PhD defense

June 21st 2016, INSA de Lyon, Bâtiment Claude chappe

Etienne Brodu

FLUXIONAL COMPILER

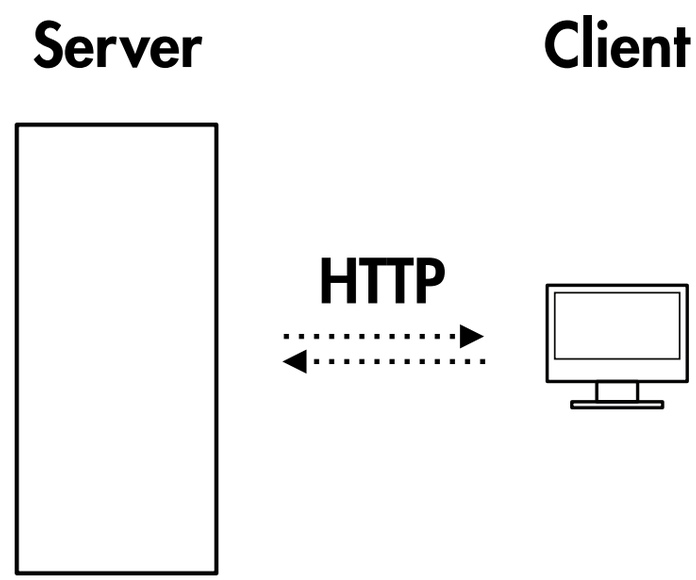
SEAMLESS SHIFT FROM DEVELOPMENT PRODUCTIVITY TO PERFORMANCE EFFICIENCY, IN THE CASE OF REAL-TIME WEB APPLICATIONS

Gaël THOMAS	Professeur, Telecom SudParis, Samovar	Rapporteur
Frédéric LOULERGUE	Professeur, University of Orléans, LIFO	Rapporteur
Floréal MORANDAT	Maître de conférences, Enseirb-Matmeca, LaBRI	Examinateur
Frédéric OBLÉ	Docteur, Atos Worldline	Examinateur
Stéphane FRÉNOT	Professeur, INSA Lyon, CITI	Directeur

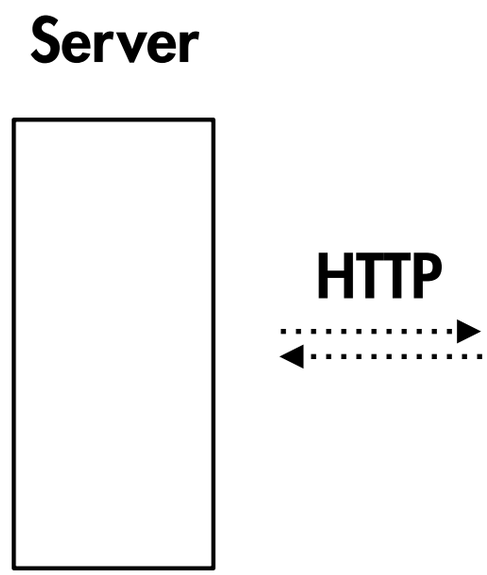
Inria INVENTORS FOR THE DIGITAL WORLD · INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON · citi lab · worldline e-payment services

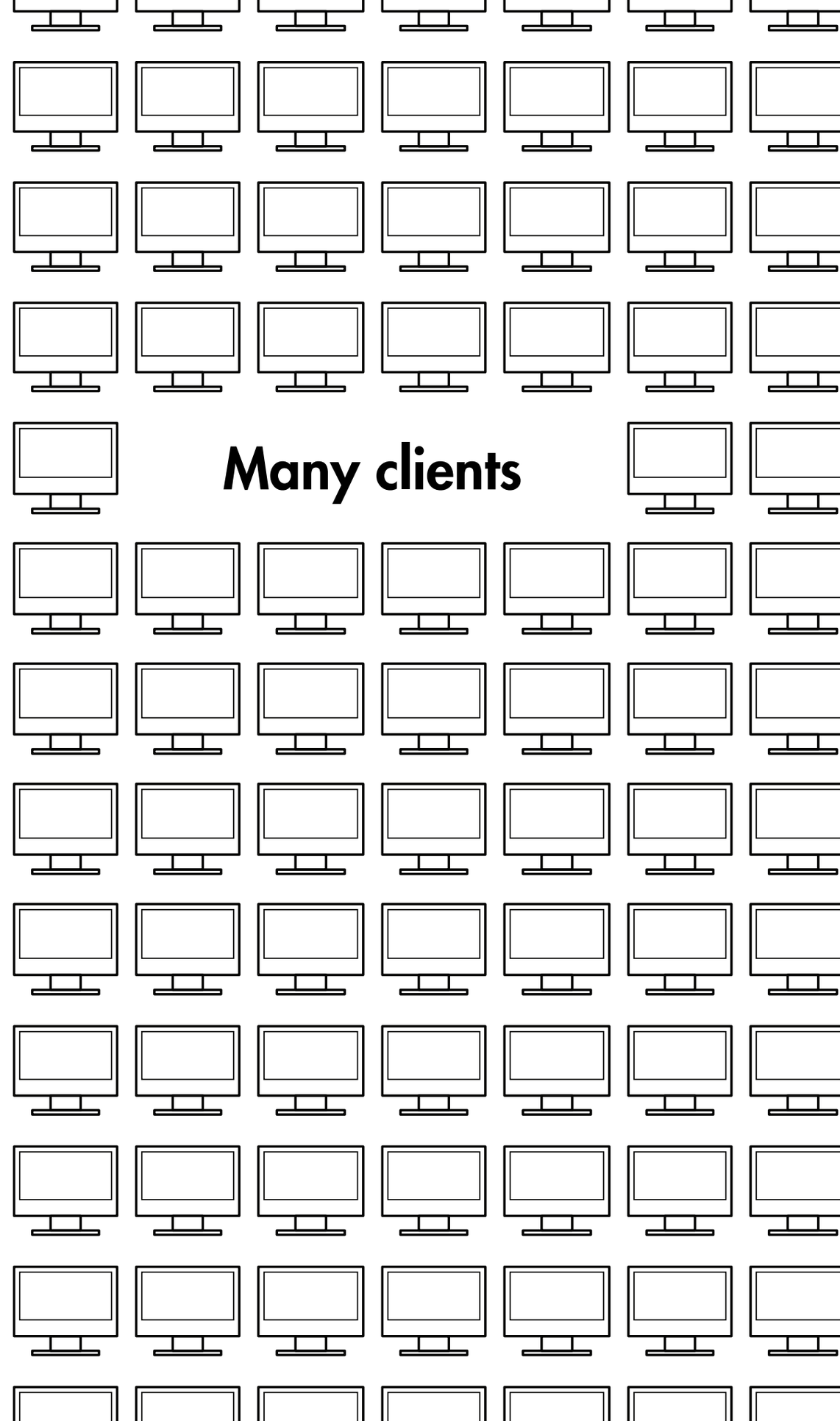
WHAT IS A WEB APPLICATION ?

A program using web technologies to accept requests from remote clients, and responds to them, in soft real-time.



A program using web technologies to accept requests from remote clients, and responds to them, in soft real-time.





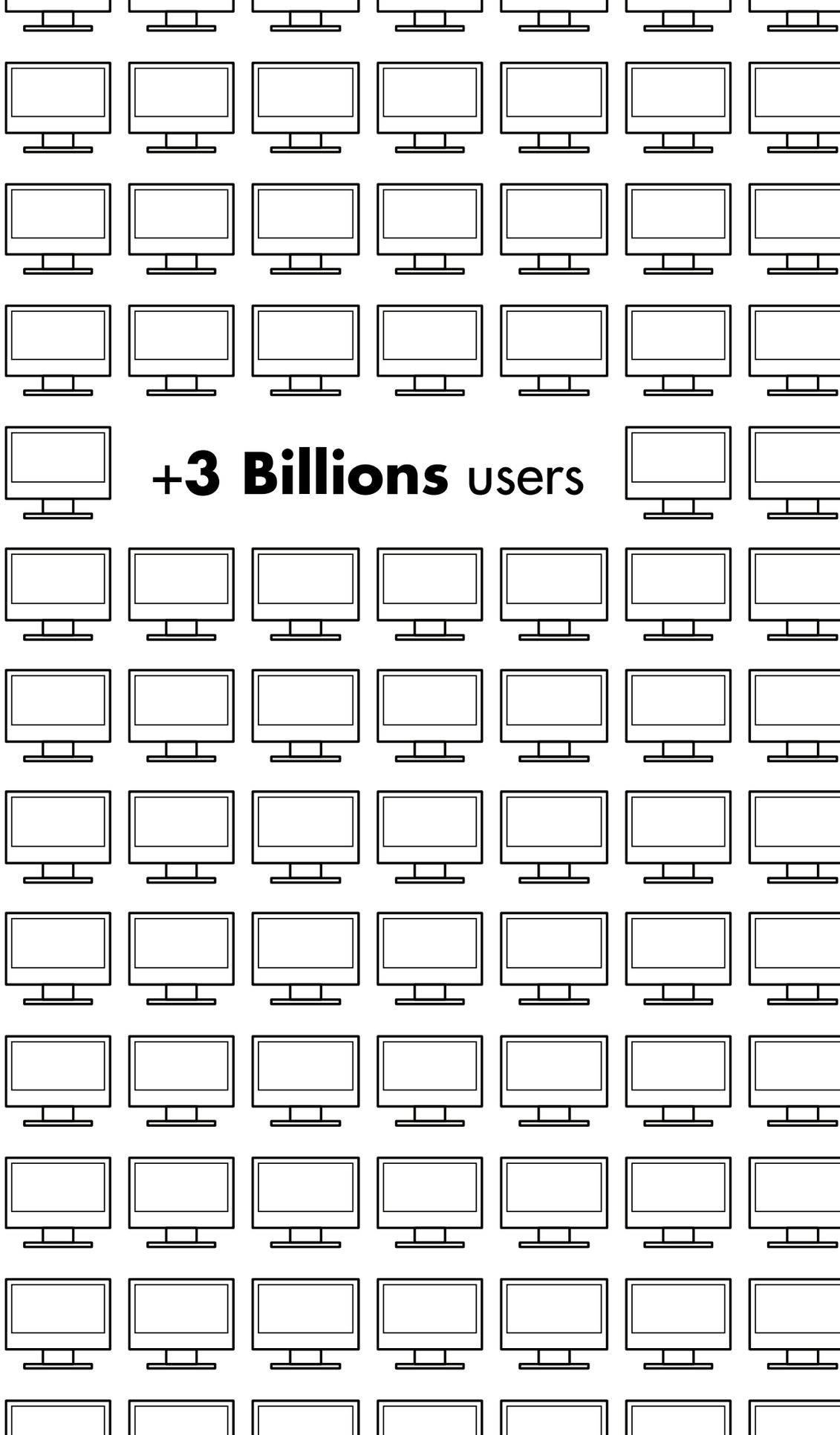
2,890,734 TB
of Internet daily traffic

oogle started 18 years ago,
in a garage, with **1 server**.

4 Billions
searches daily

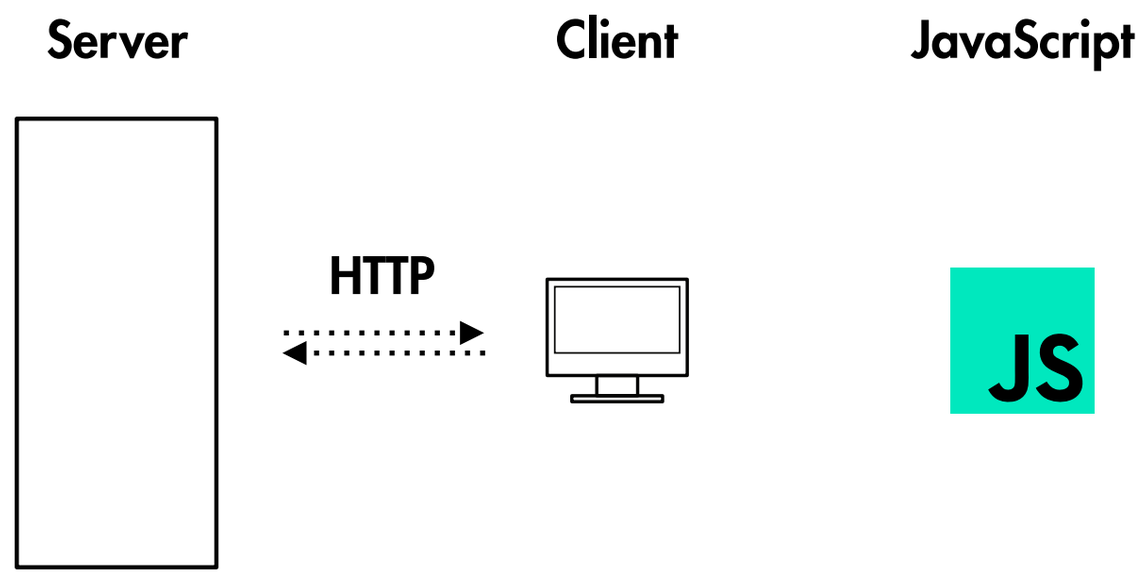
1 Million
servers

This growth is not an exception.



http://www.internetlivestats.com/

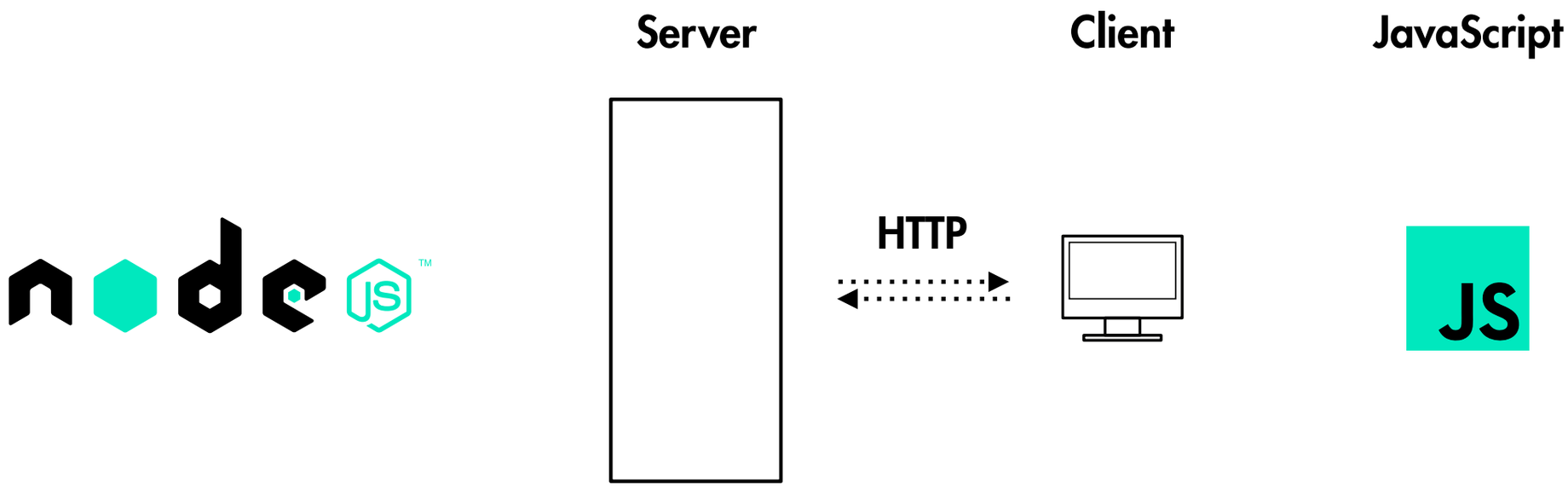
WEB TECHNOLOGIES



Javascript is the only language available natively on the browser.

Follows an event-based paradigm, to manage the stream of user interactions.

WEB TECHNOLOGIES



Available on the server since 2009 with node.js.

The event-based paradigm is well suited to handle the stream of requests for web applications.

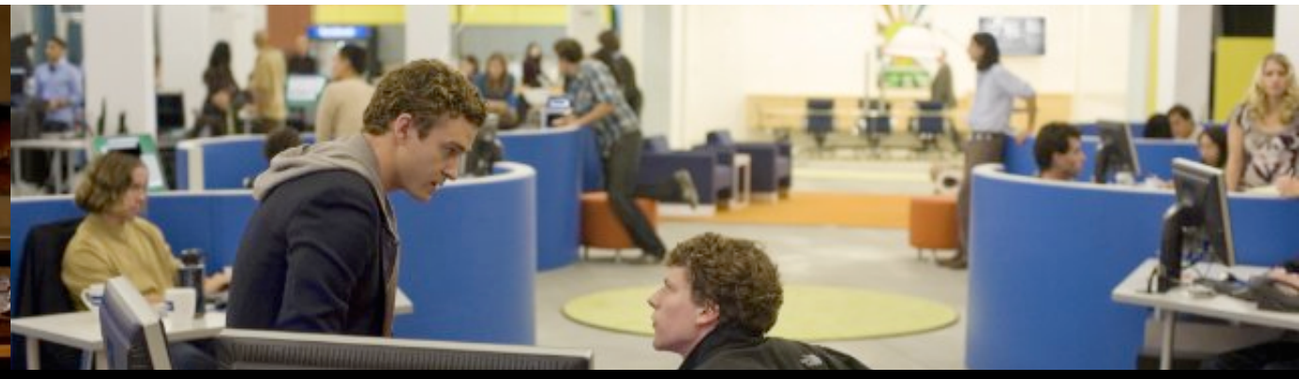
WEB APPLICATION DEVELOPMENT

development productivity

or

execution efficiency

1k users

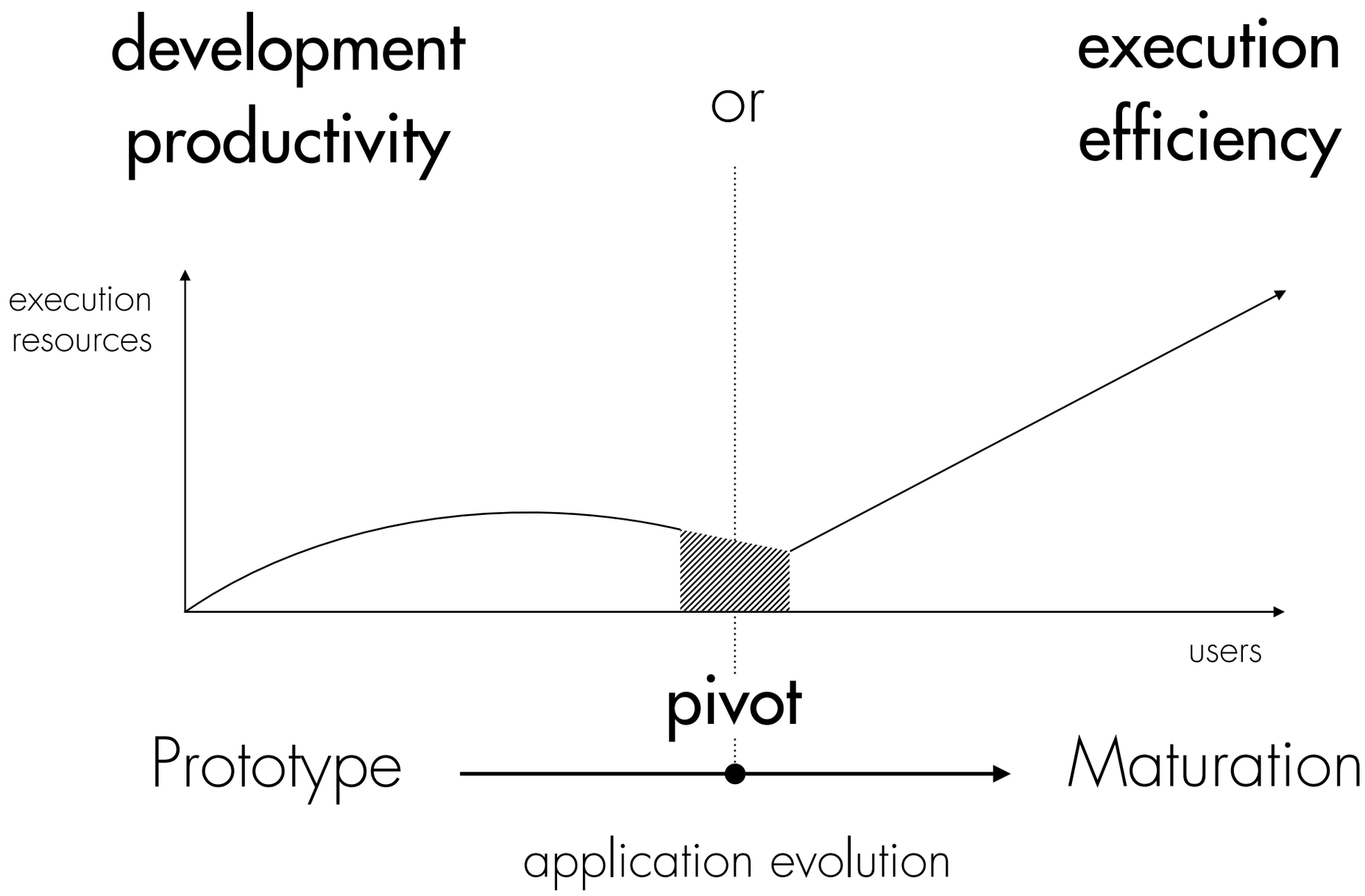


1B users

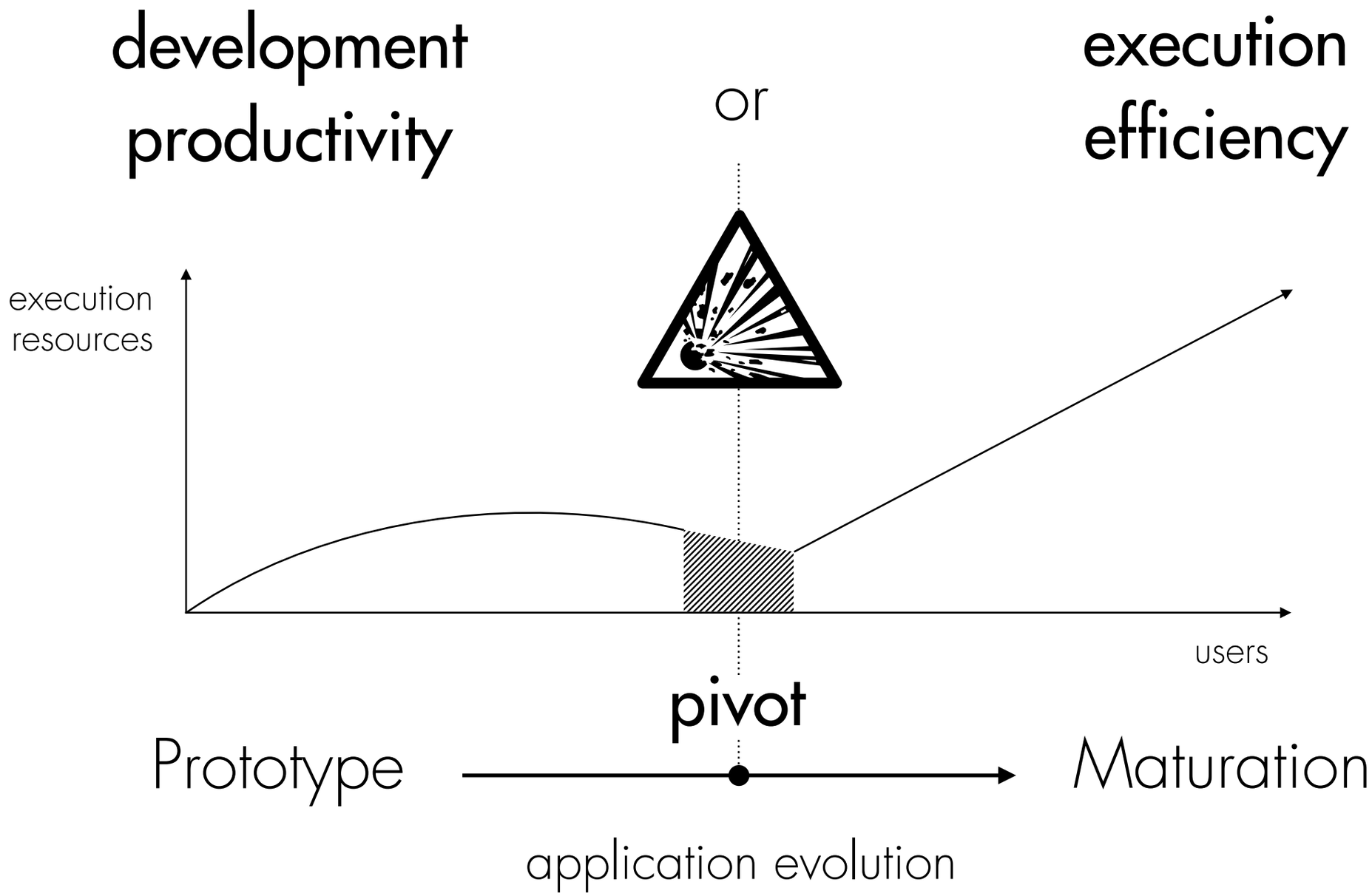
Prototype → Maturation

application evolution

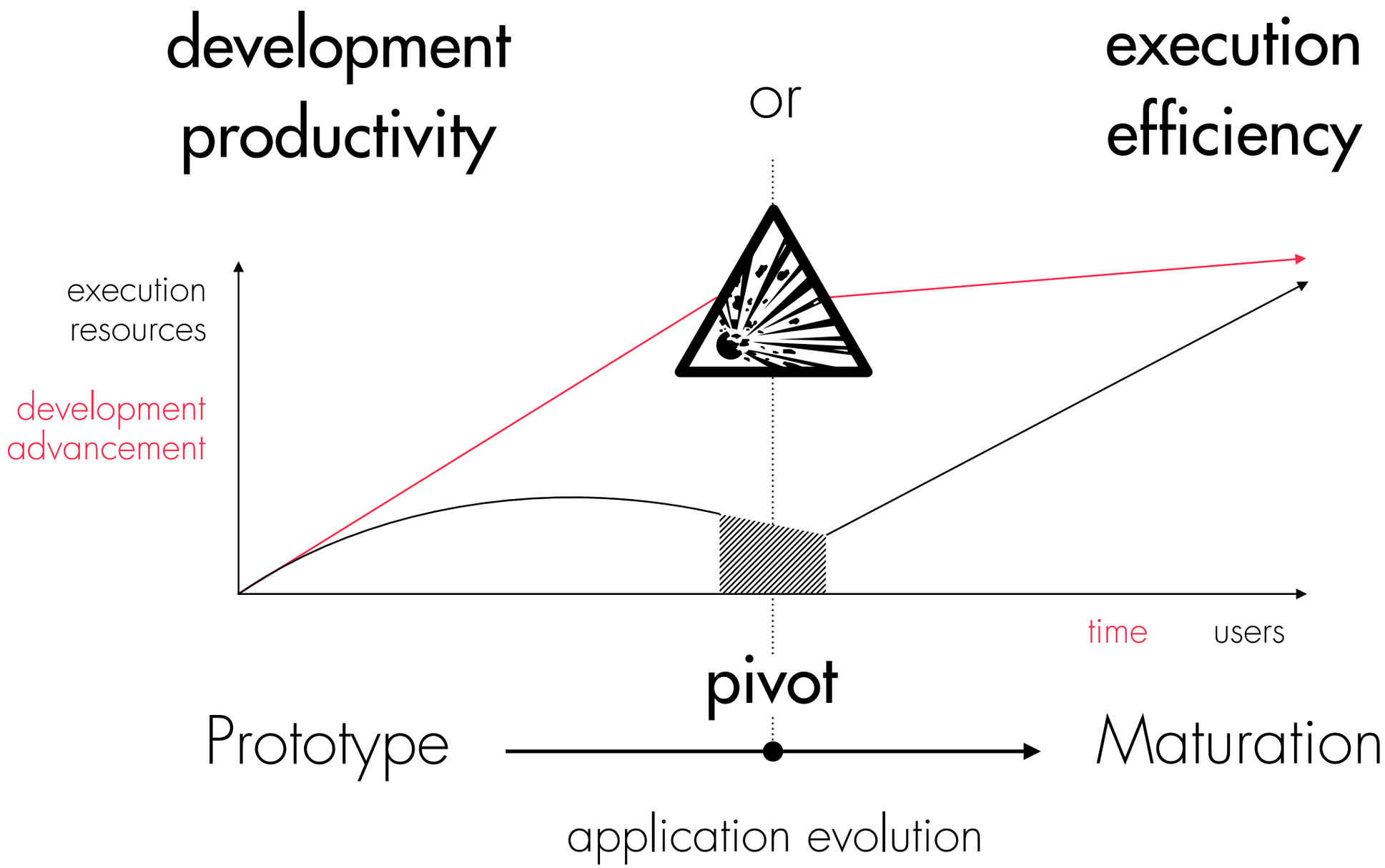
THE PIVOT DILEMMA



THE PIVOT DILEMMA



THE PIVOT DILEMMA



THE PIVOT DILEMMA

development productivity ≠ execution efficiency

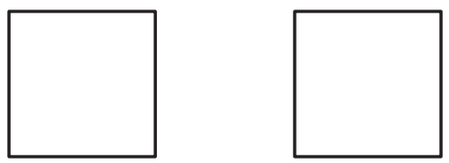
The pivot is due to the incompatibility between development productivity and execution efficiency.

