev [s]	**0.38**	0.79	0.81

Async	Luna	Haskell	Node.js
Mean [s]	**34.55**	33.32	34.35
StdDev [s]	**0.81**	2.67	0.58

Future work

- Enable the deployment of functions written in Luna (!)
- Support other cloud providers
- Develop a more sound typing scheme for calls and responses
- Develop a formal model for proving the correctness of Serverless applications

Closing remarks

- Long way to go until serverless functions are supported as a first-class citizens in a programming language but we are getting there.
- Serverless and functional are a promising match!
- Visual solutions for serverless are necessary: a visual language provides that out-of-the-box.

Get in touch!

https://www.icsr.agh.edu.pl/

- GitHub: github.com/luna
- Website: luna-lang.org
- Chat: chat.luna-lang.org

- GitHub: https://github.com/piotrMocz/
- Twitter: https://twitter.com/PMoczurad
- Mail: piotr.moczurad@gmail.com