THE PIVOT DILEMMA

development productivity ≠ execution efficiency

The pivot is due to the incompatibility between development productivity and execution efficiency.

Modularity

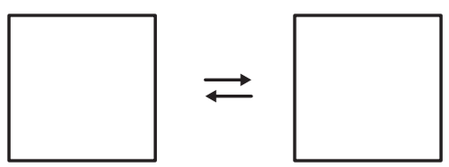


THE PIVOT DILEMMA

development productivity ≠ execution efficiency

The pivot is due to the incompatibility between development productivity and execution efficiency.

Composition

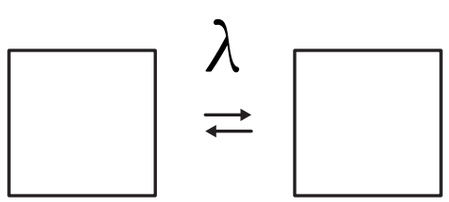


THE PIVOT DILEMMA

development productivity ≠ execution efficiency

The pivot is due to the incompatibility between development productivity and execution efficiency.

Higher-order Programming



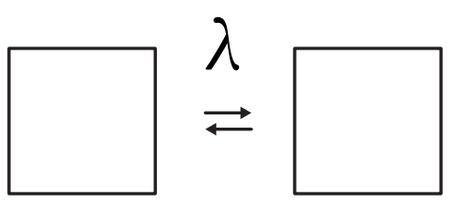
Mutability **or** Immutability

THE PIVOT DILEMMA

development productivity ≠ execution efficiency

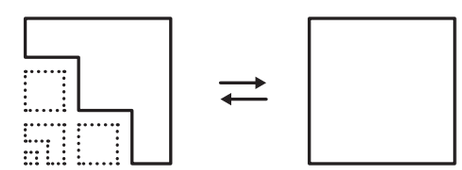
The pivot is due to the incompatibility between development productivity and execution efficiency.

Higher-order Programming



Mutability **or** Immutability

Parallelism



Mutability **and** Immutability

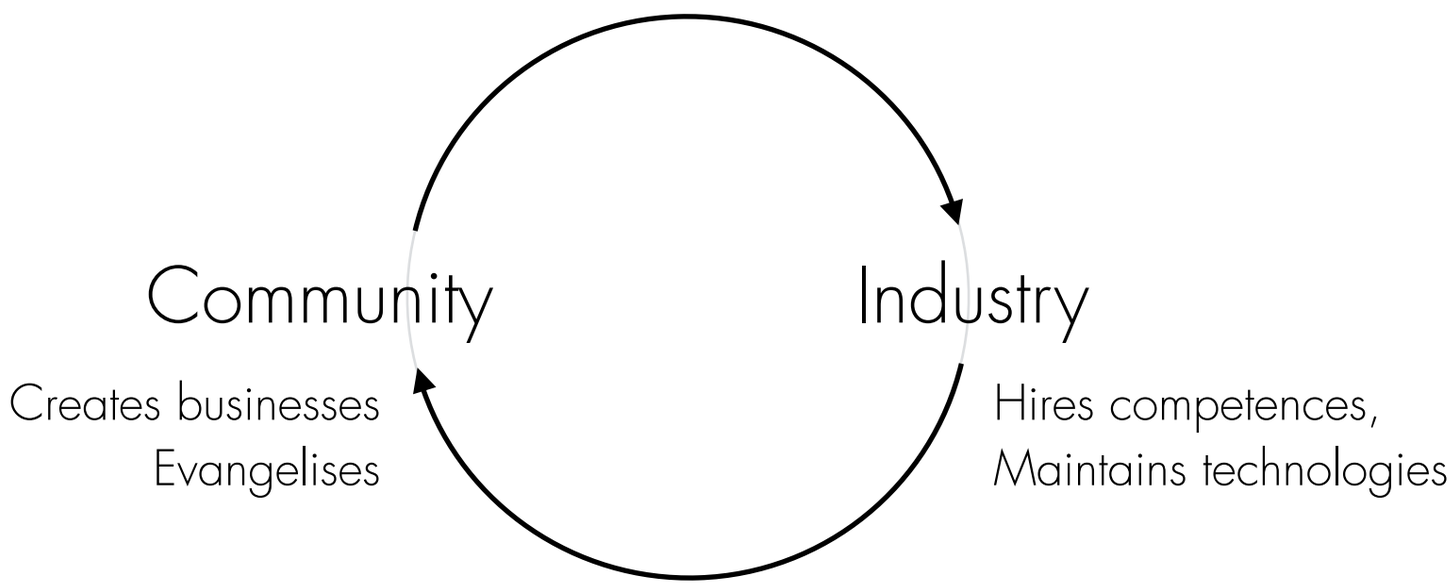
at fine grain — at coarse grain

THE PIVOT DILEMMA

Adoption

requires

development productivity + execution efficiency



STATE OF THE ART

Model	Implementations	Productivity	Efficiency	Adoption
Imperative Programming	Fortran, Algol, Cobol and C	3	1	2
Object-Oriented Programming	C++ and Java	4	1	4
Functional Programming	Scheme, Miranda, Haskell …	4	1	1
Multi Paradigm	Javascript, Python, Ruby and Scala	5	1	3
Event-driven programming	TAME, Node.js and Vert.X	5	2	5
Lock-free Data-Structures	linked list, queue, tree …	5	2	0
Multi-threading programming	semaphores, guarded commands …	4	1	3
Hybrid Models	Fibers, Capriccio …	4	1	0
Actor Model	Erlang, Scala Actors, Akka …	2	5	1
Communicating Sequential Processes	Go	2	5	1
Data Stream System Management	DryadLINQ, Apache Hive, Timestream, Shark …	2	5	1
Pipeline Stream Processing	SEDA, StreaMIT, Spark Streaming, Storm …	2	5	1

0 1 2 3 4 5

1

THE PIVOT DILEMMA

The risk of failing in the transition from development productivity to execution efficiency.

2

THE STATE OF THE ART

Focus either on one or the other.
Or sacrifices productivity to ease the transition.

3

THE NODE.JS PARTICULARITY

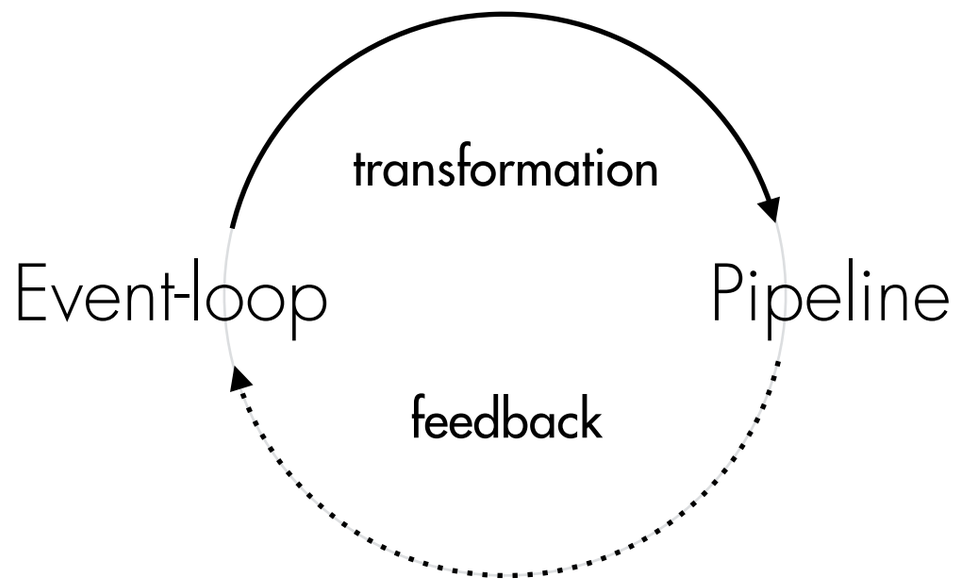
Javascript is in a unique position, strengthening its adoption.
The event-loop execution model is close to a pipeline

Proposition

ELIMINATE THE PIVOT REQUIREMENT

Liquid IT from worldline e-payment services

development productivity

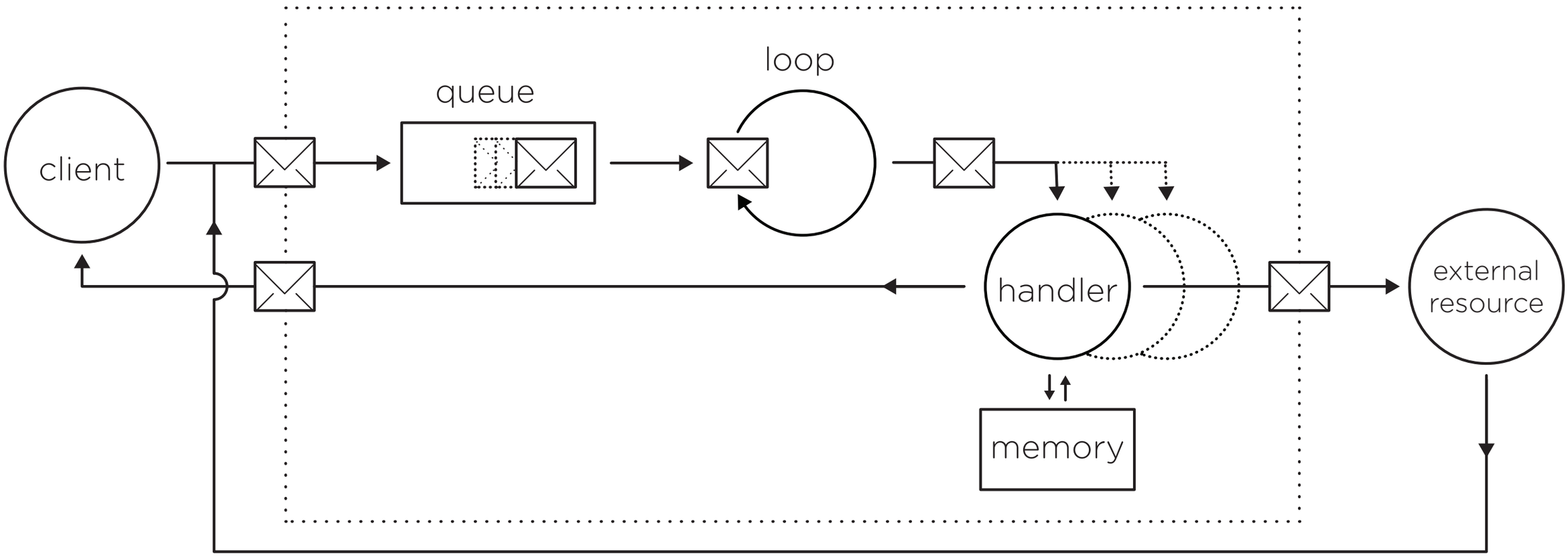


execution efficiency

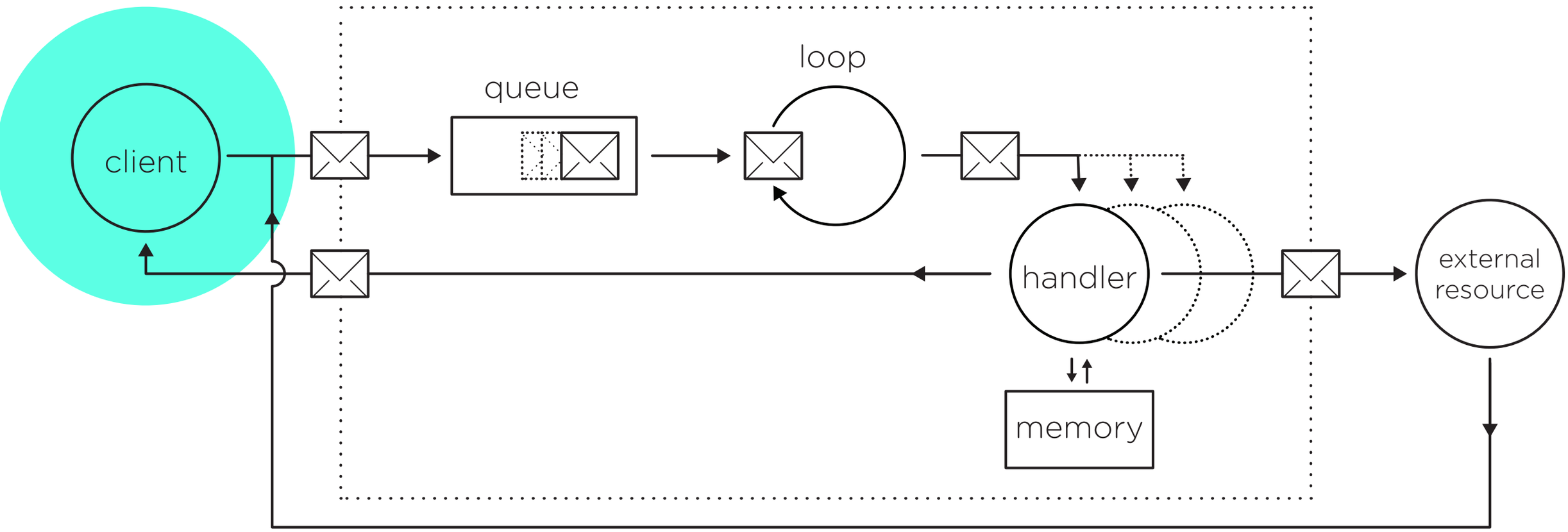
1. **FLUXIONAL EXECUTION MODEL**
   Compatible with both programming models

2. **FLUXIONAL COMPILER**
   From one programming model to the other

EVENT-LOOP

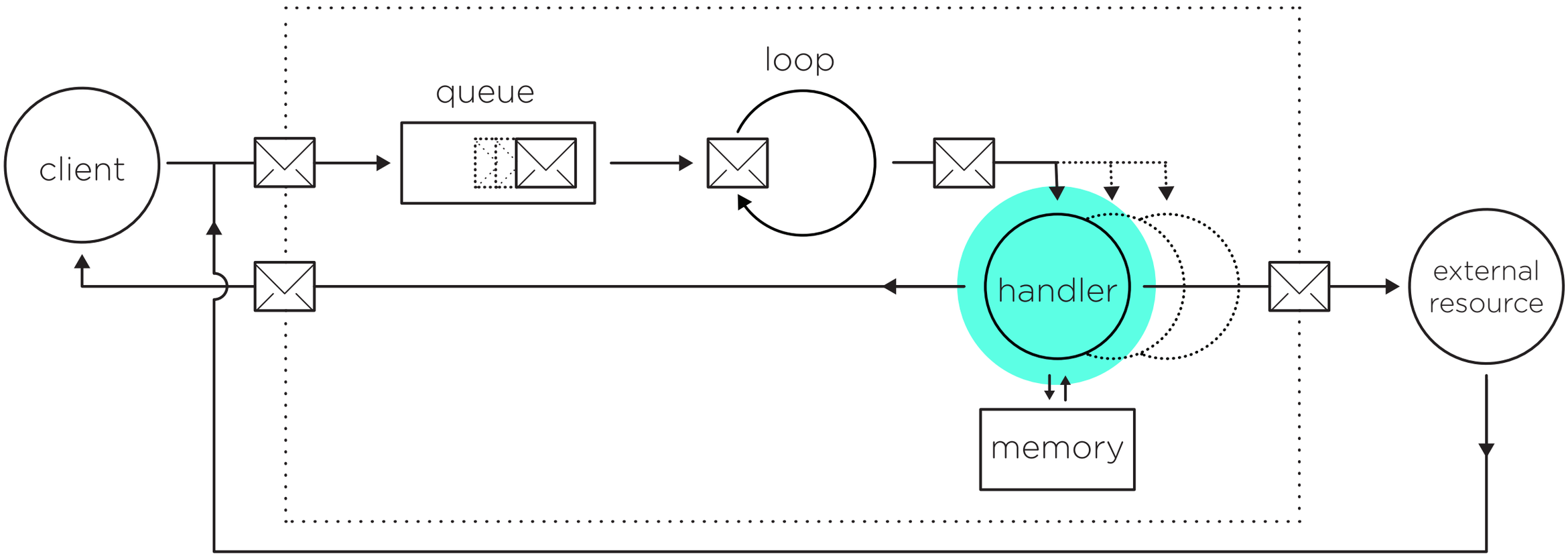


EVENT-LOOP



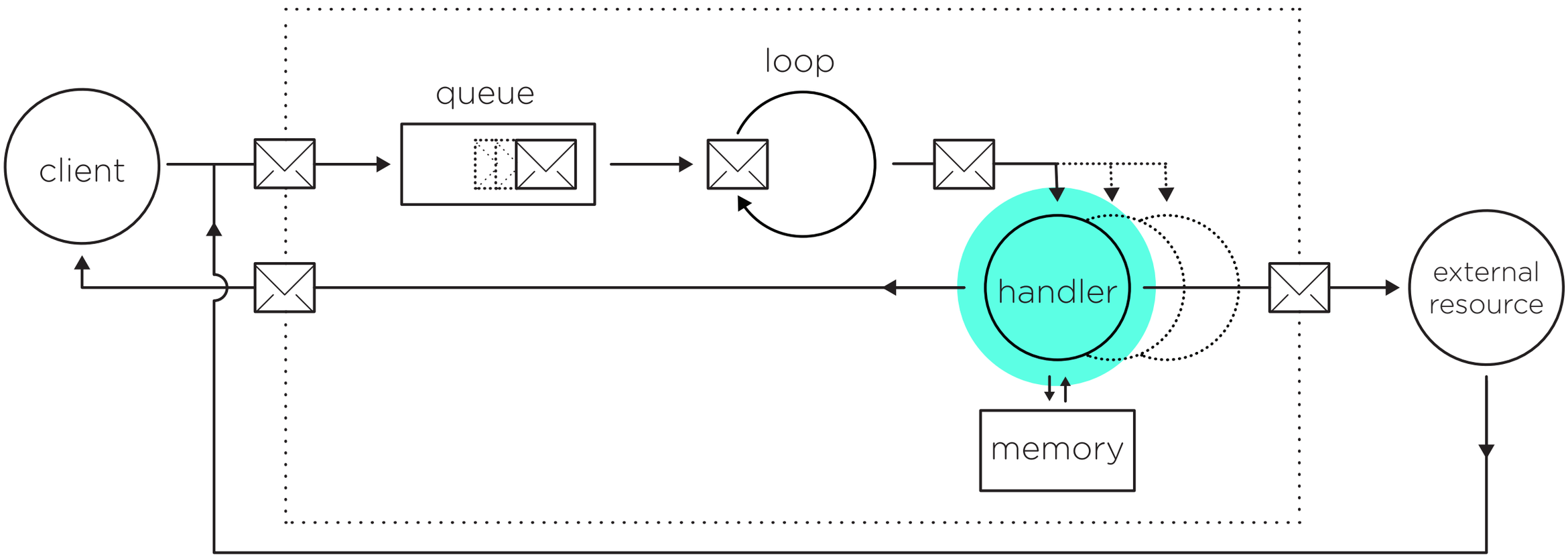
A client sends a request.

EVENT-LOOP



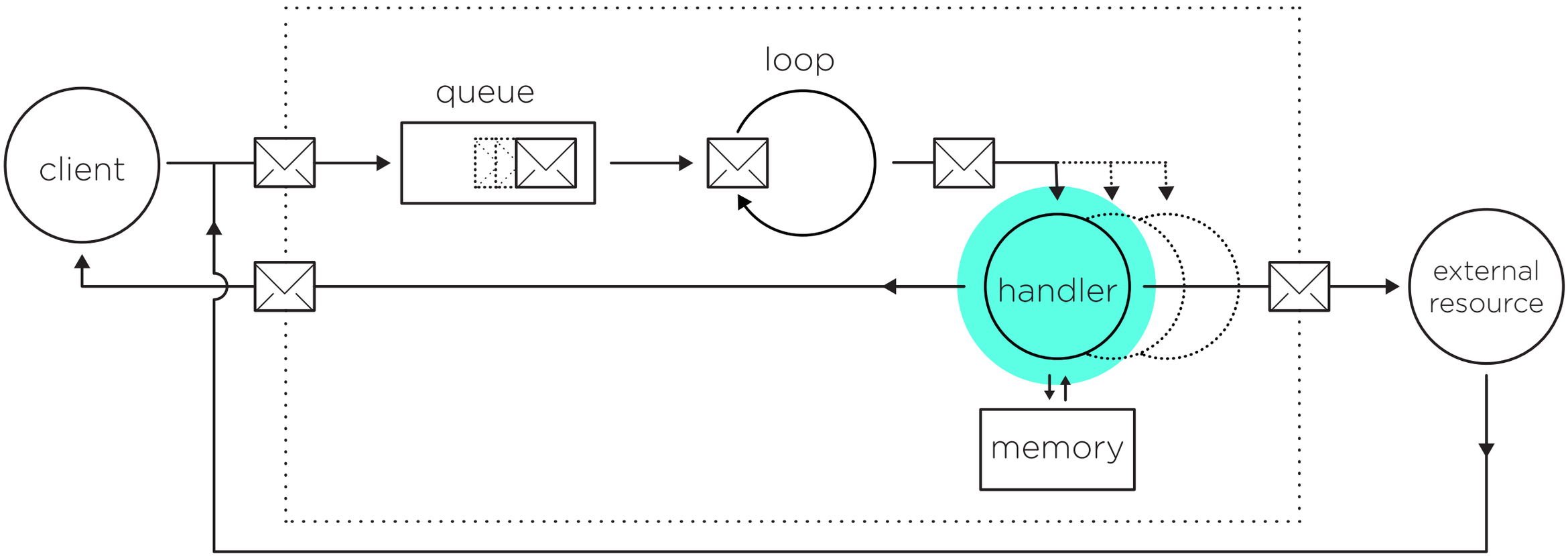
The handler of this request has exclusivity on the memory.

EVENT-LOOP



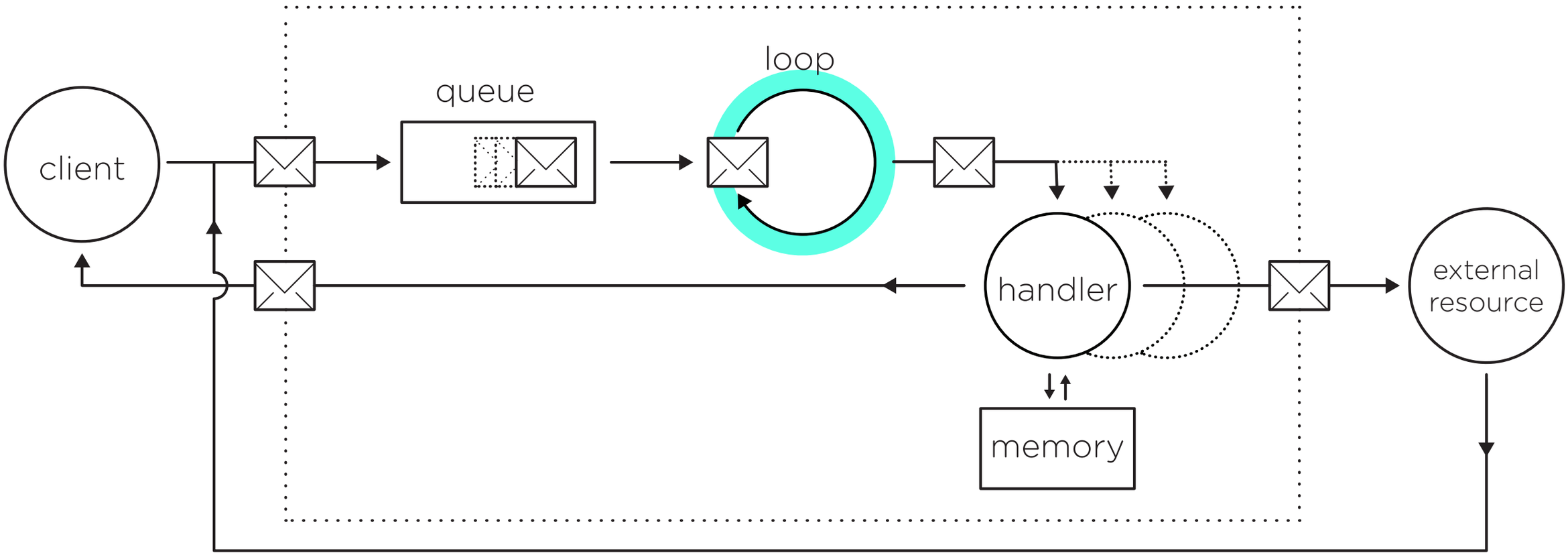
The handler can call an external resource, and yield execution to another handler.

EVENT-LOOP

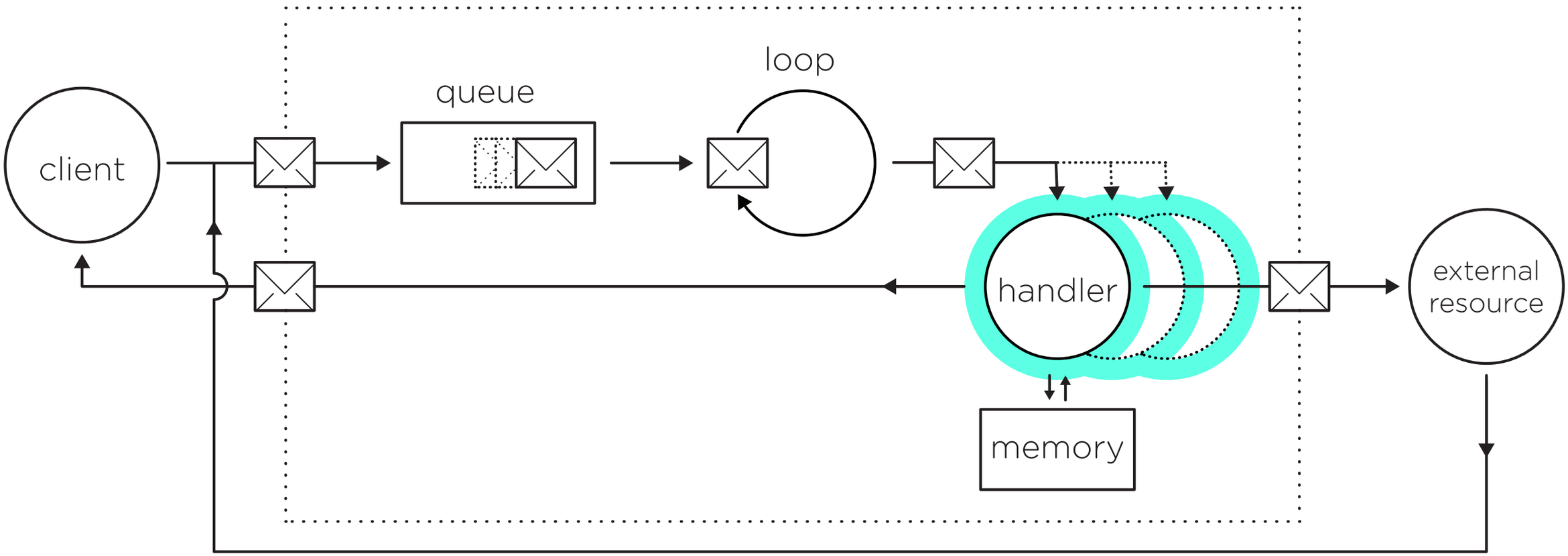


The final handler will answer back to the client, closing the loop.

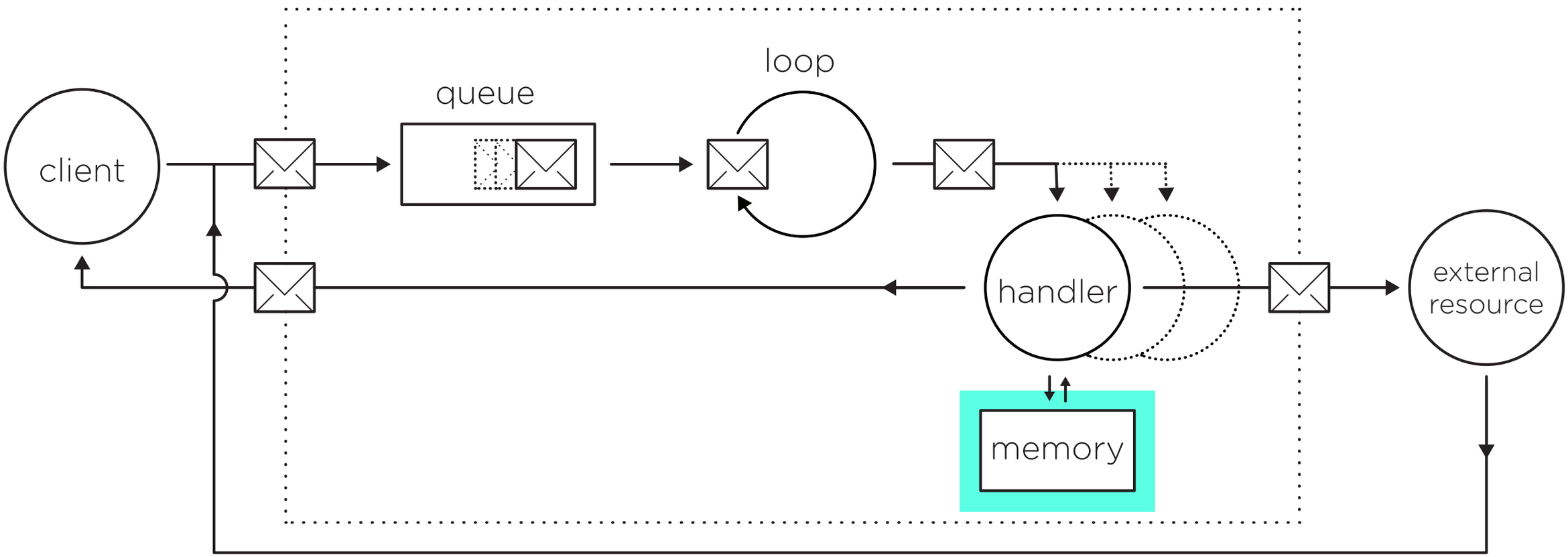
EVENT-LOOP



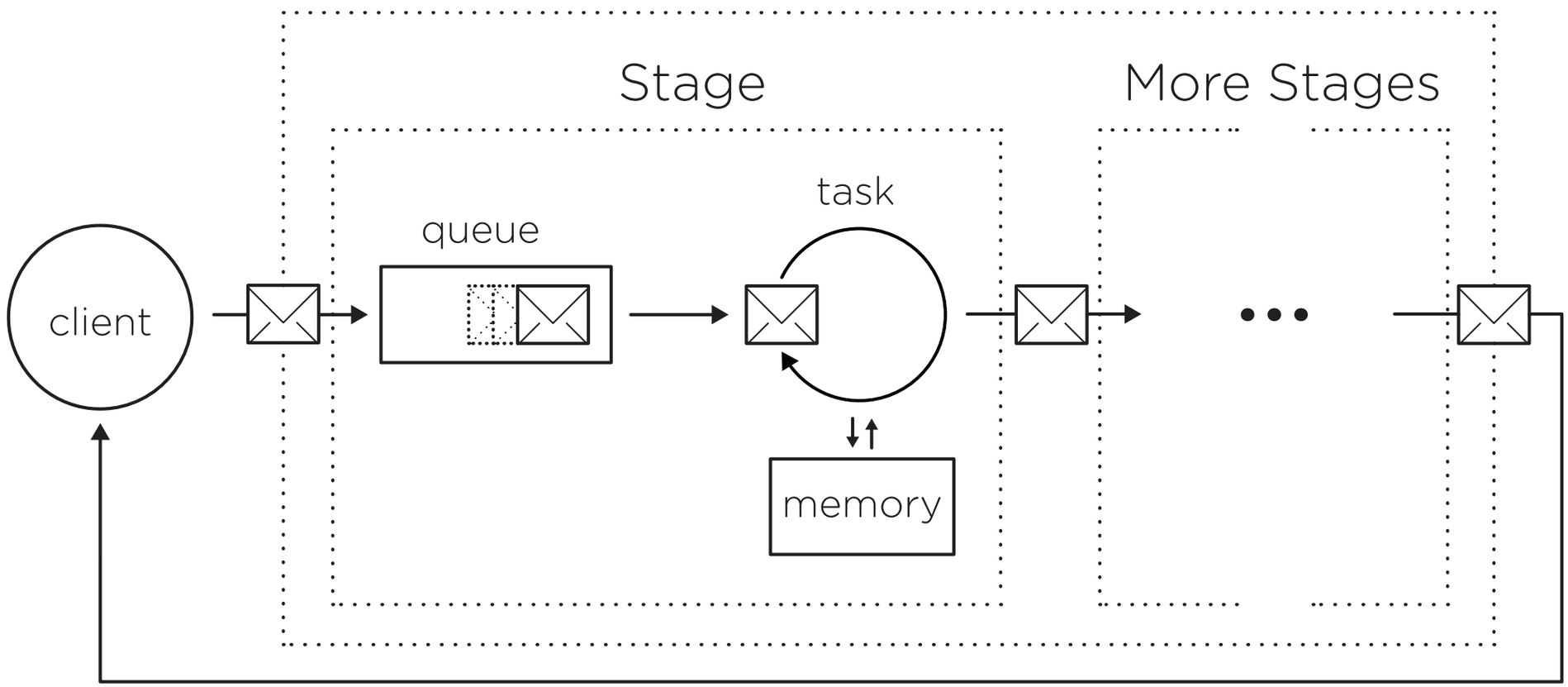
EVENT-LOOP



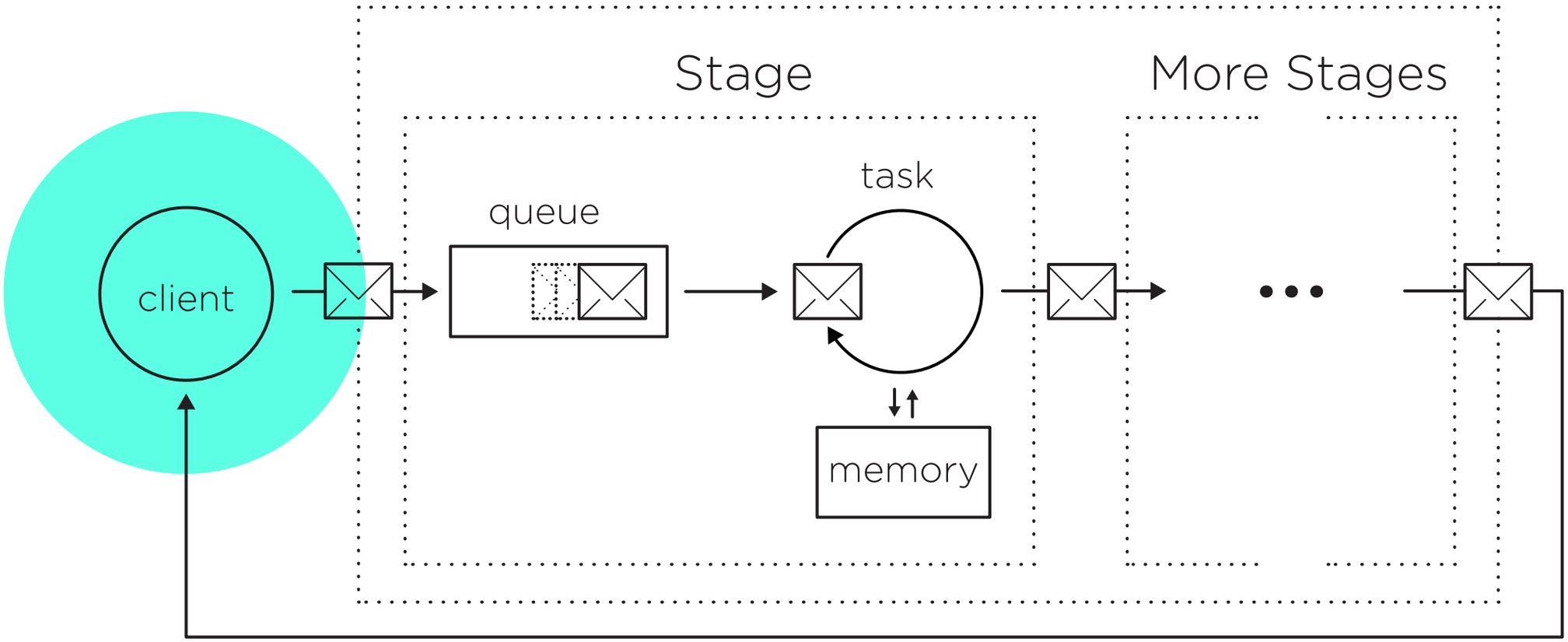
EVENT-LOOP



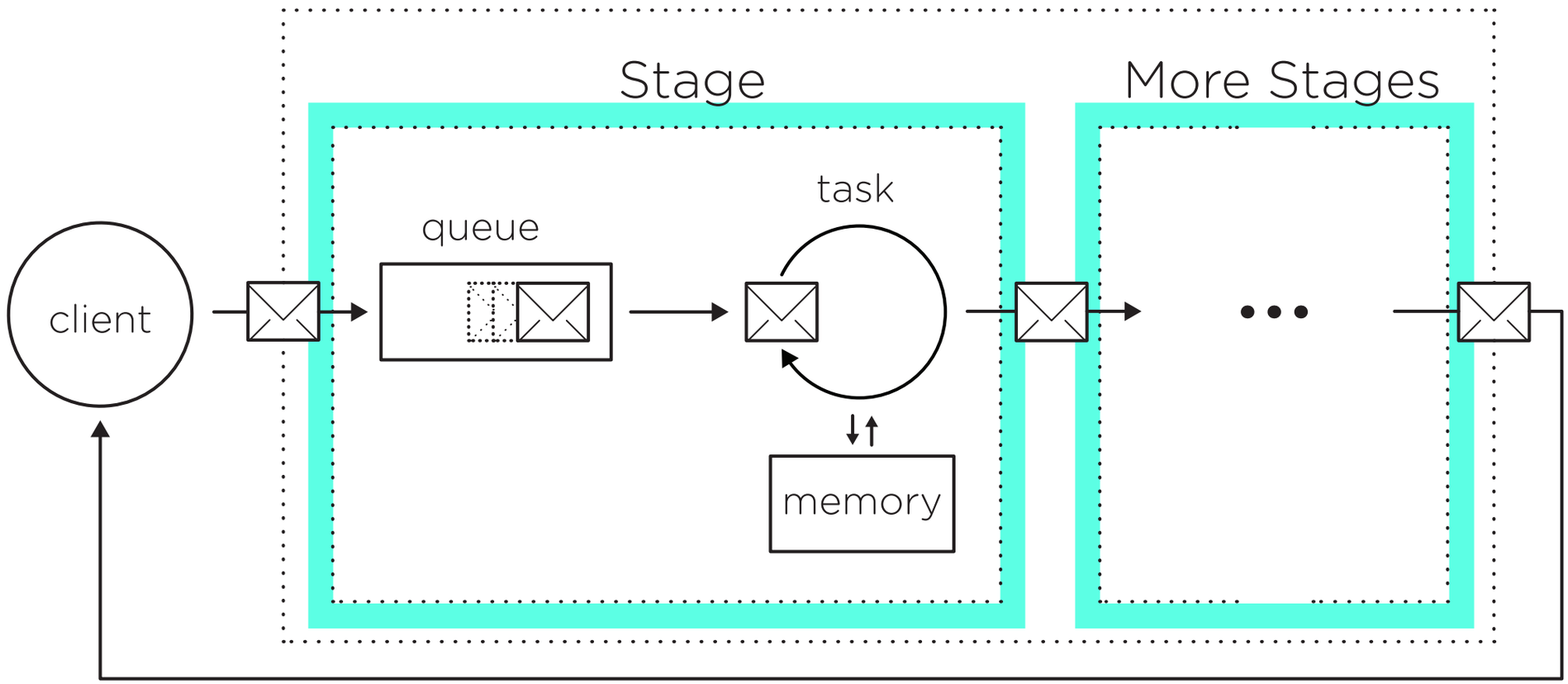
PIPELINE



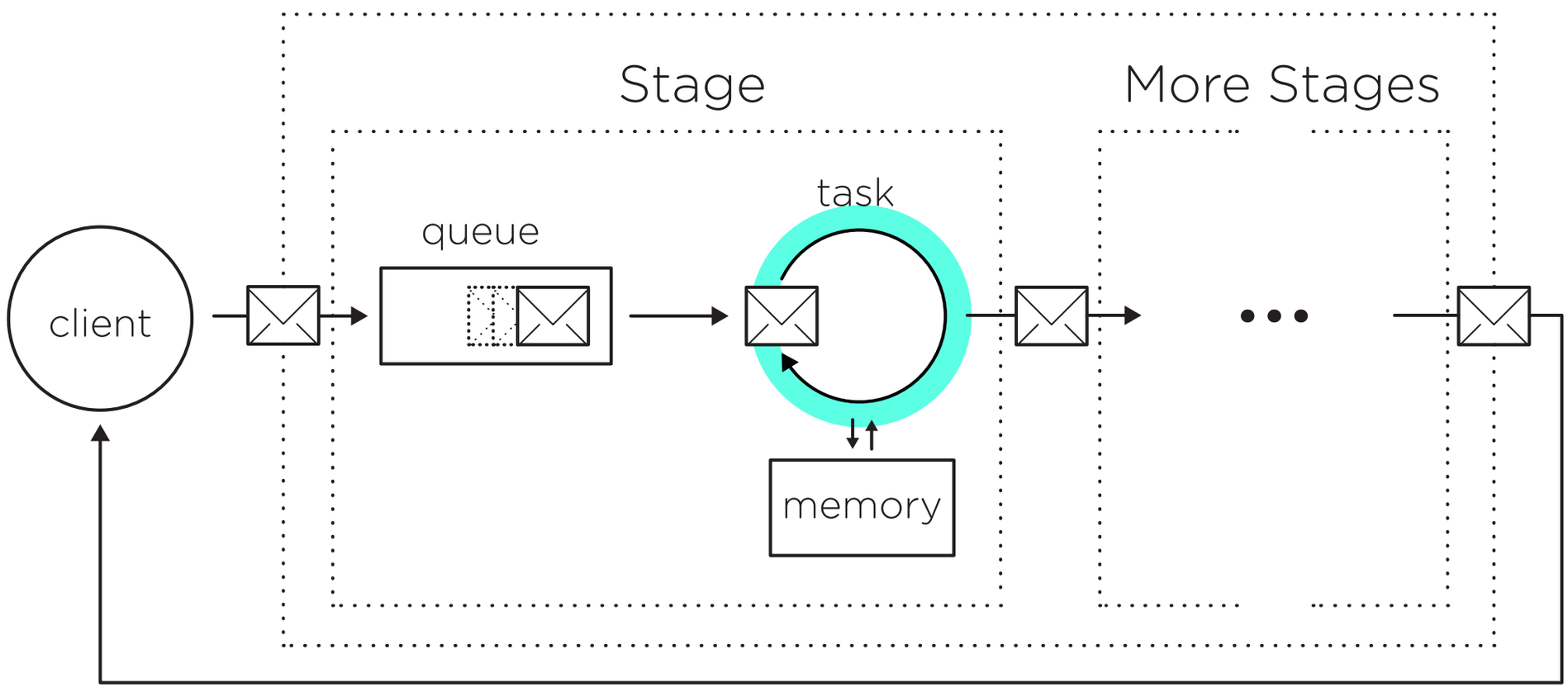
PIPELINE



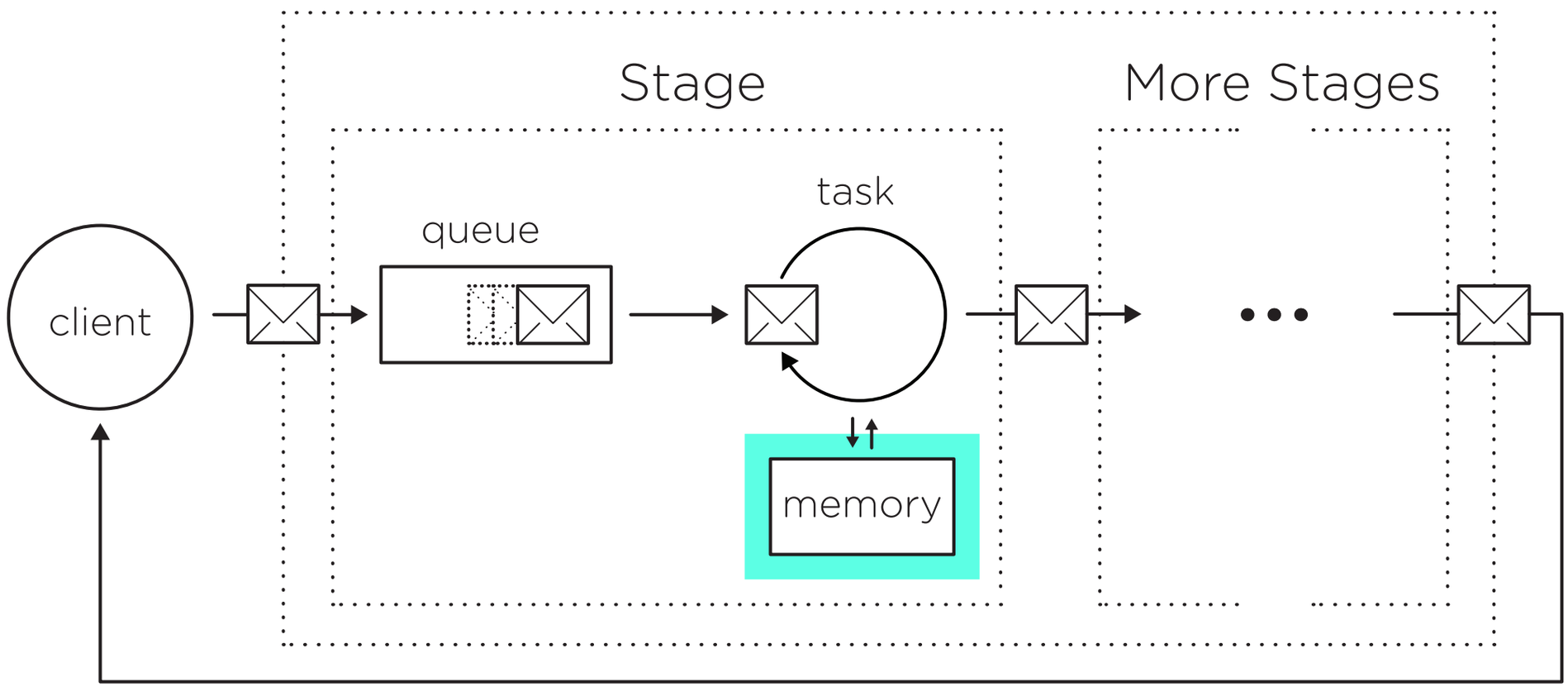
PIPELINE



PIPELINE

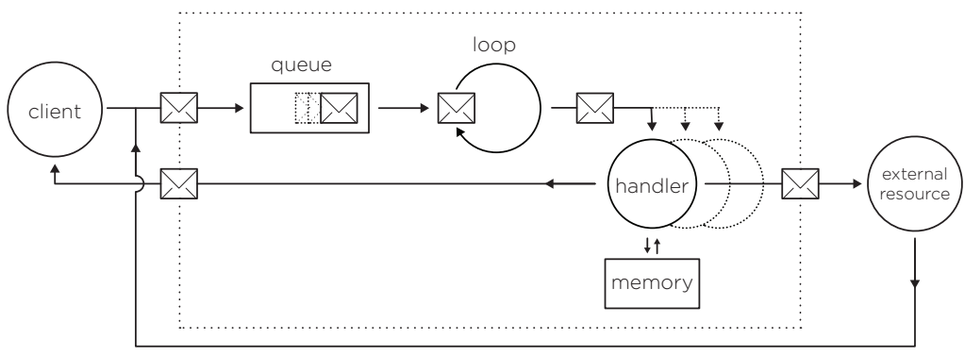


PIPELINE



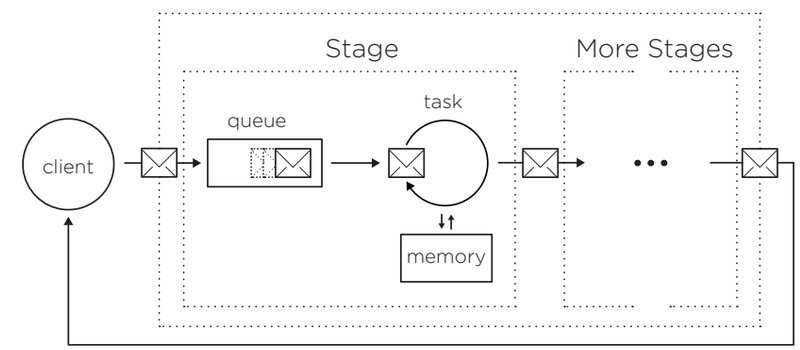
FLUXIONAL EXECUTION MODEL

Event-loop



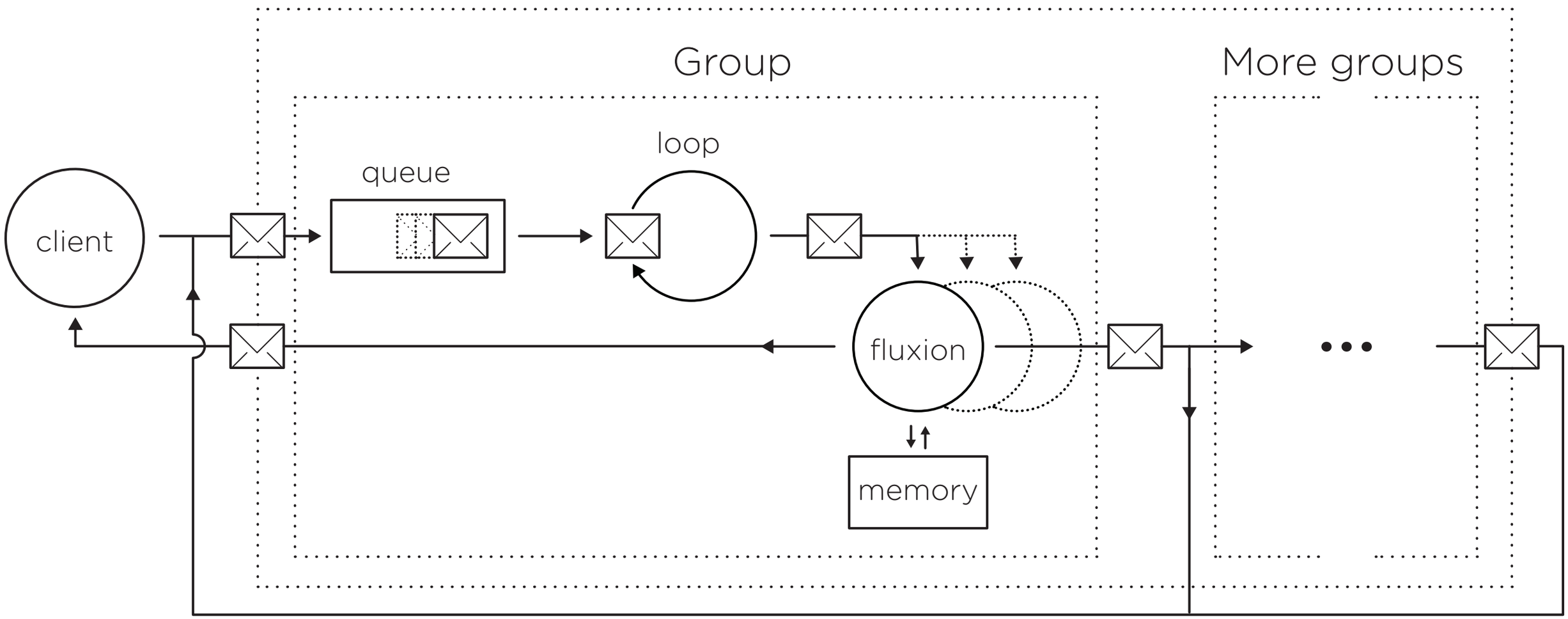
+

Pipeline

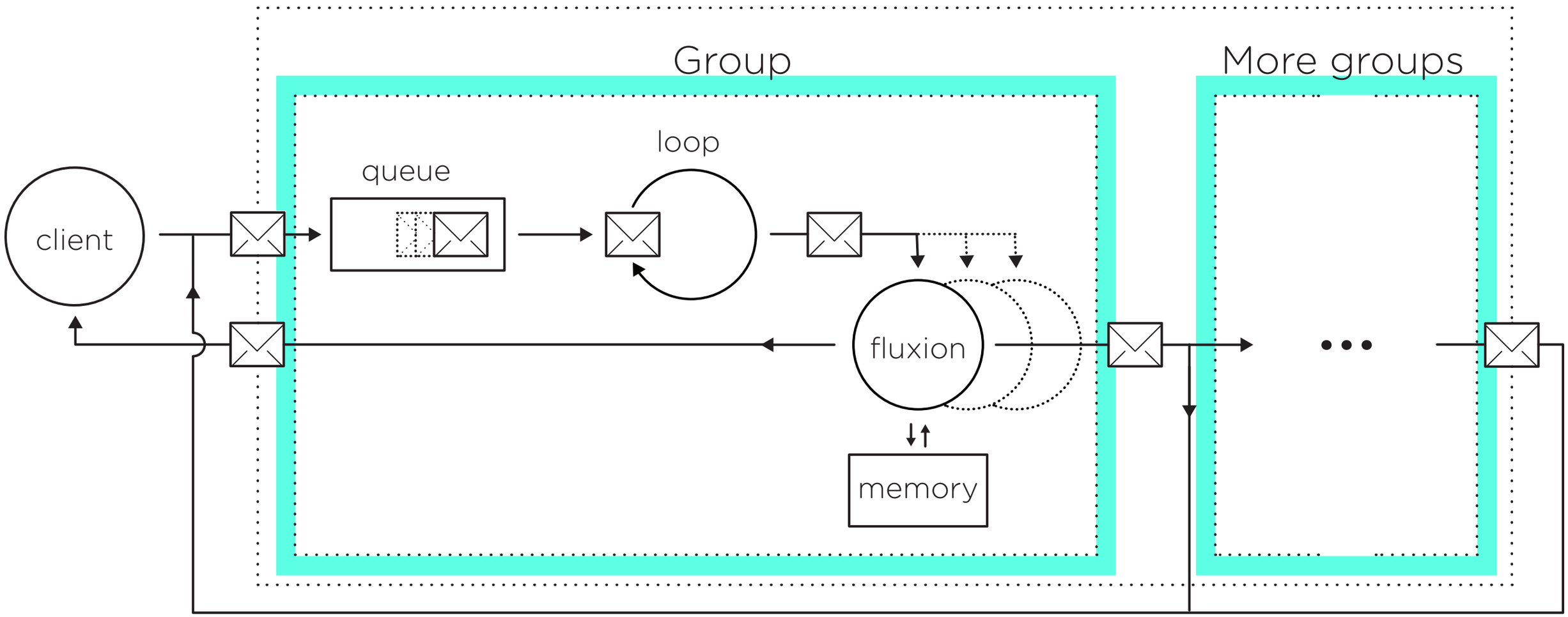


The fluxional execution model executes both programs targeting event-loop and programs targeting pipeline.

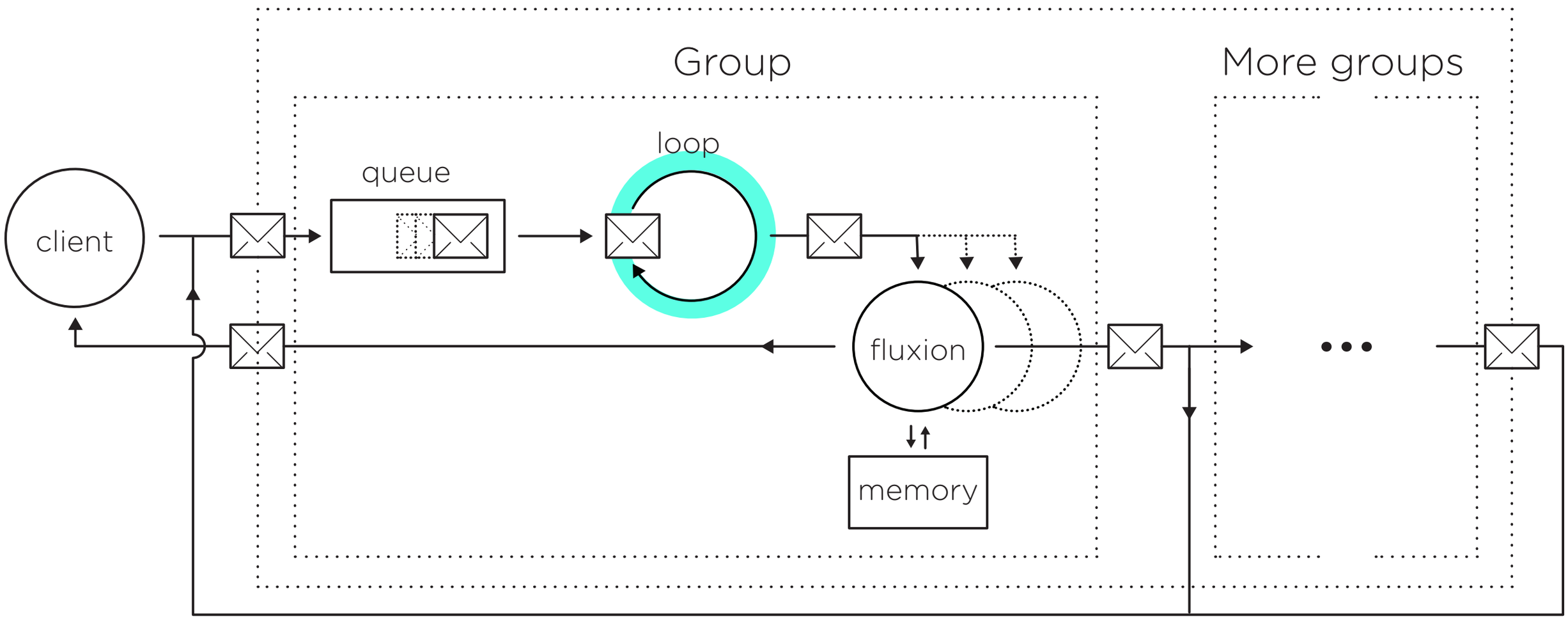
FLUXIONAL EXECUTION MODEL



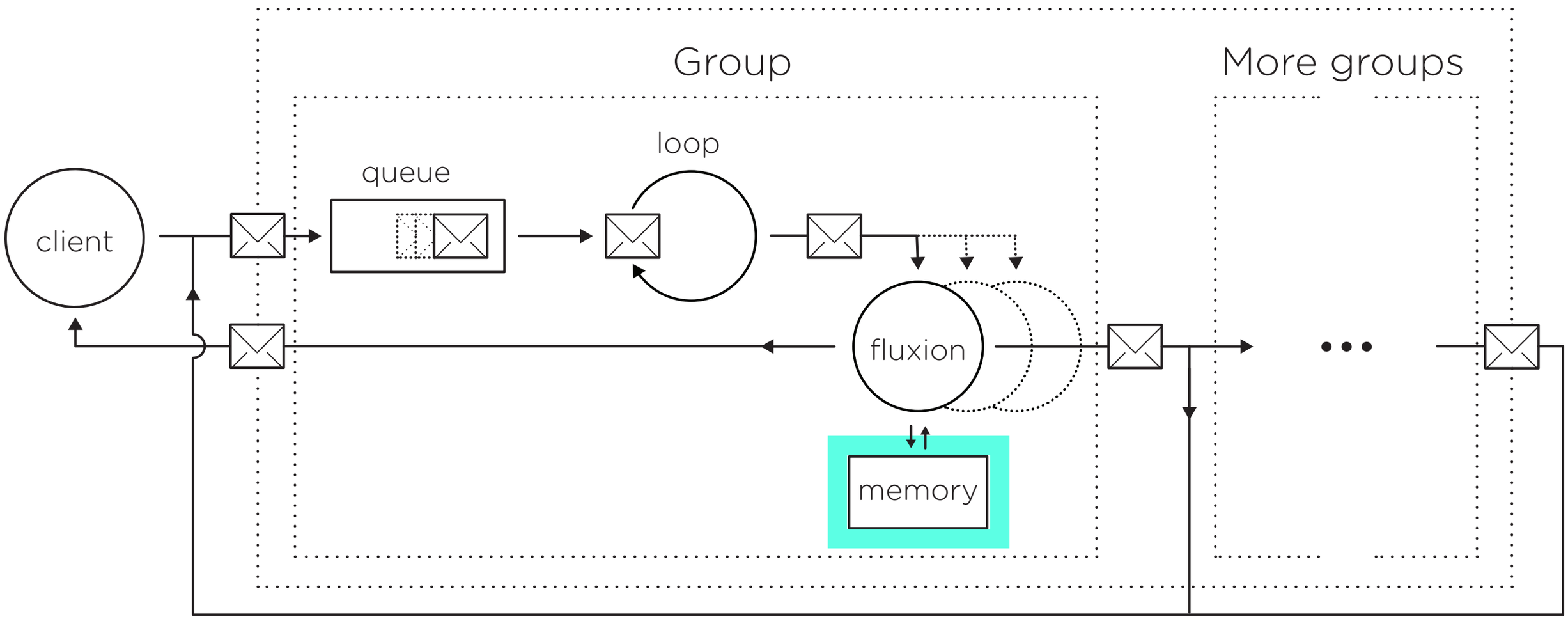
FLUXIONAL EXECUTION MODEL



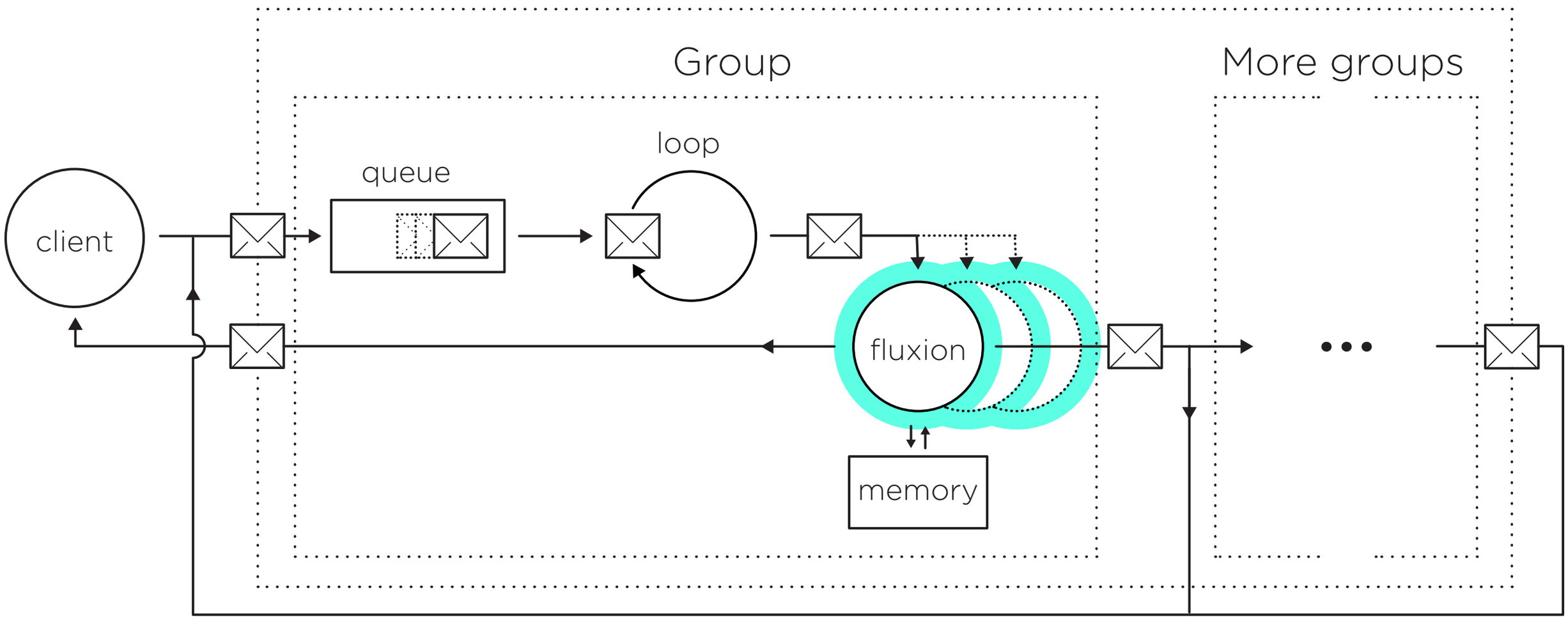
FLUXIONAL EXECUTION MODEL



FLUXIONAL EXECUTION MODEL

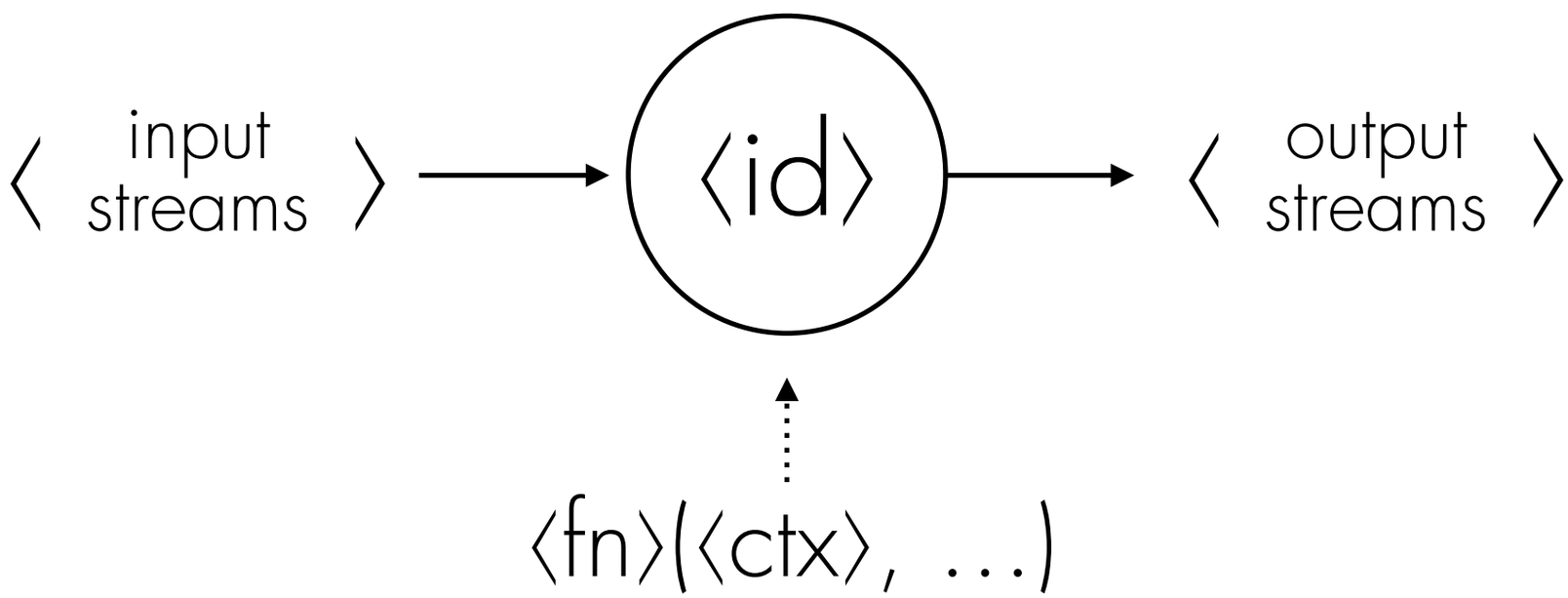


FLUXIONAL EXECUTION MODEL



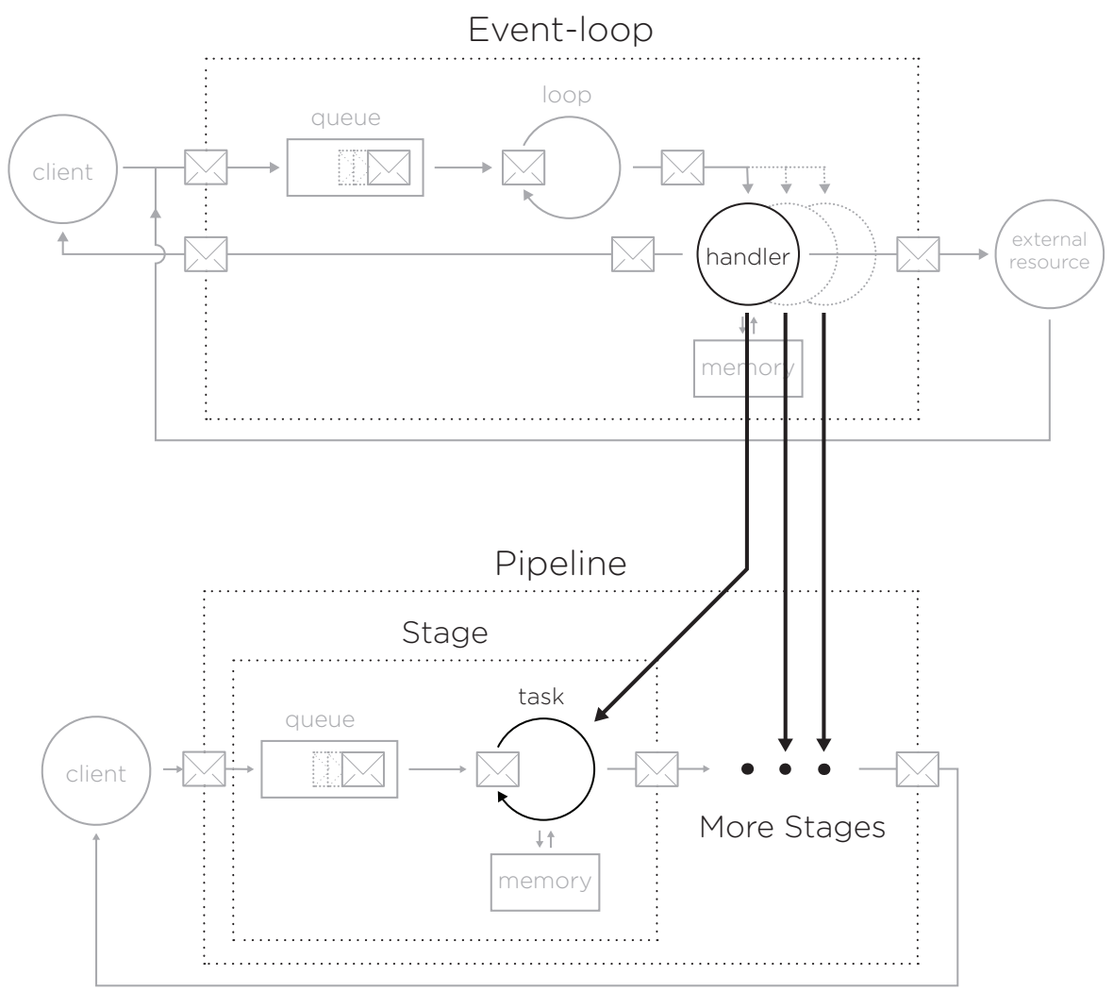
FLUXIONAL LANGUAGE

```
flx <id> & <tags> {<ctx>}
>> <destination> [<message>]
 <fn>
```

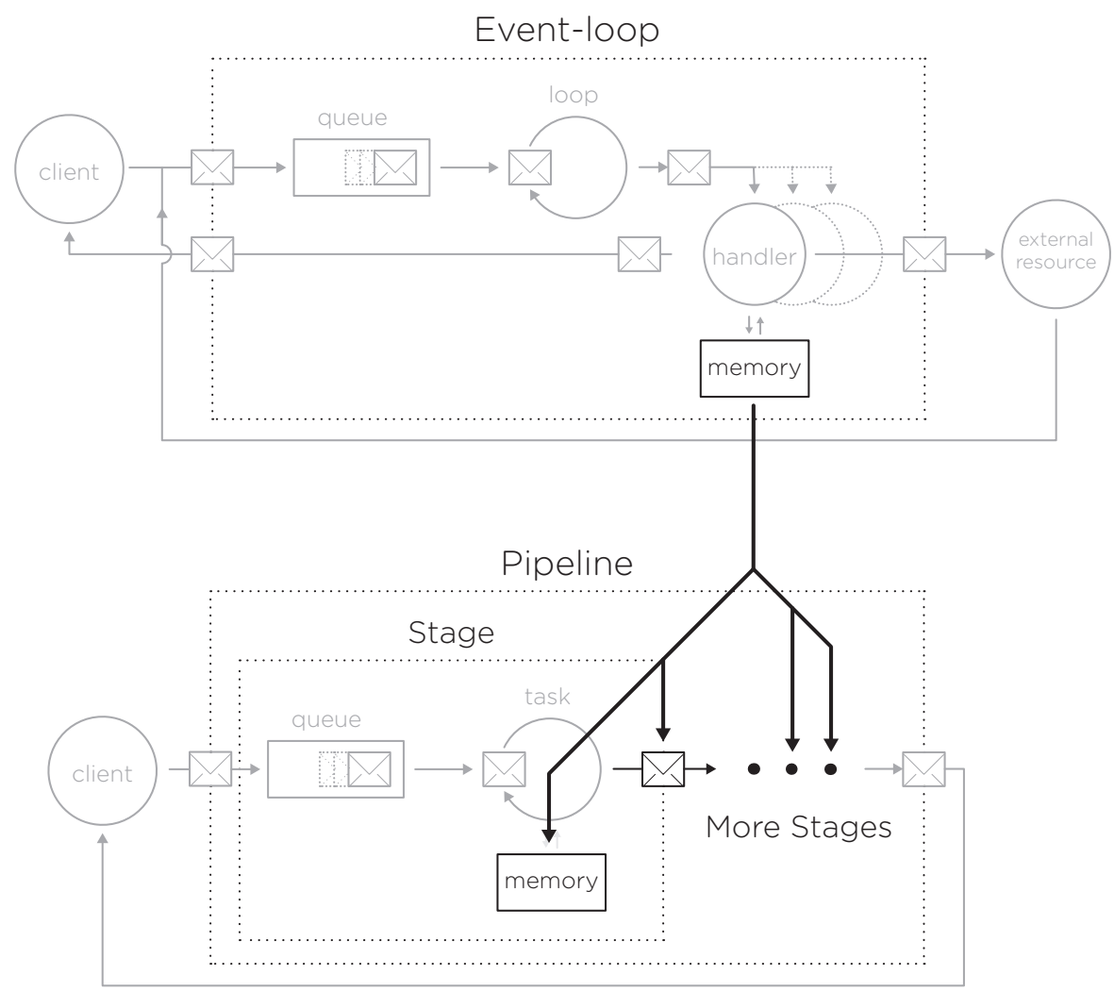


FLUXIONAL COMPILER

1 RUPTURE POINTS

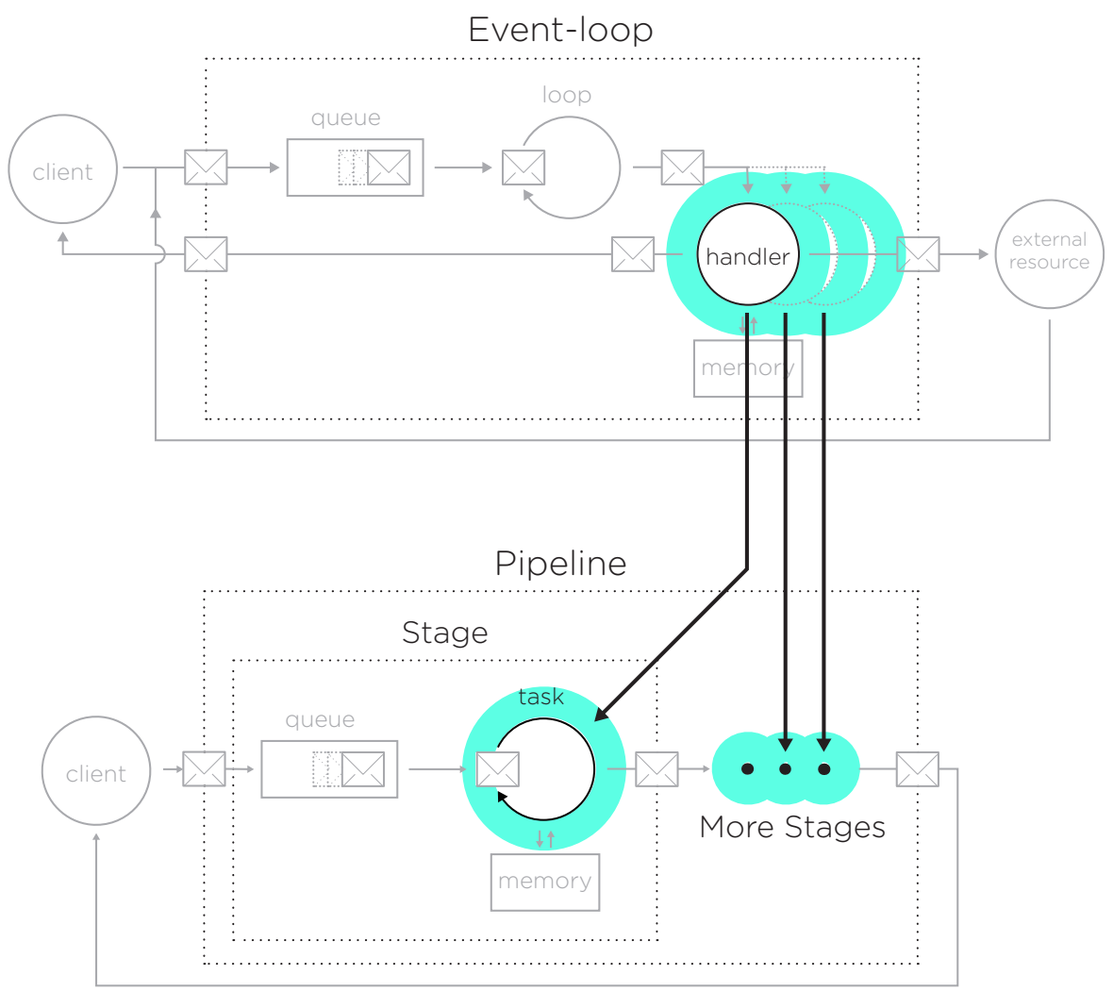


2 INDEPENDENCE

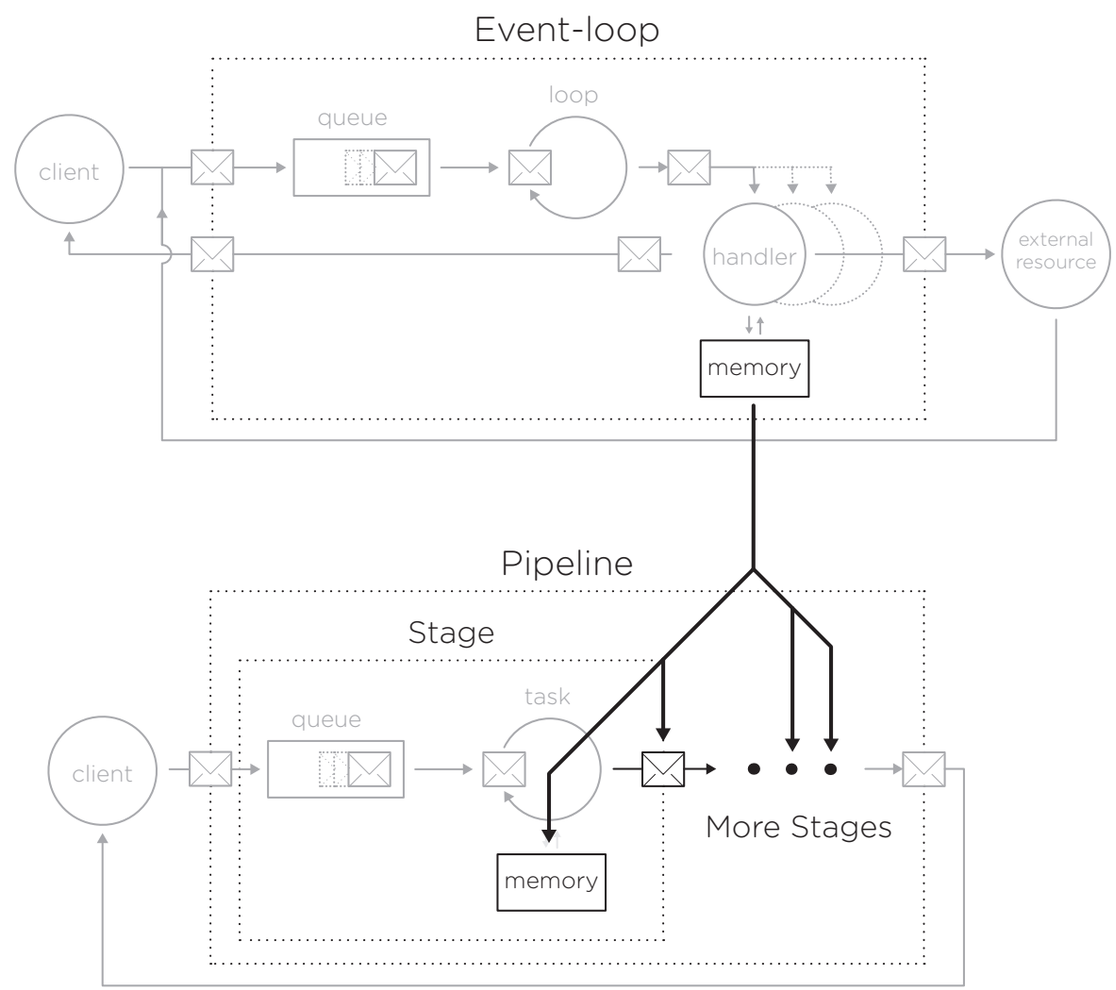


FLUXIONAL COMPILER

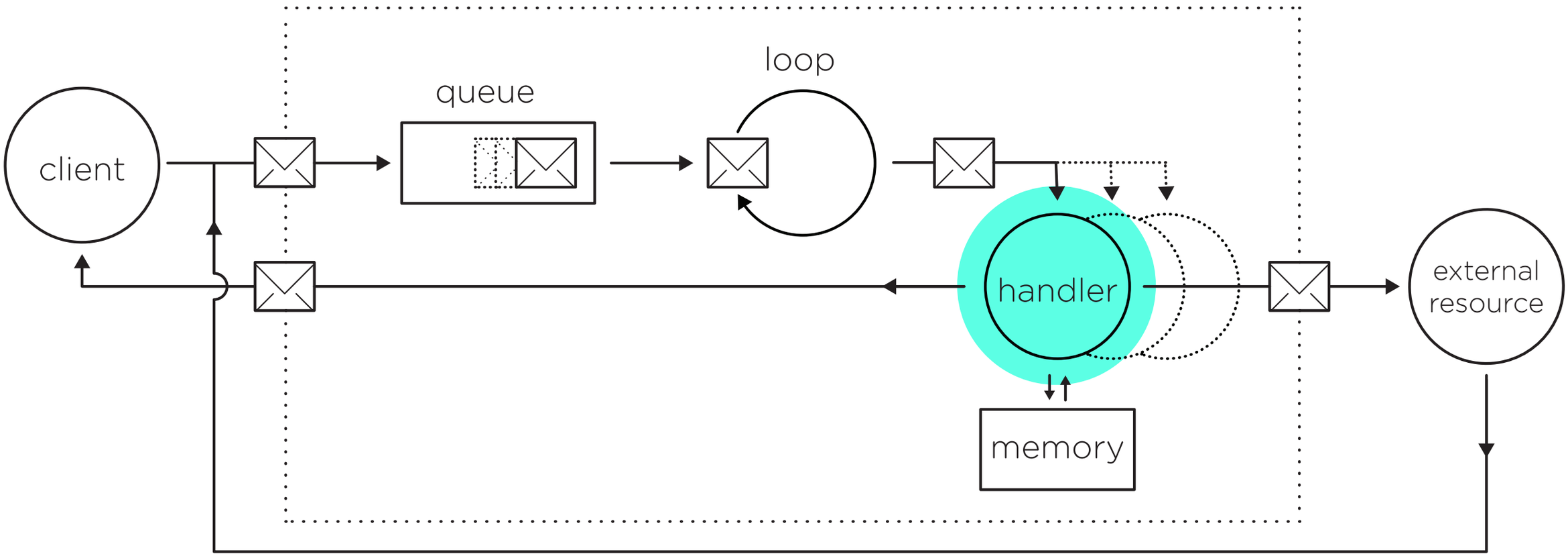
1 RUPTURE POINTS

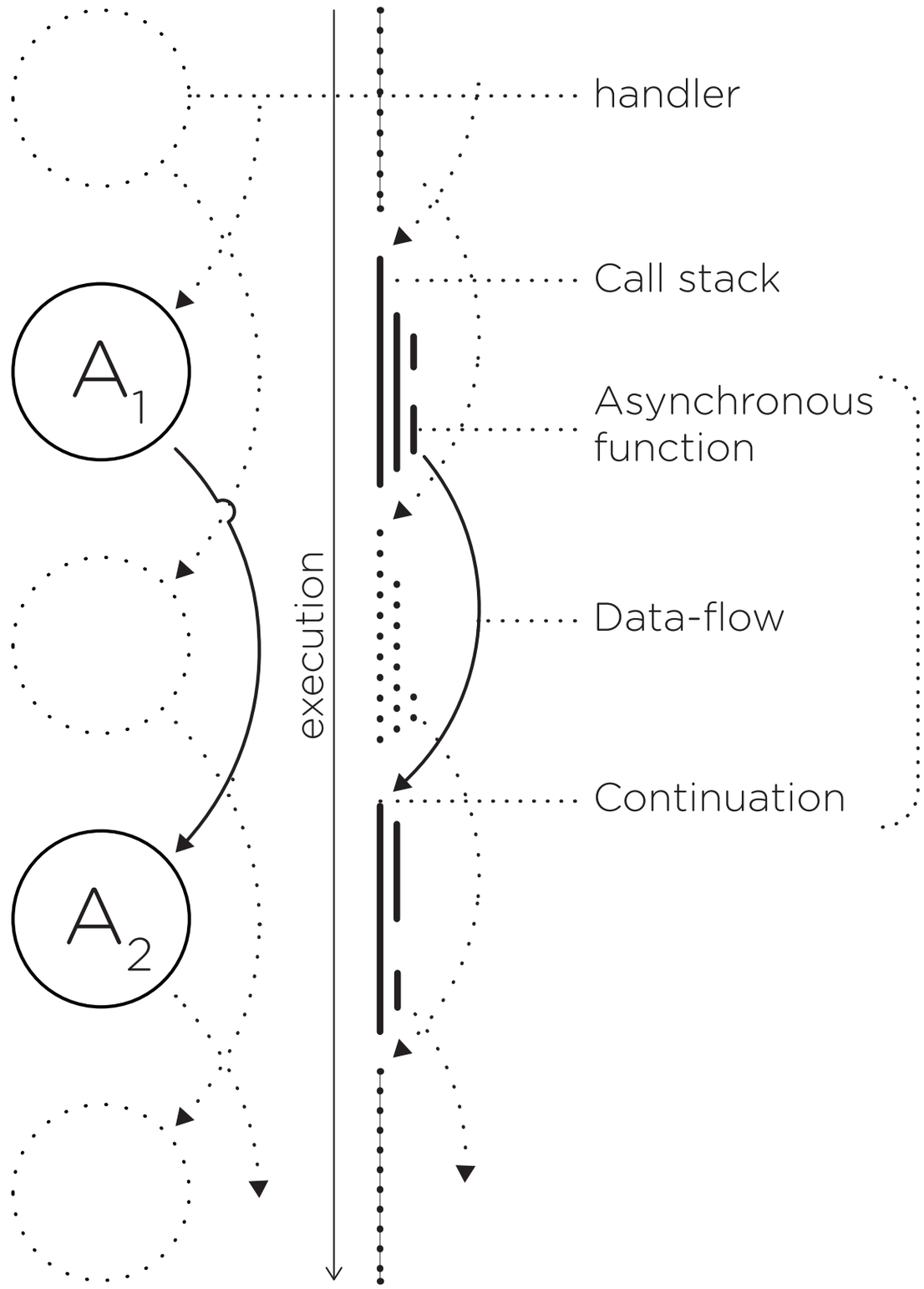


2 INDEPENDENCE

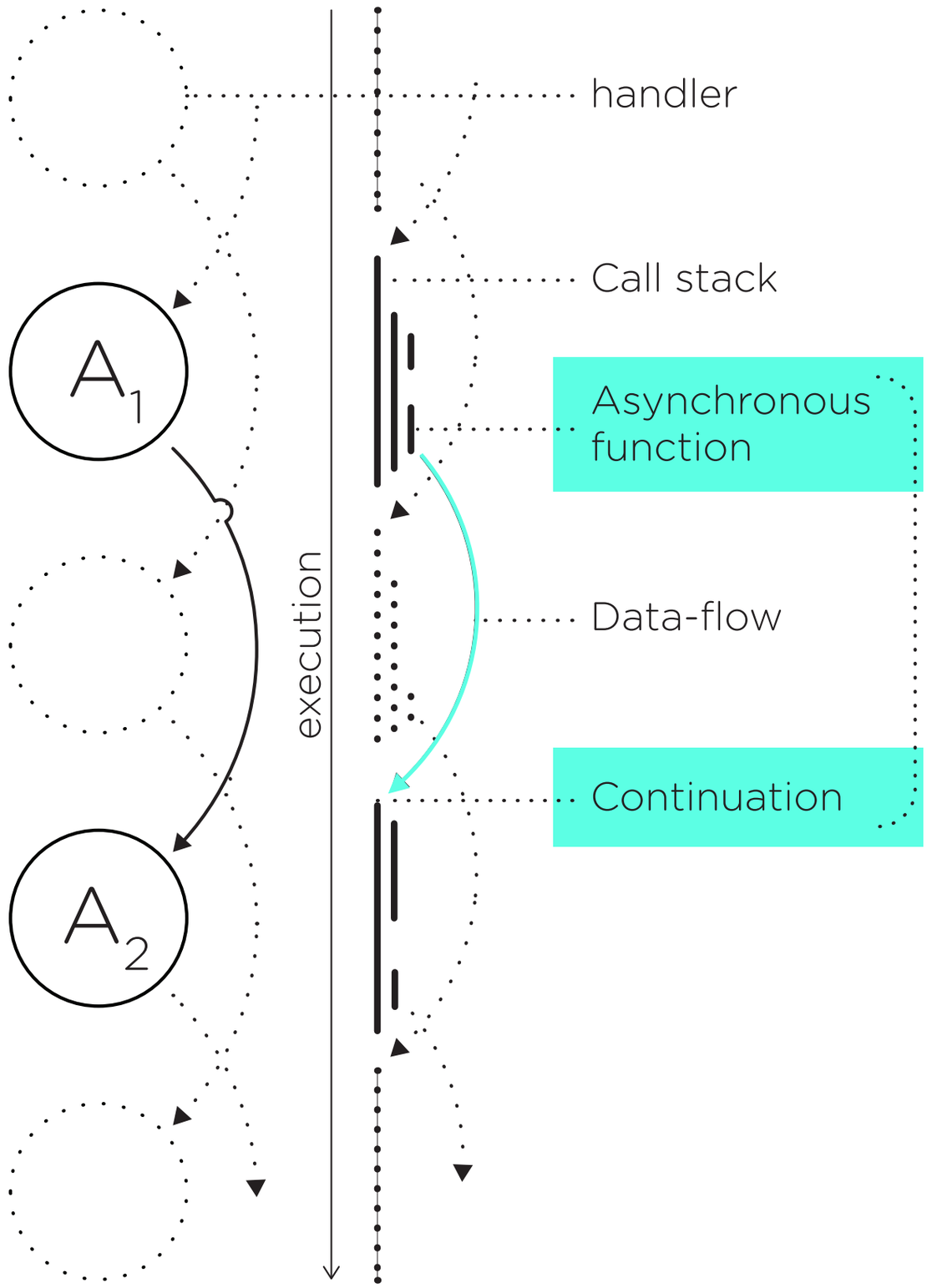


RUPTURE POINTS





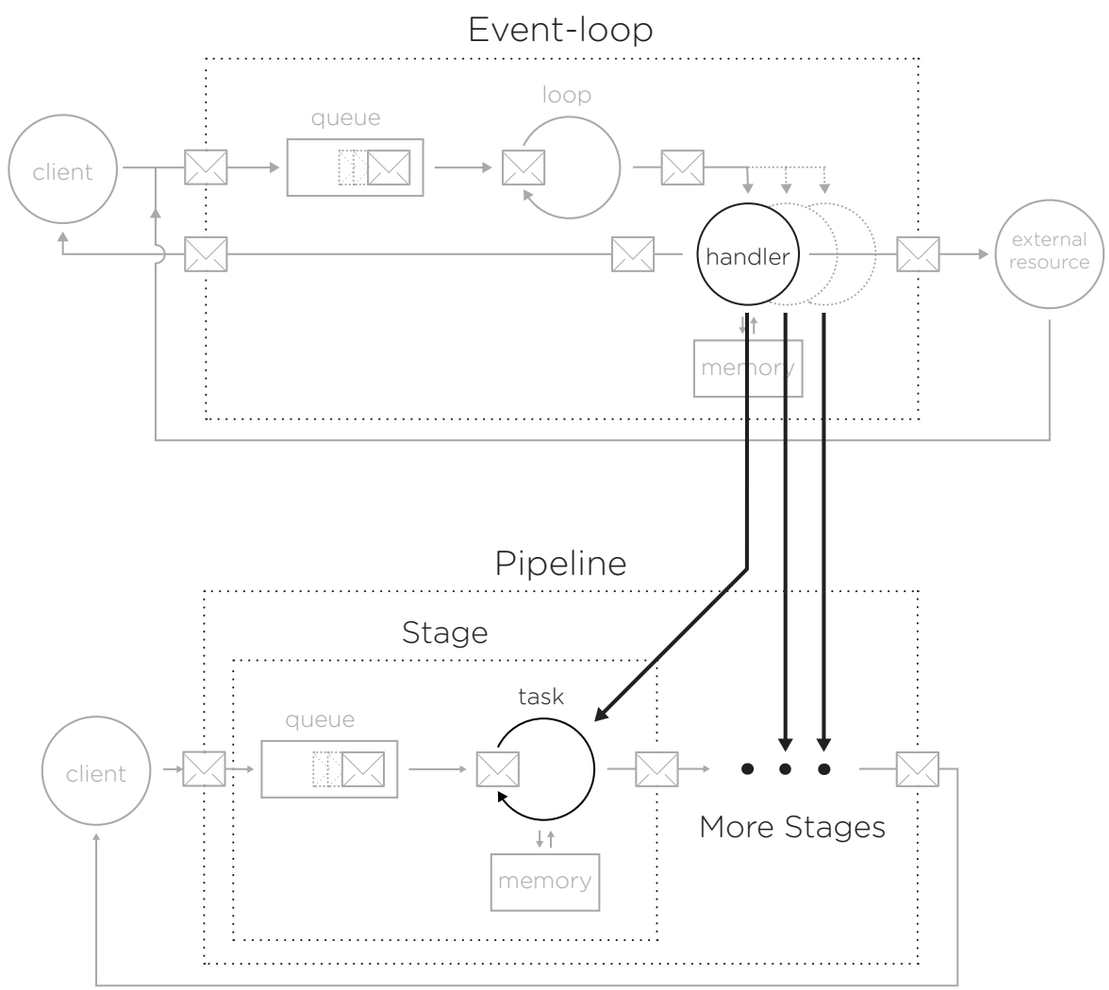
Rupture point



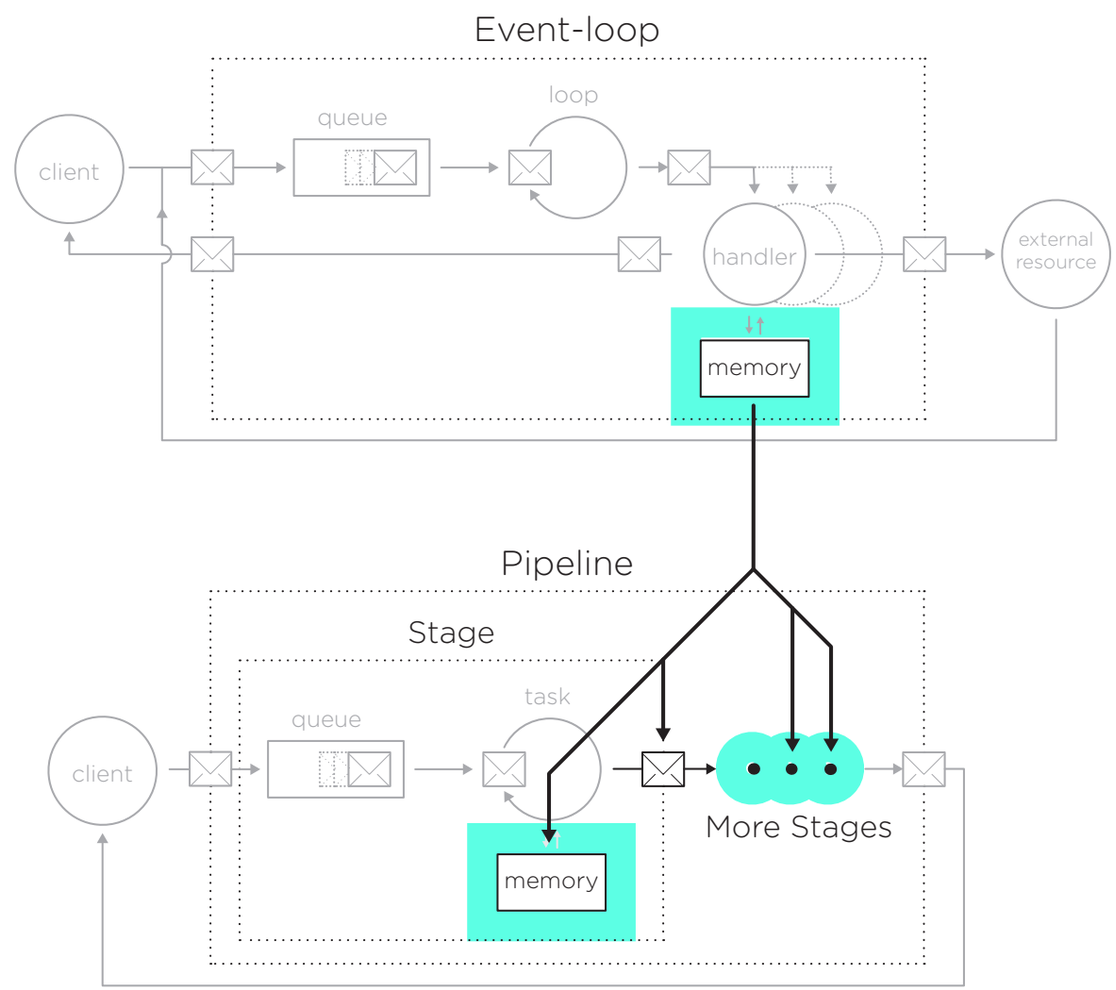
Rupture point

FLUXIONAL COMPILER

1 RUPTURE POINTS

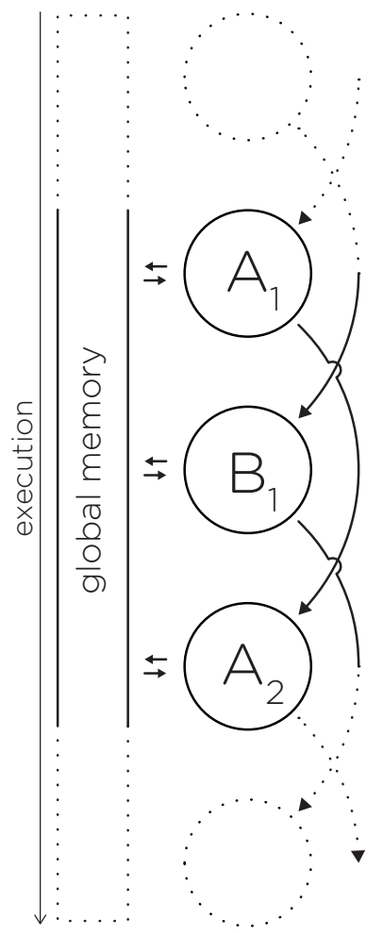


2 INDEPENDENCE

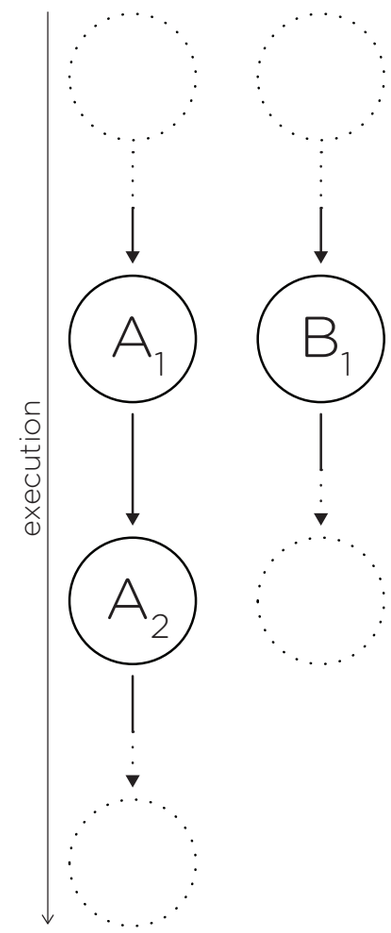


INDEPENDENCE

Shared memory
Sequential execution



Stateless
Parallel execution



INDEPENDENCE

We introduce 3 rules to qualify levels of independence

STATELESS ● Replication

SCOPE ❶ Task Parallelism

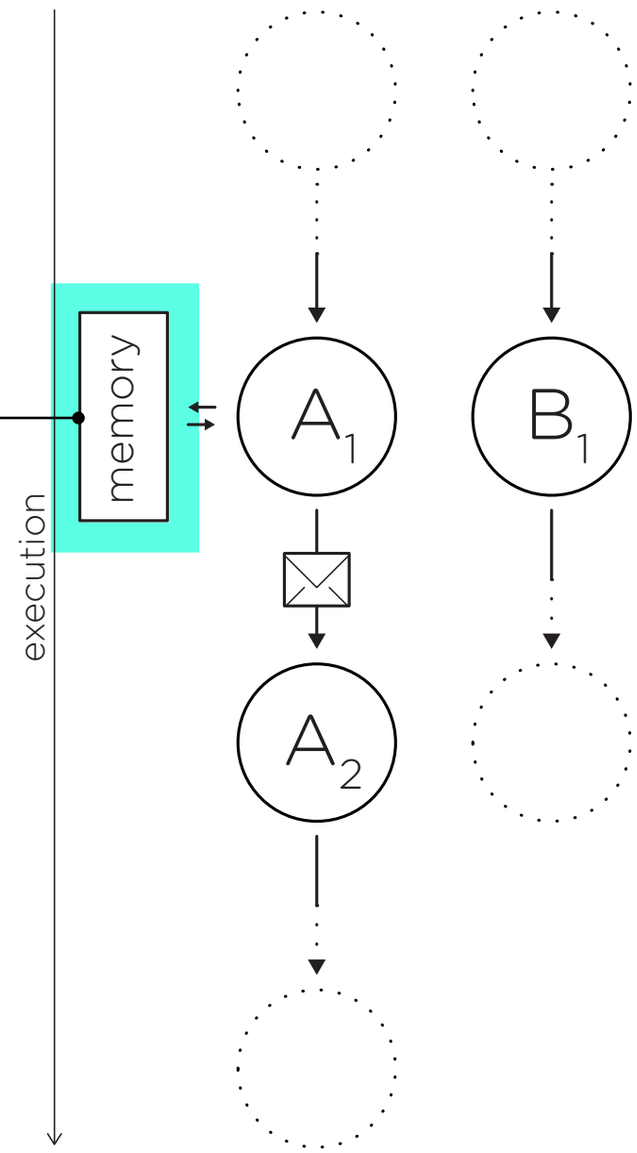
STREAM ❷ Pipeline Parallelism

SHARE ❸ Sequentiality

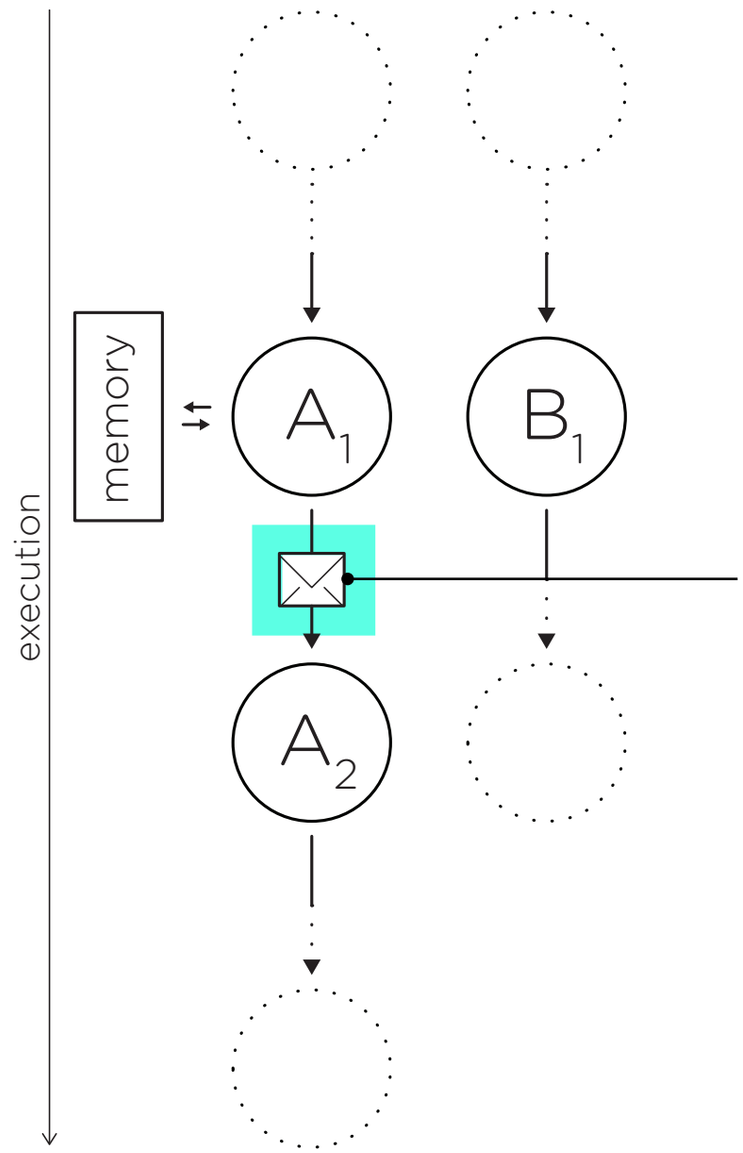
INDEPENDENCE

1 SCOPE

A variable that must be persisted from one message reception to the other.



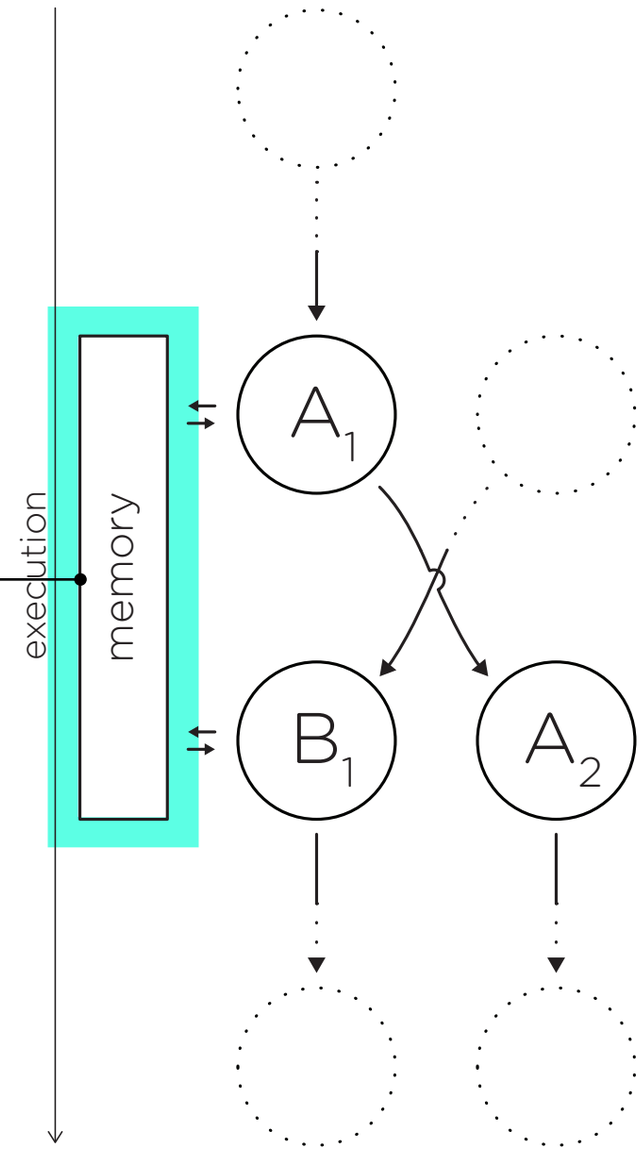
INDEPENDENCE



STREAM ❷

A variable required by a downstream handler.

INDEPENDENCE



3 SHARE

A variable required simultaneously by several handlers.

1 FLUXIONAL EXECUTION MODEL

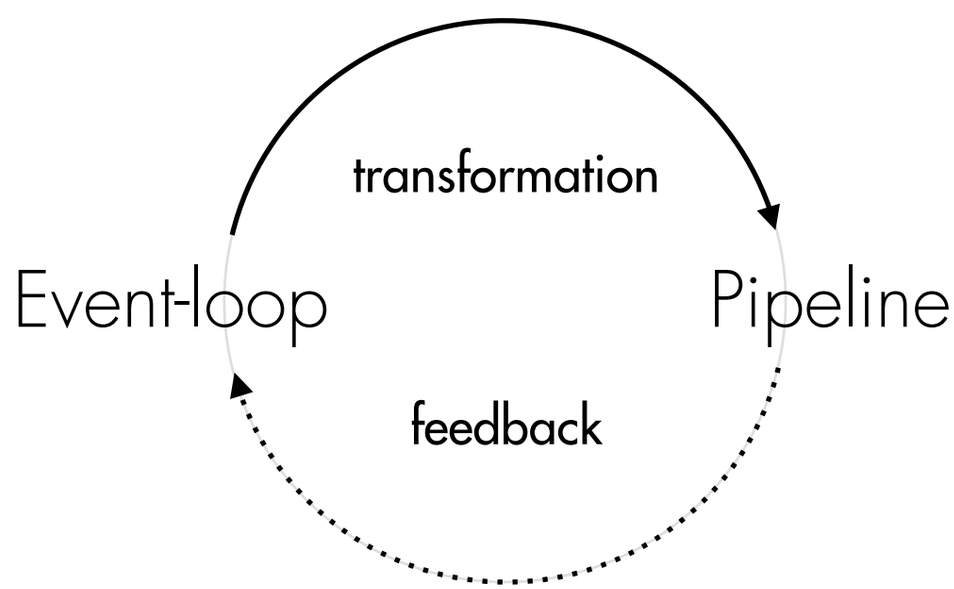
Compatible with both programming models

2 FLUXIONAL COMPILER

From one programming model to the other

1. RUPTURE POINTS
2. INDEPENDENCE

development productivity



execution efficiency

Implementations

IMPLEMENTATIONS

1. FLUXIONAL EXECUTION MODEL
2. COMPILERS
   1. DUE COMPILER
   2. FLUXIONAL COMPILER

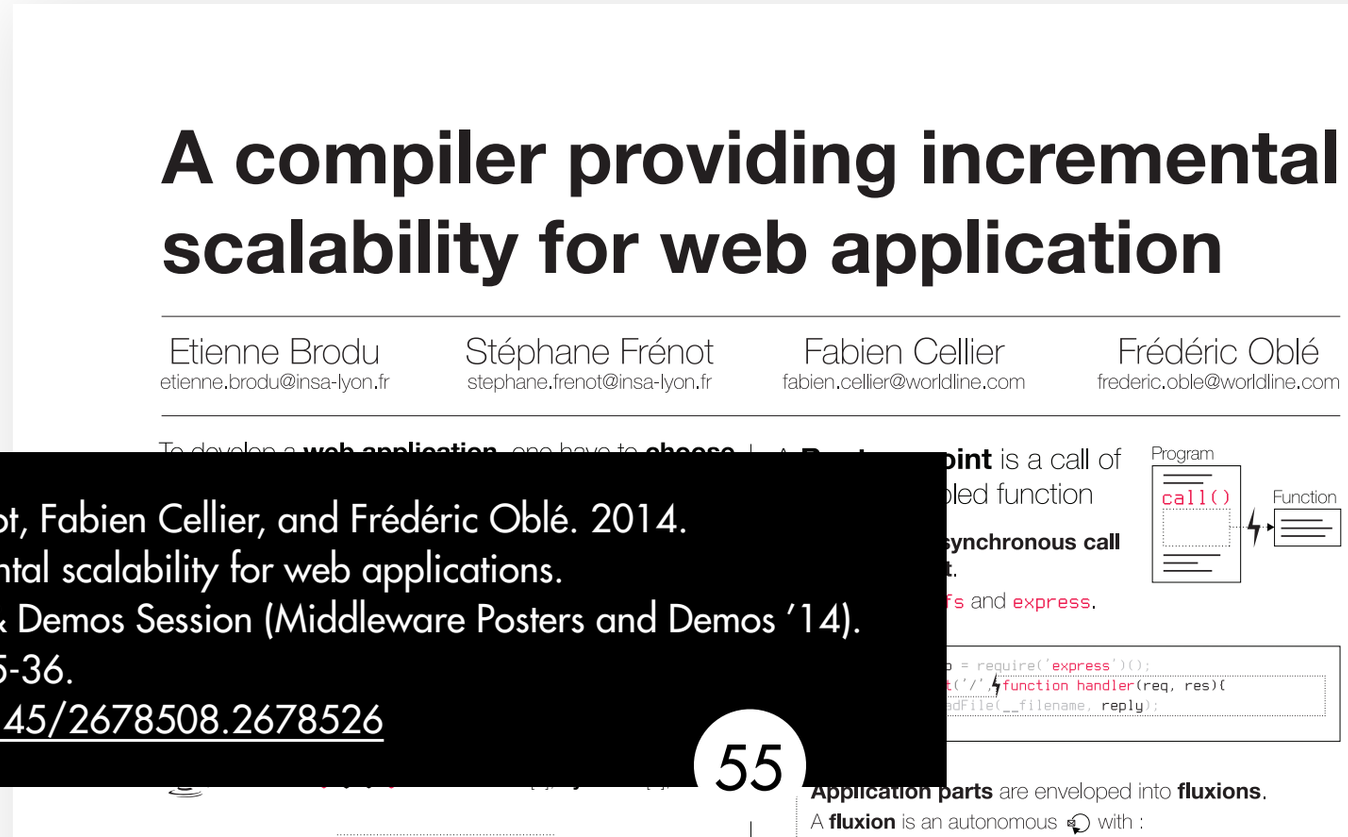
FLUXIONAL EXECUTION MODEL

220

lines of code

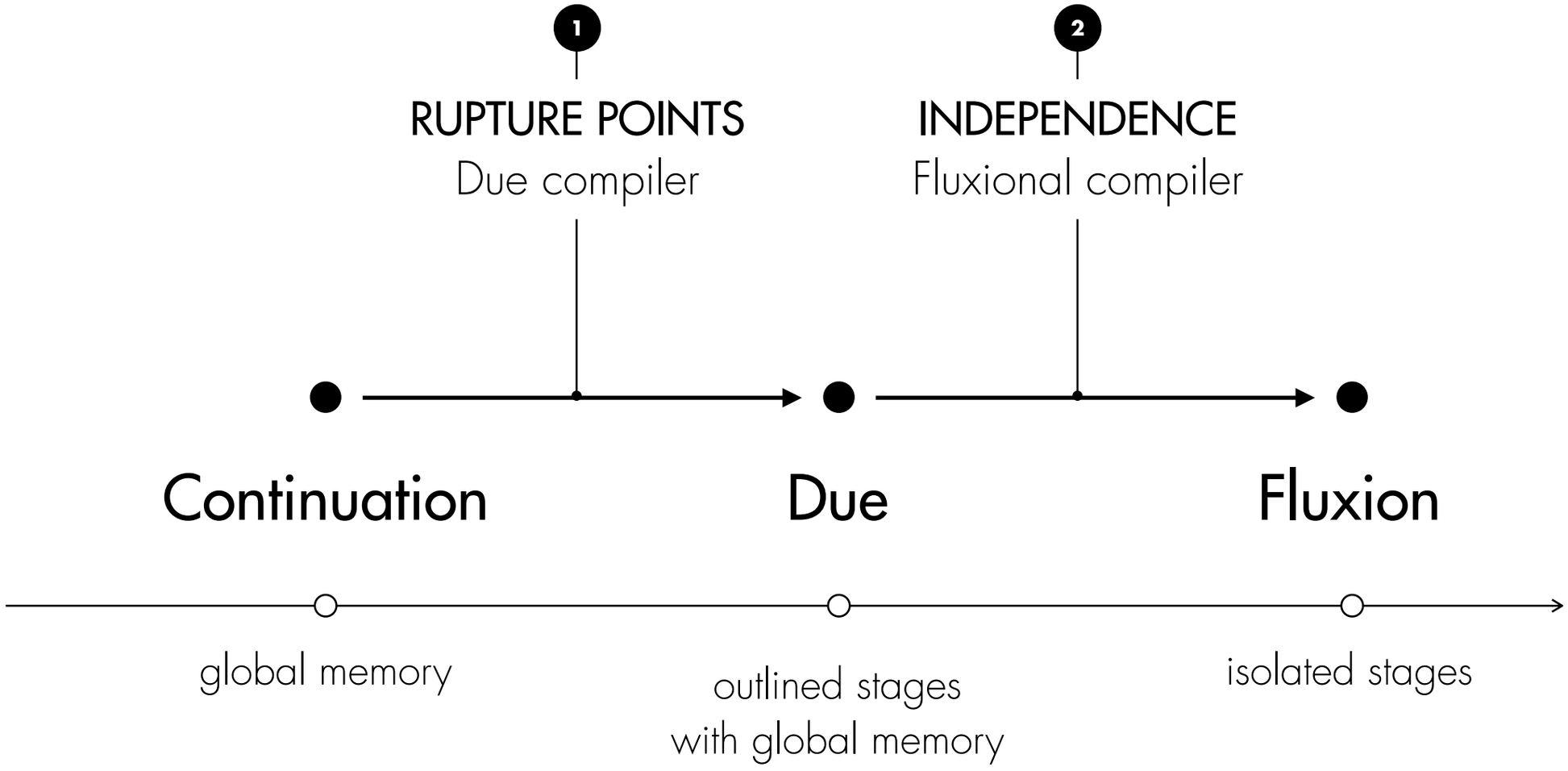
GitHub https://github.com/etnbrd/flx-lib

npm https://www.npmjs.com/package/flx

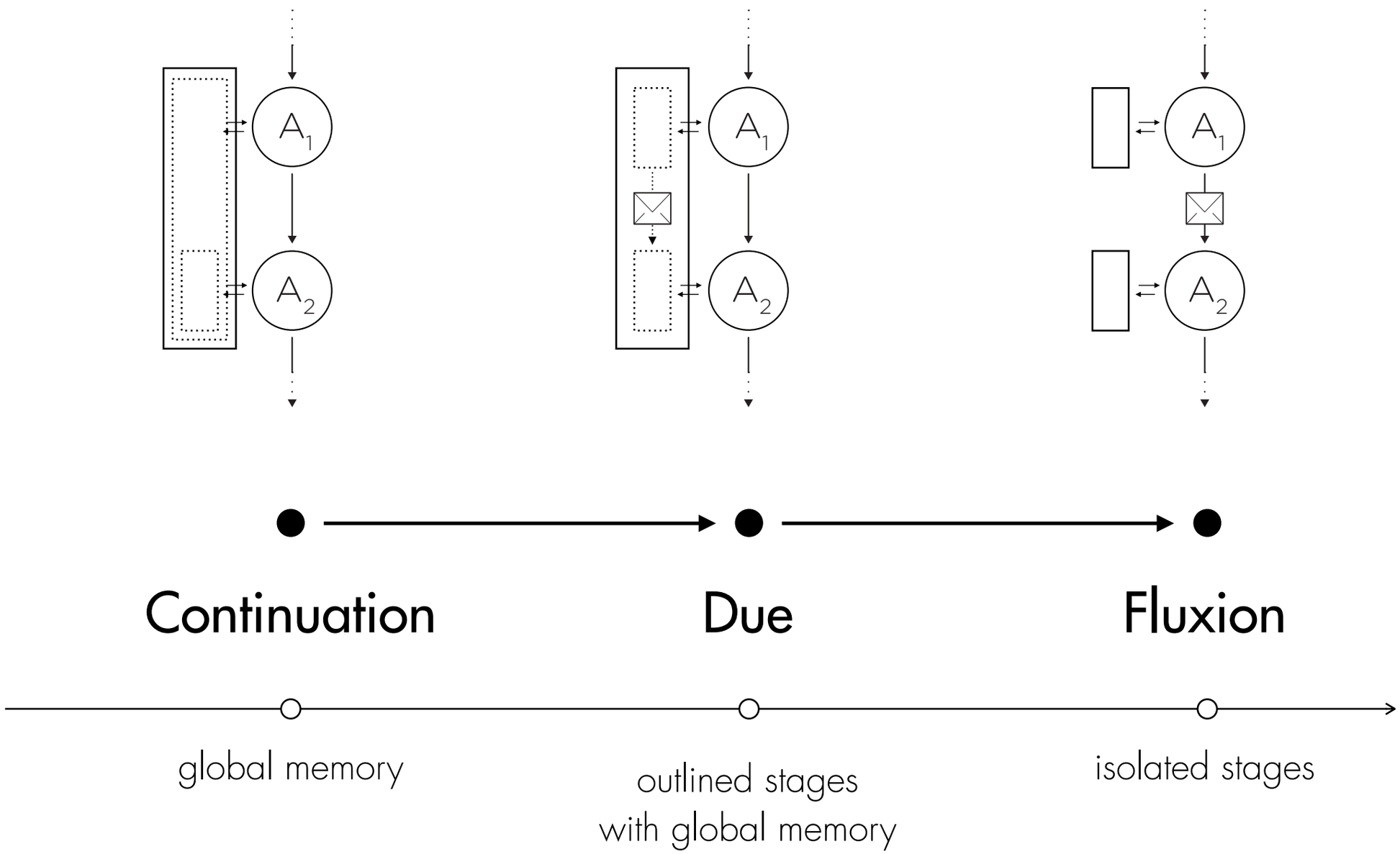


Etienne Brodu, Stéphane Frénot, Fabien Cellier, and Frédéric Oblé. 2014.
A compiler providing incremental scalability for web applications.
In Proceedings of the Posters & Demos Session (Middleware Posters and Demos '14).
ACM, New York, NY, USA, 35-36.
DOI=http://dx.doi.org/10.1145/2678508.2678526

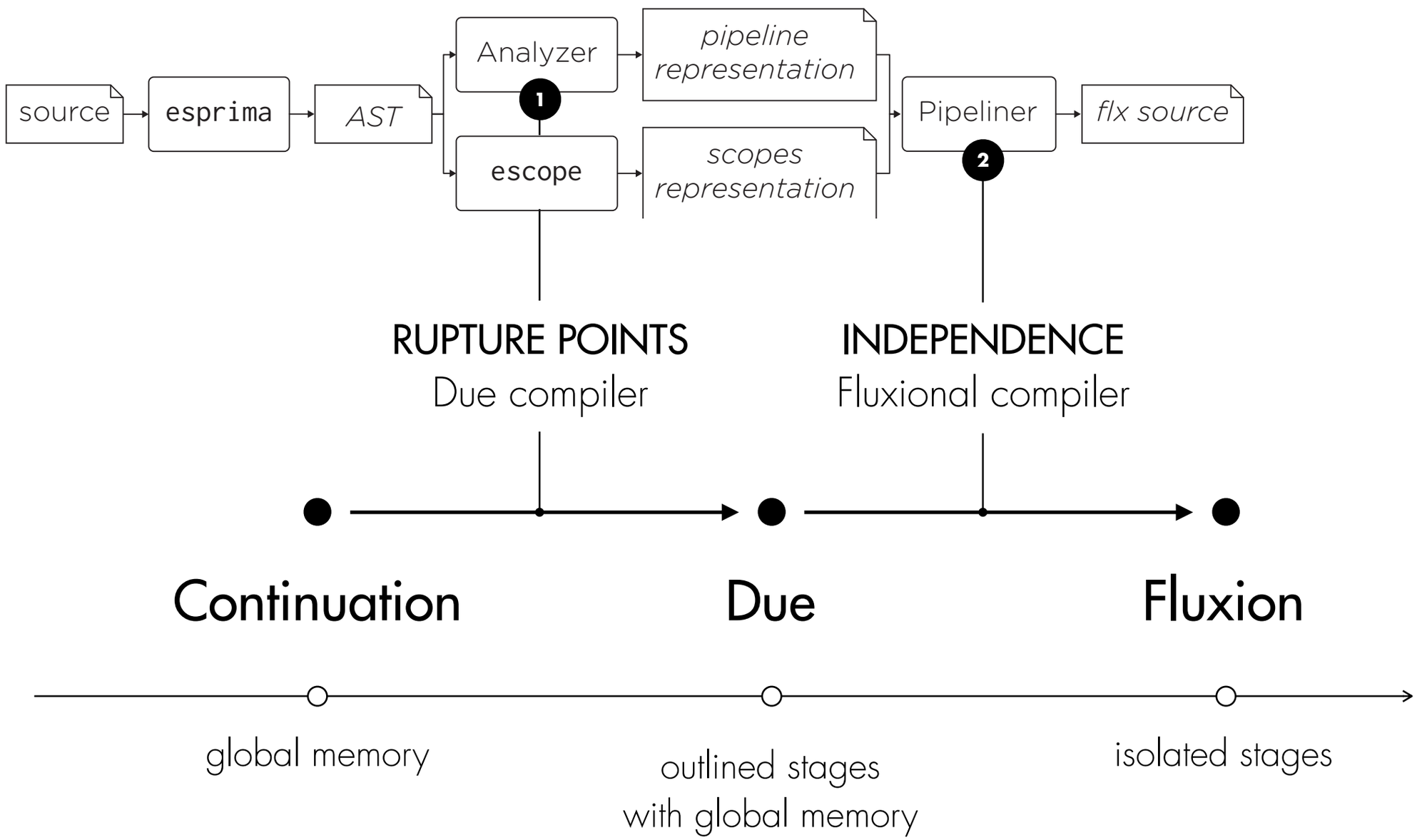
COMPILERS


1
RUPTURE POINTS
Due compiler
2
INDEPENDENCE
Fluxional compiler
Continuation
Due
Fluxion
global memory
outlined stages
with global memory
isolated stages

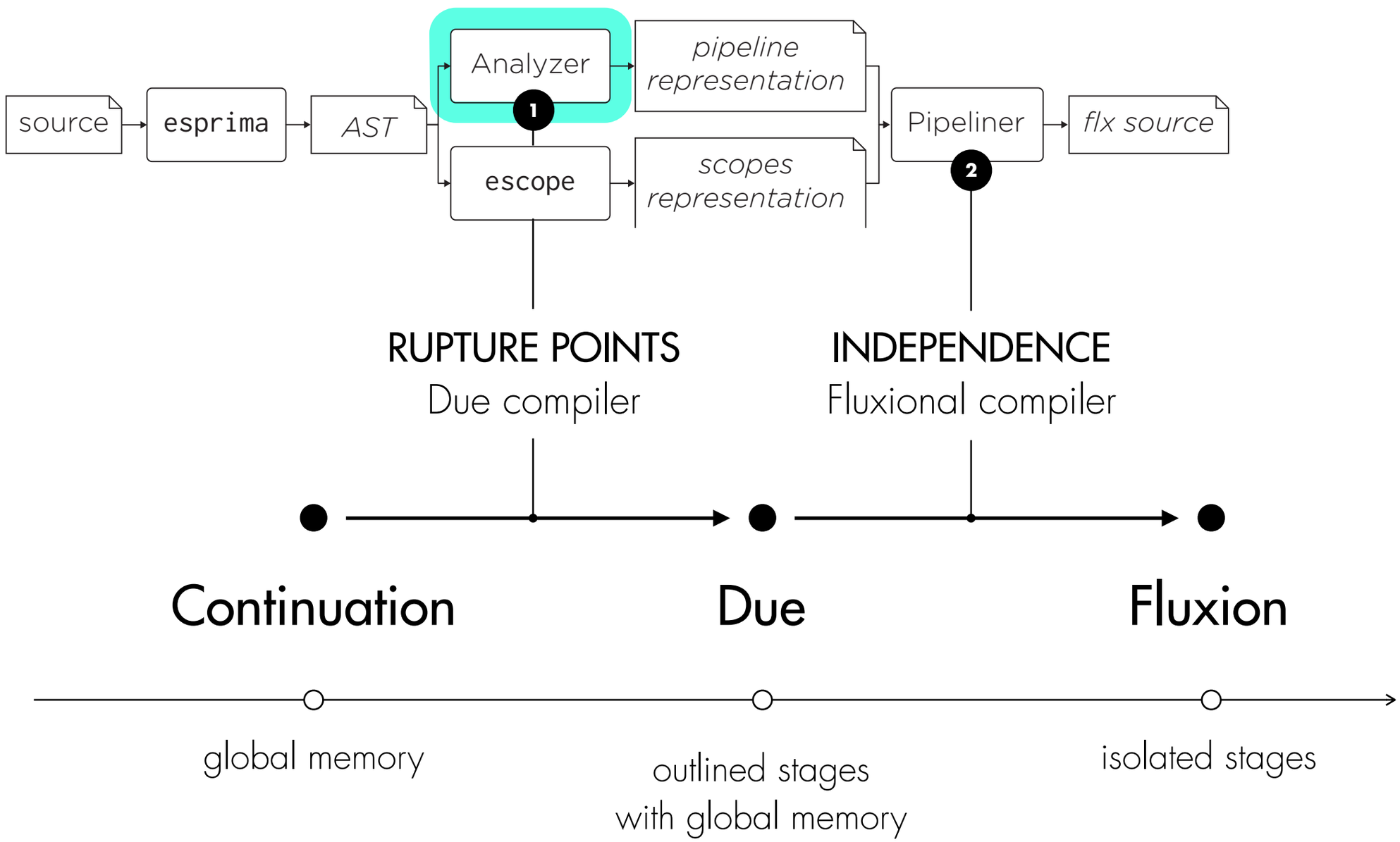
COMPILERS


A1
A2
A1
A2
A1
A2
Continuation
Due
Fluxion
global memory
outlined stages
with global memory
isolated stages

COMPILERS


source
esprima
AST
Analyzer
1
escope
pipeline representation
scopes representation
Pipeliner
2
flx source
RUPTURE POINTS
Due compiler
INDEPENDENCE
Fluxional compiler
Continuation
Due
Fluxion
global memory
outlined stages with global memory
isolated stages

COMPILERS


source
esprima
AST
Analyzer
1
escope
pipeline representation
scopes representation
Pipeliner
2
flx source
RUPTURE POINTS
Due compiler
INDEPENDENCE
Fluxional compiler
Continuation
Due
Fluxion
global memory
outlined stages with global memory
isolated stages

ANALYZER

```
asyncCall(arguments, function callback(result){ ② });
// Following statements ①
```

ANALYZER

```
asyncCall(arguments, function callback(result){ ② });
// Following statements ①
```

Rupture points detection based on a list of known asynchronous functions.

ANALYZER

```
asyncCall(arguments, function callback(result){ ② });
// Following statements ①
```

It identifies the callback declared in situ to encapsulate it in the downstream fluxion.

ANALYZER

Two types of callbacks

LISTENER

CONTINUATION

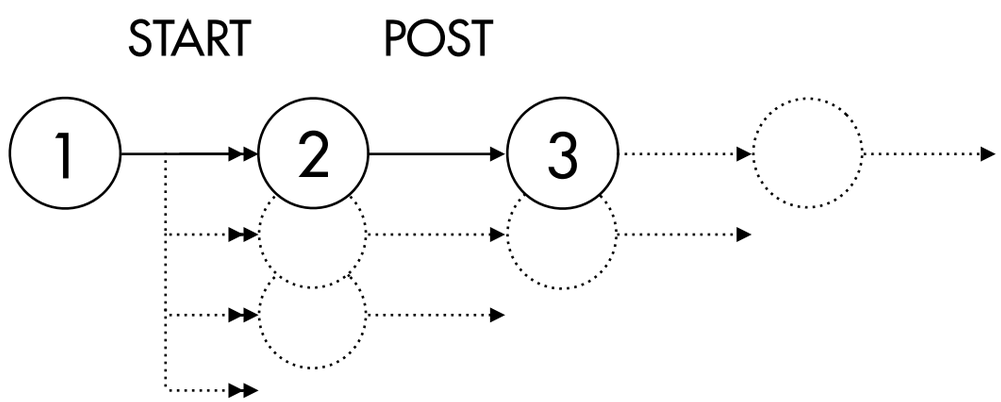
Two types of rupture points

START >>

It indicates the input of a data stream.
It is the beginning of a chain.

POST ->

It indicates a continuity in the chain.



DUE COMPILER

600+ / 35K+

lines of code

GitHub https://github.com/etnbrd/due-compiler

web app http://compiler-due.apps.zone52.org

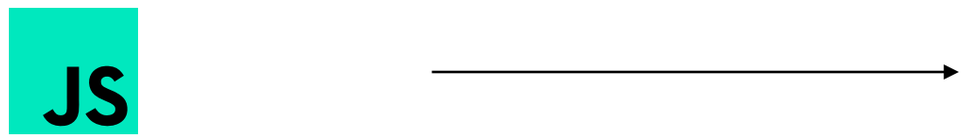
Toward automatic update from callbacks to Promises

Etienne Brodu, Stéphane Frénot
firstname.lastname@insa-lyon.fr
Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA, F-69621, Villeurbanne, France

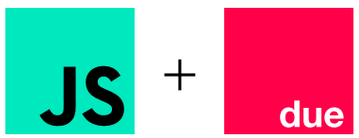
Frédéric Oblé
frederic.oble@worldline.com
Worldline
53 avenue Paul Krüger - CS 60195
69624 Villeurbanne Cedex

Etienne Brodu, Stéphane Frénot, and Frédéric Oblé. 2015.
Toward automatic update from callbacks to promises.
In Proceedings of the 1st Workshop on All-Web Real-Time Systems (AWeS '15).
ACM, New York, NY, USA, , Article 1 , 8 pages.
DOI=http://dx.doi.org/10.1145/2749215.2749216

DUE COMPILER



Javascript with continuations

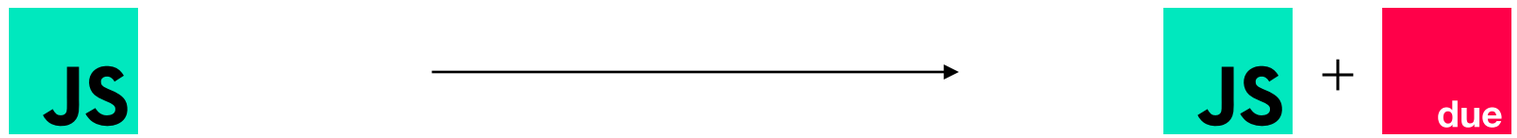


Javascript with dues

```
app.get('/', function handler(req, res){
  fs.readFile(__filename,
    function sendFile(err, file){
      var fileDesc = file.name + file.size;
      …
      res.send(file, function end(err){
        console.log('sent ' + fileDesc);
        …
    }
  );
});
```

```
app.get('/', function handler(req, res){
  var fileDesc;
  fs.readFile(__filename)
  .then(function sendFile(err, file){
    fileDesc = file.name + file.size;
    …
    return res.send(file);
  })
  .then(function end(err, file){
    console.log('sent ' + fileDesc);
    …
  });
});
```

DUE COMPILER



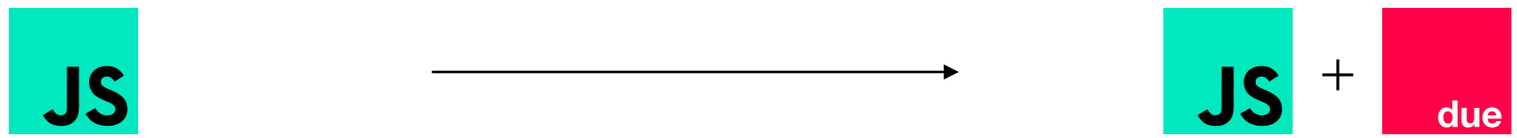
Javascript with continuations

Javascript with dues

```
app.get('/', function handler(req, res){
  fs.readFile(__filename,
   function sendFile(err, file){
     var fileDesc = file.name + file.size;
     …
     res.send(file, function end(err){
       console.log('sent ' + fileDesc);
       …
    }
  );
});
```

```
app.get('/', function handler(req, res){
  var fileDesc;
  fs.readFile(__filename)
  .then(function sendFile(err, file){
    fileDesc = file.name + file.size;
    …
    return res.send(file);
  })
  .then(function end(err, file){
    console.log('sent ' + fileDesc);
    …
  });
});
```

DUE COMPILER



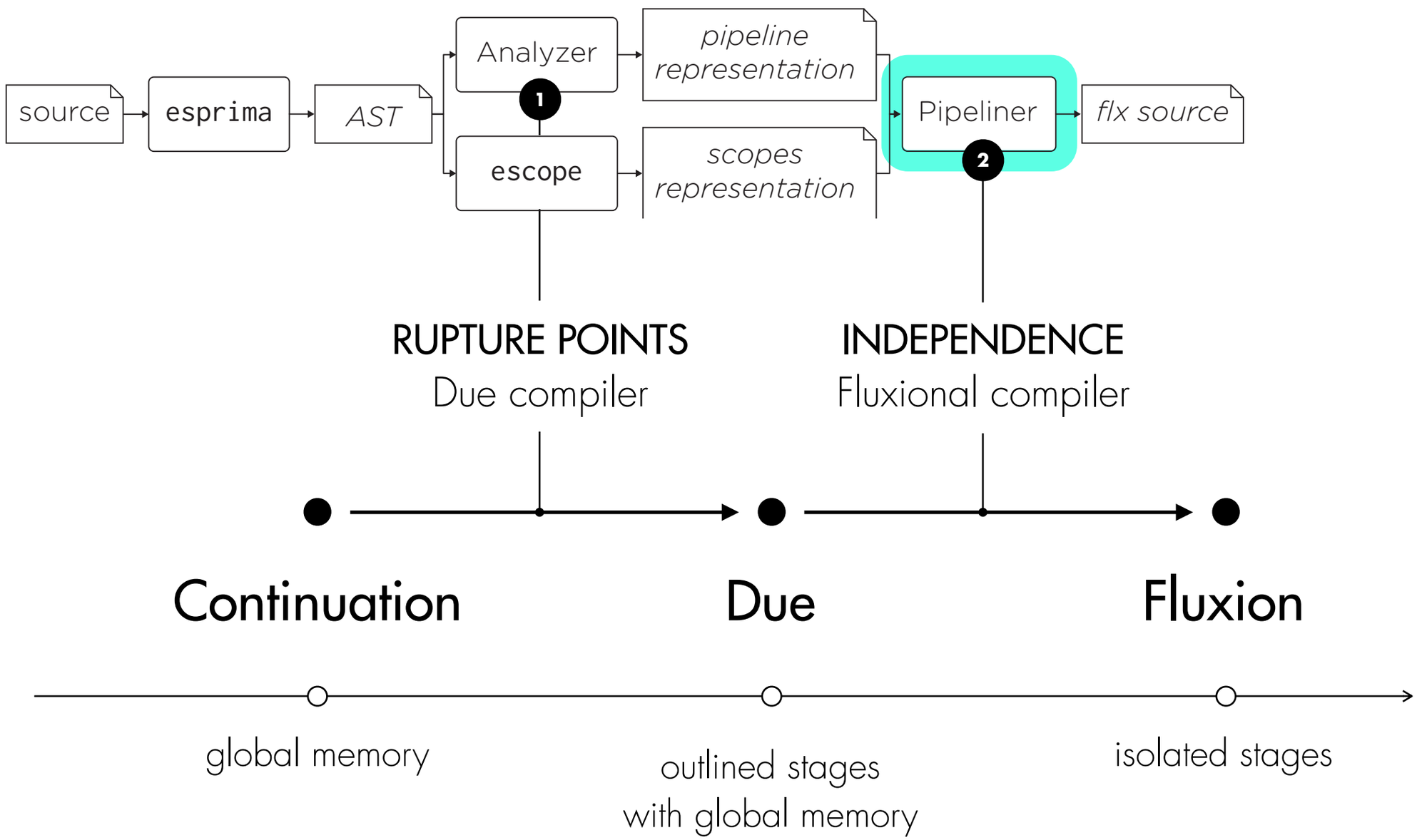
Javascript with continuations

Javascript with dues

```
app.get('/', function handler(req, res){
  fs.readFile(__filename,
   function sendFile(err, file){
     var fileDesc = file.name + file.size;
     …
     res.send(file, function end(err){
       console.log('sent ' + fileDesc);
       …
    }
  );
});
```

```
app.get('/', function handler(req, res){
  var fileDesc;
  fs.readFile(__filename)
  .then(function sendFile(err, file){
    fileDesc = file.name + file.size;
    …
    return res.send(file);
  })
  .then(function end(err, file){
    console.log('sent ' + fileDesc);
    …
  });
});
```

COMPILERS


source
esprima
AST
Analyzer
1
escope
pipeline representation
scopes representation
Pipeliner
2
flx source
RUPTURE POINTS
Due compiler
INDEPENDENCE
Fluxional compiler
Continuation
Due
Fluxion
global memory
outlined stages with global memory
isolated stages

PIPELINER

We introduce 3 rules to qualify levels of independence

STATELESS ● Replication

SCOPE ❶ Task Parallelism

STREAM ❷ Pipeline Parallelism

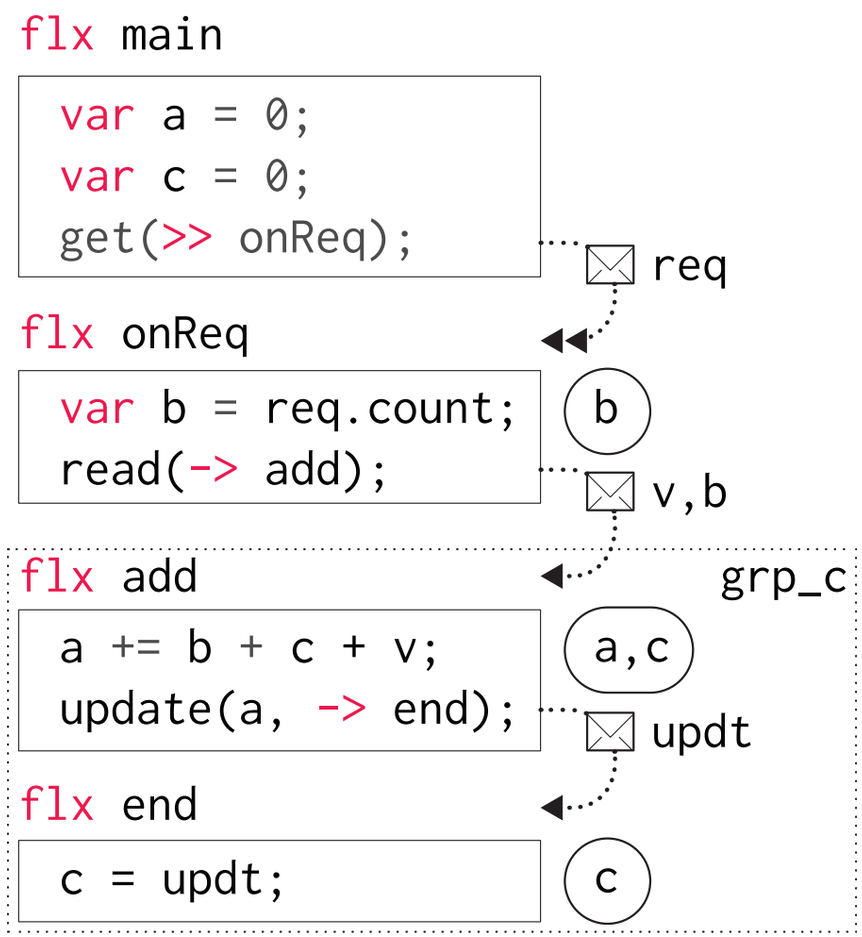
SHARE ❸ Sequentiality

PIPELINER

SCOPE ❶

Variable **a** is modified inside only one handler in the current chain. It needs to be stored in the context to be accessible from one message to another

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

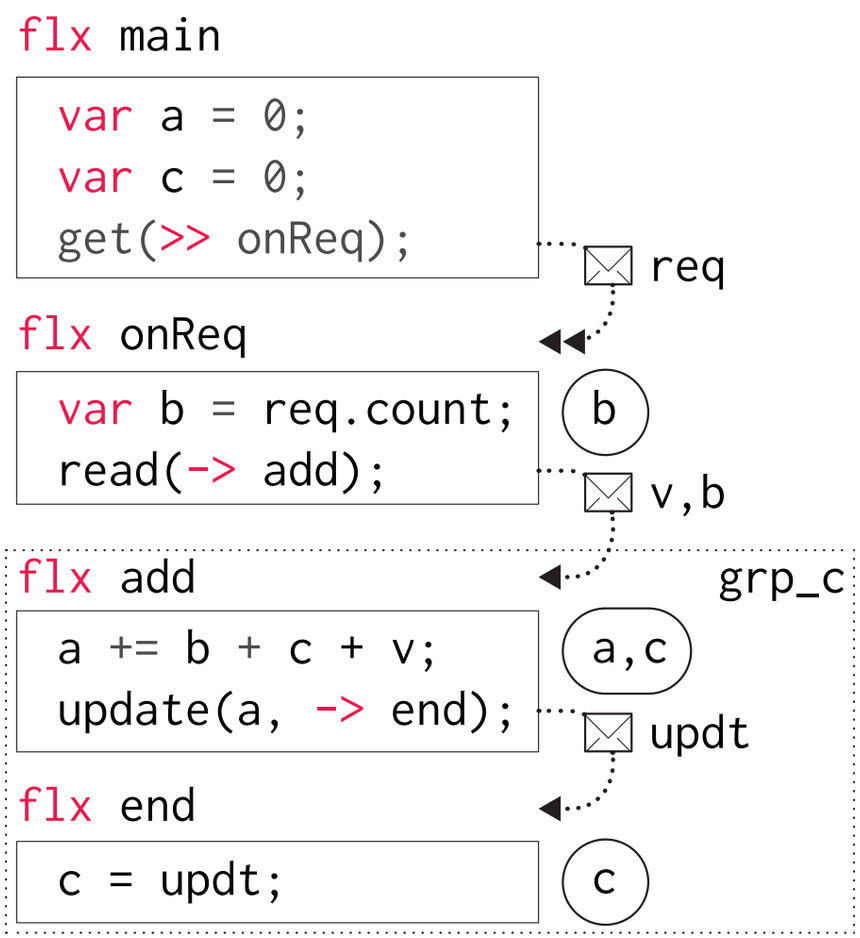


PIPELINER

SCOPE ❶

Variable **a** is modified inside only one handler in the current chain. It needs to be stored in the context to be accessible from one message to another

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

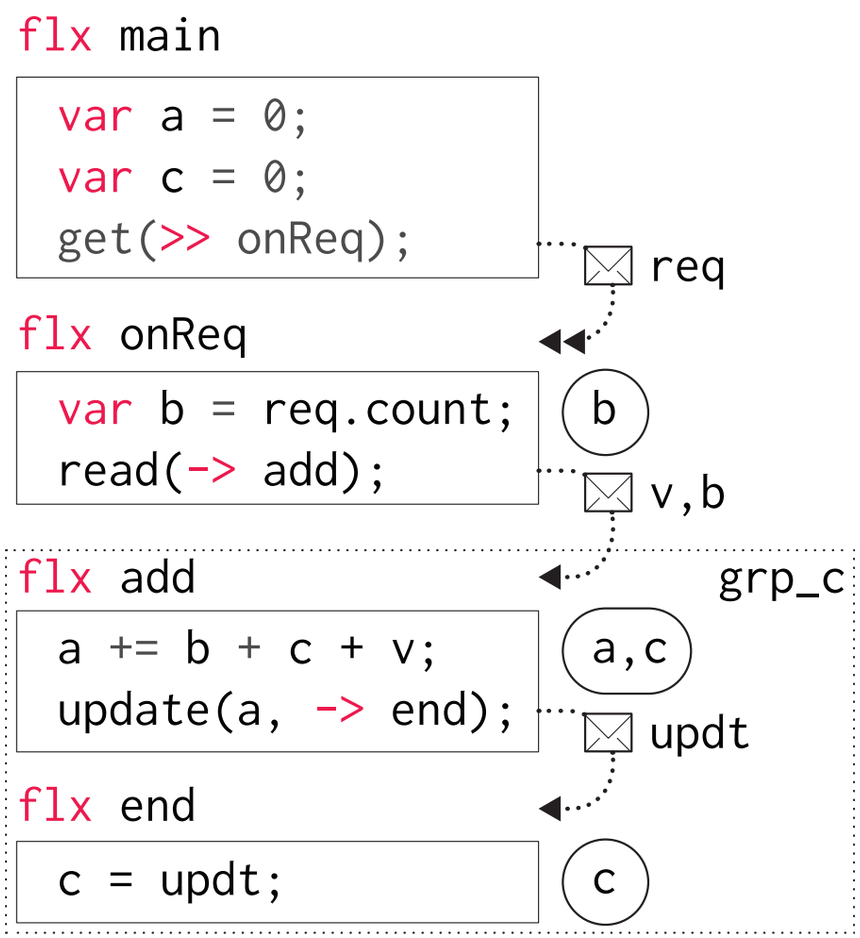


PIPELINER

SCOPE ❶

Variable **a** is modified inside only one handler in the current chain. It needs to be stored in the context to be accessible from one message to another

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

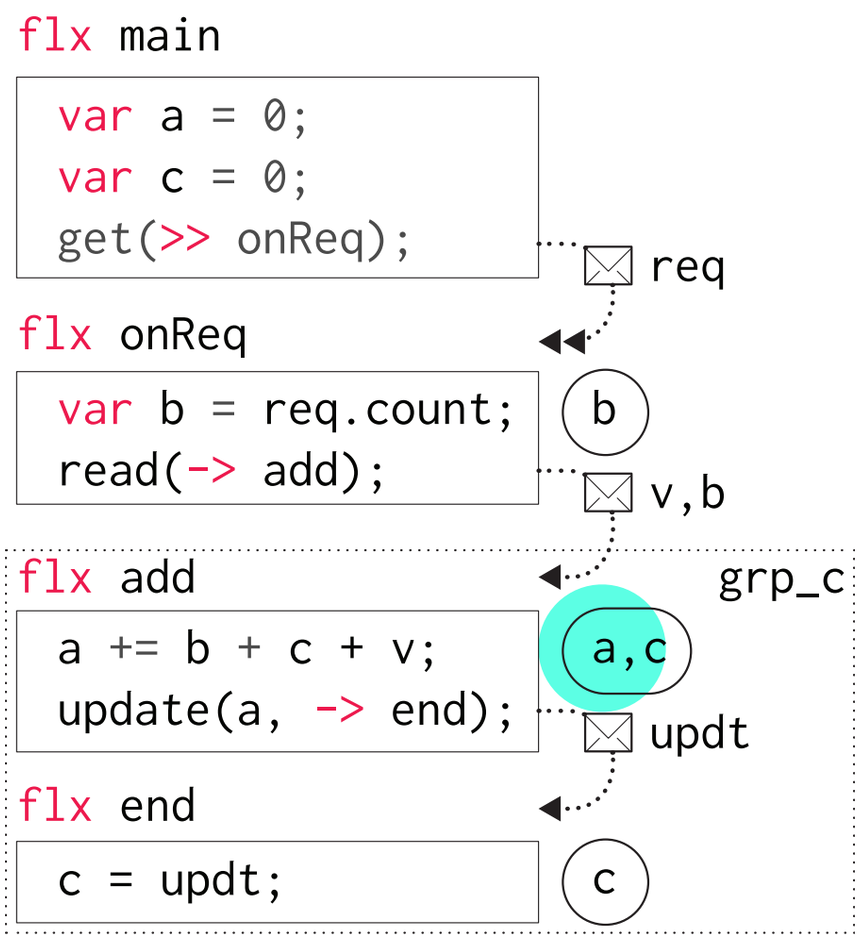


PIPELINER

SCOPE ❶

Variable **a** is modified inside only one handler in the current chain. It needs to be stored in the context to be accessible from one message to another

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

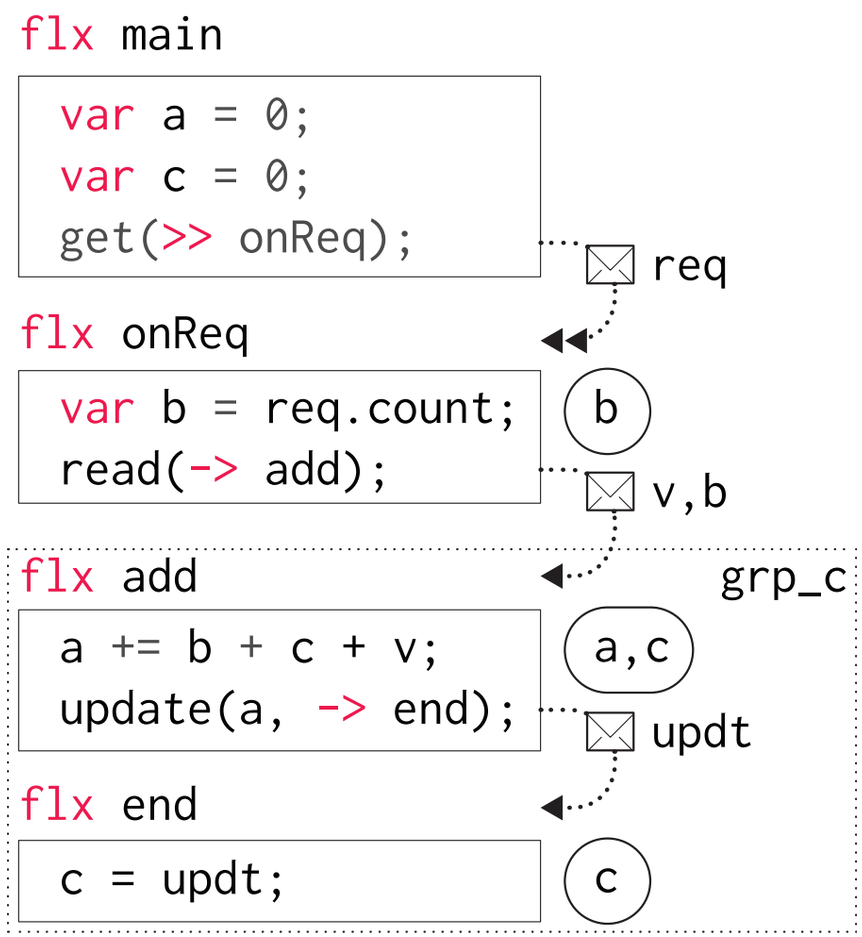


PIPELINER

STREAM ❷

Variable **b** is modified inside an handler, and read inside downstream handlers. This variable is propagated downstream.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

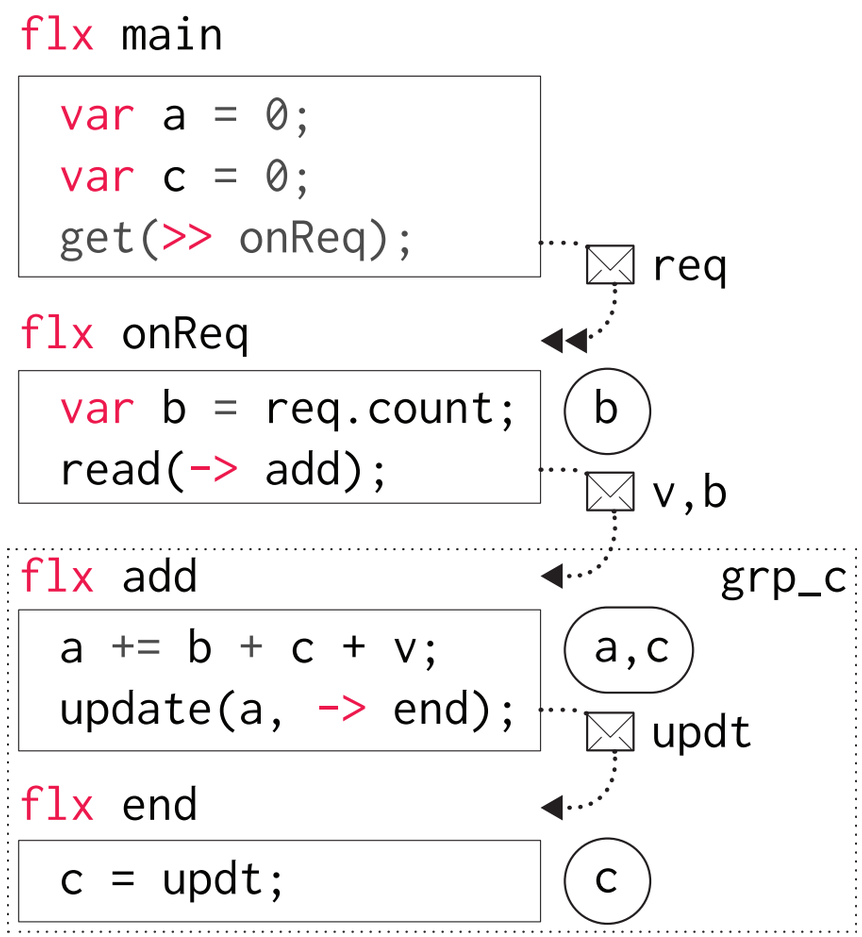


PIPELINER

STREAM ❷

Variable **b** is modified inside an handler, and read inside downstream handlers. This variable is propagated downstream.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```



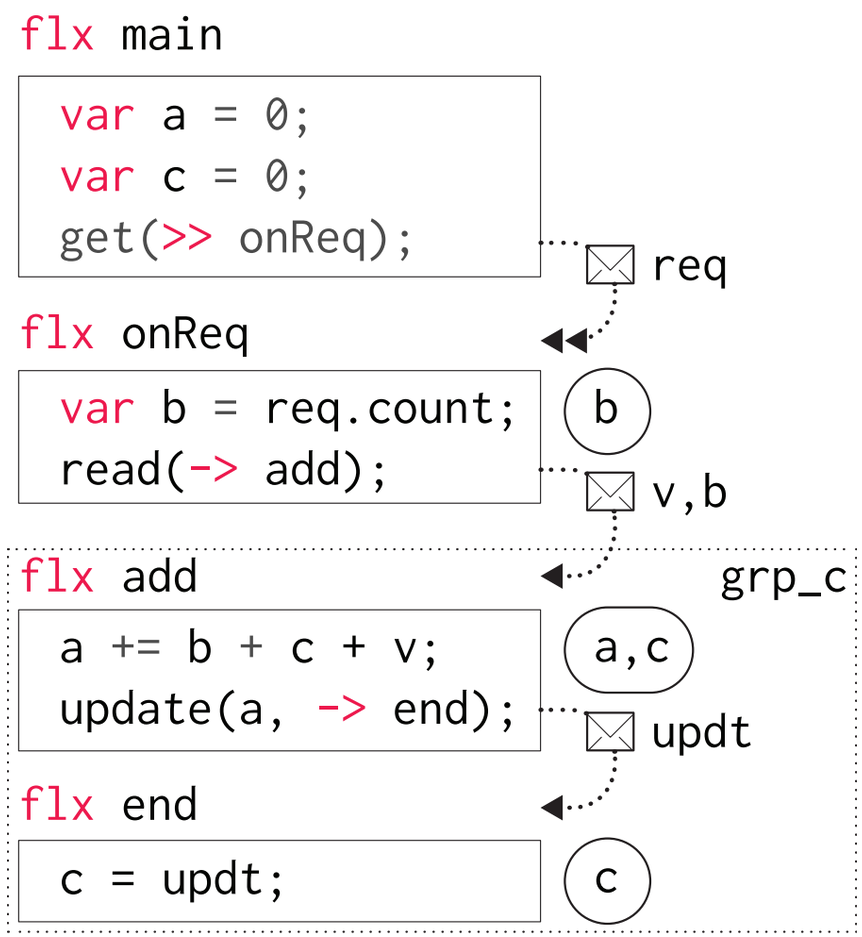
PIPELINER

STREAM

2

Variable b is modified inside an handler, and read inside downstream handlers. This variable is propagated downstream.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

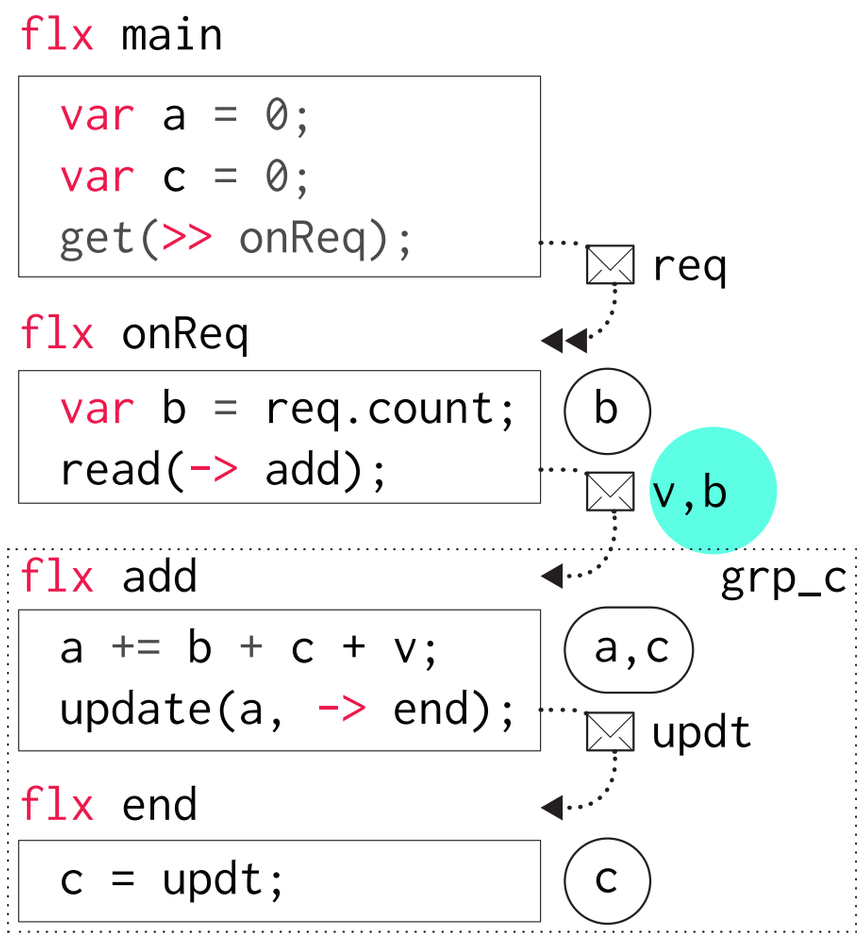


PIPELINER

STREAM

2 Variable b is modified inside an handler, and read inside downstream handlers. This variable is propagated downstream.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

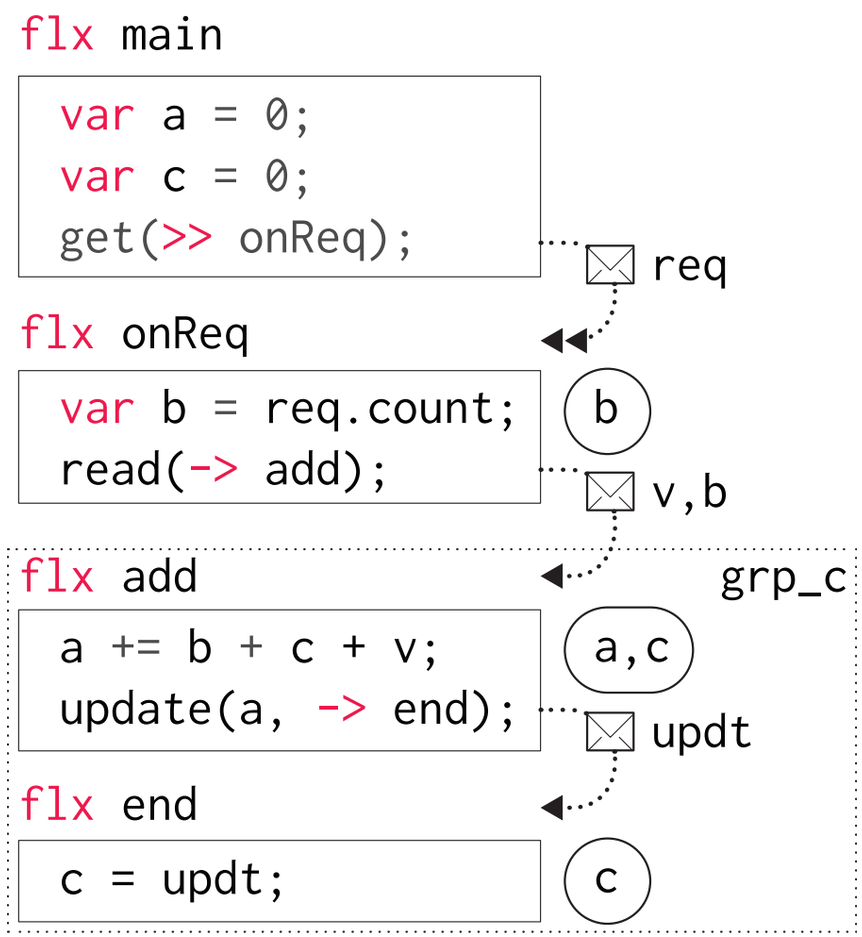


PIPELINER

SHARE ❸

Variable **c** is needed for modification by several handlers, or read by an upstream handler.
The handlers are gathered in the same group.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

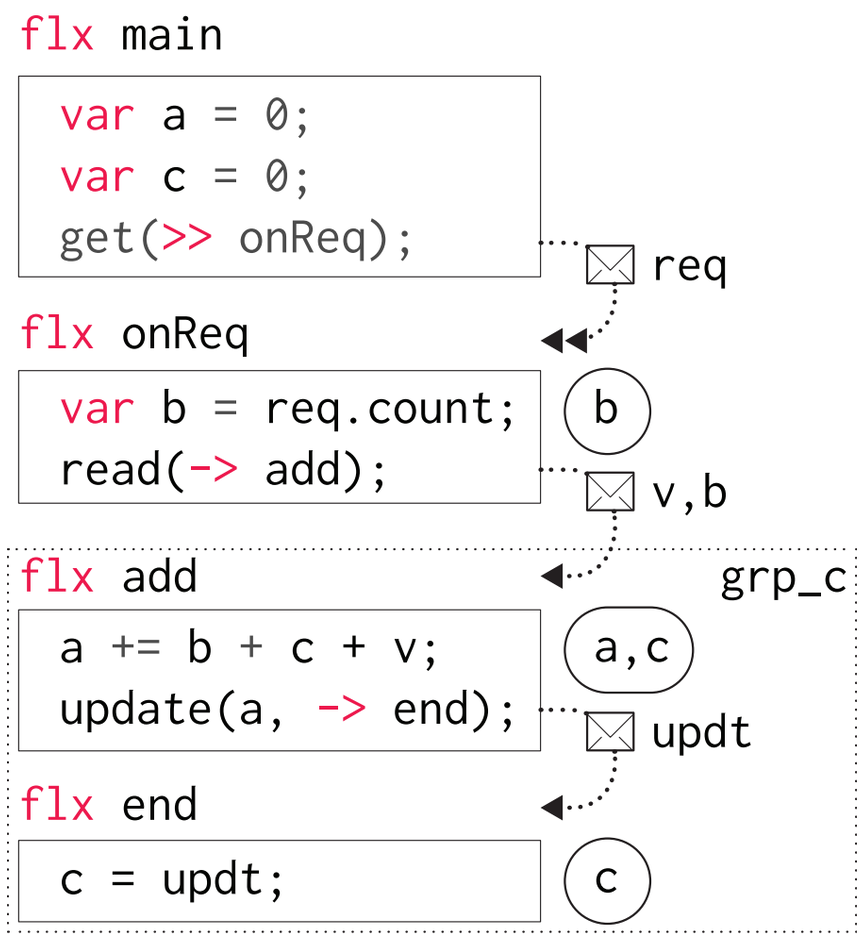


PIPELINER

SHARE ❸

Variable **c** is needed for modification by several handlers, or read by an upstream handler.
The handlers are gathered in the same group.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

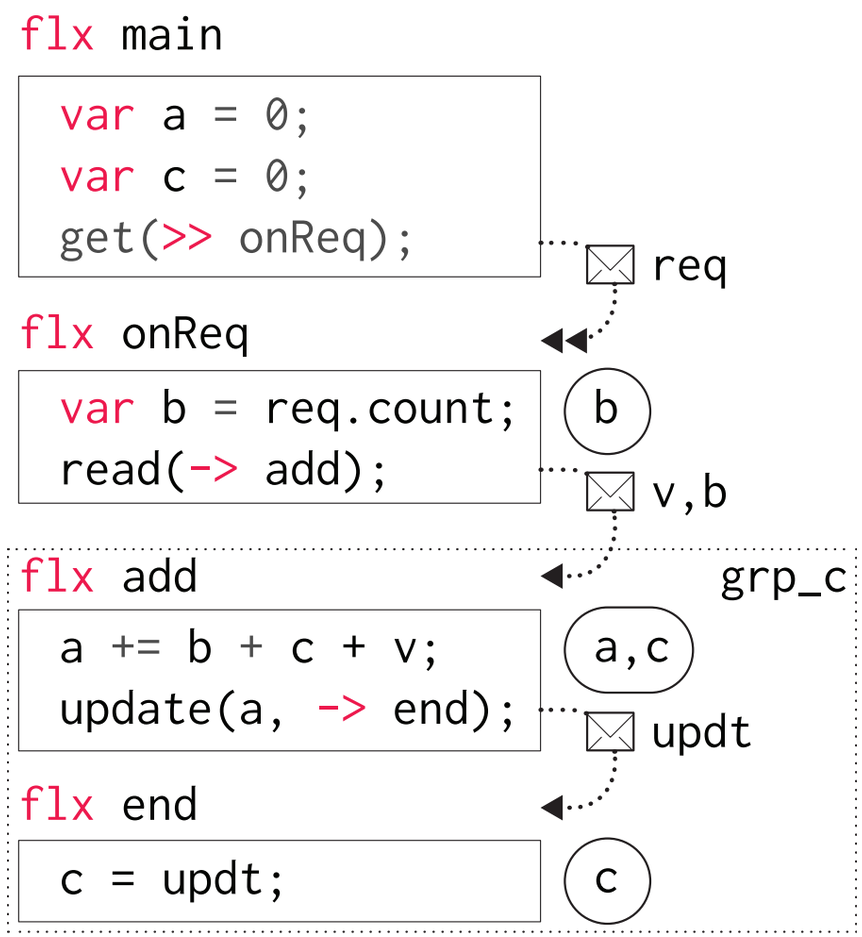


PIPELINER

SHARE ❸

Variable **c** is needed for modification by several handlers, or read by an upstream handler.
The handlers are gathered in the same group.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

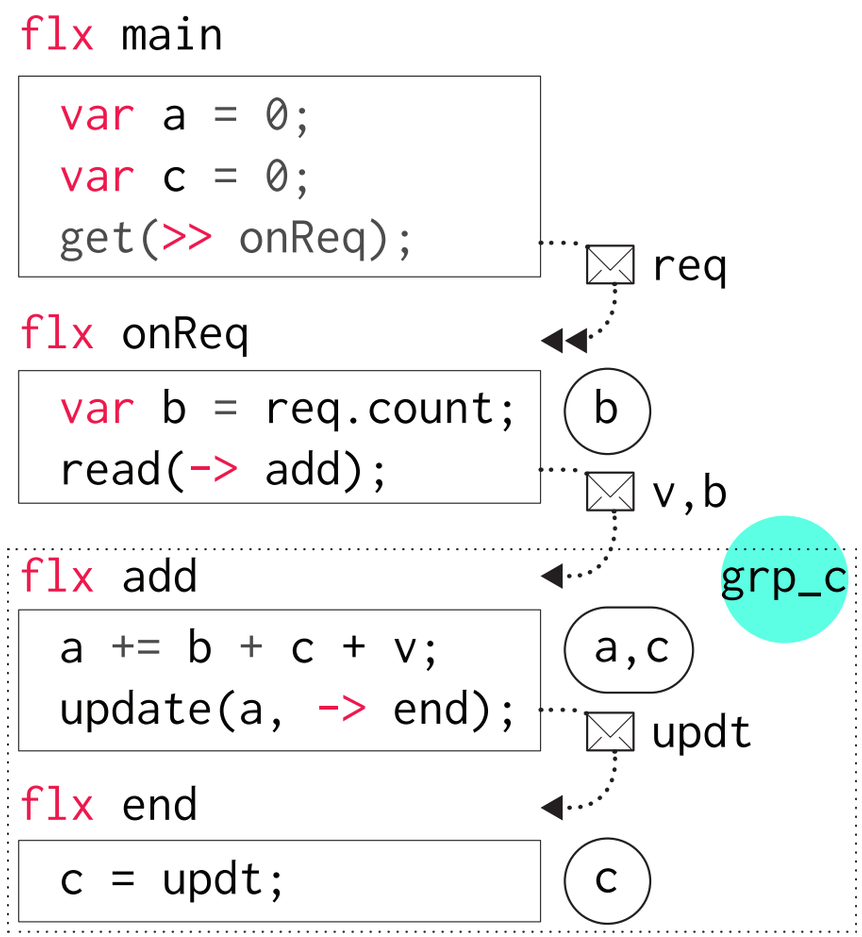


PIPELINER

SHARE ❸

Variable **c** is needed for modification by several handlers, or read by an upstream handler.
The handlers are gathered in the same group.

```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

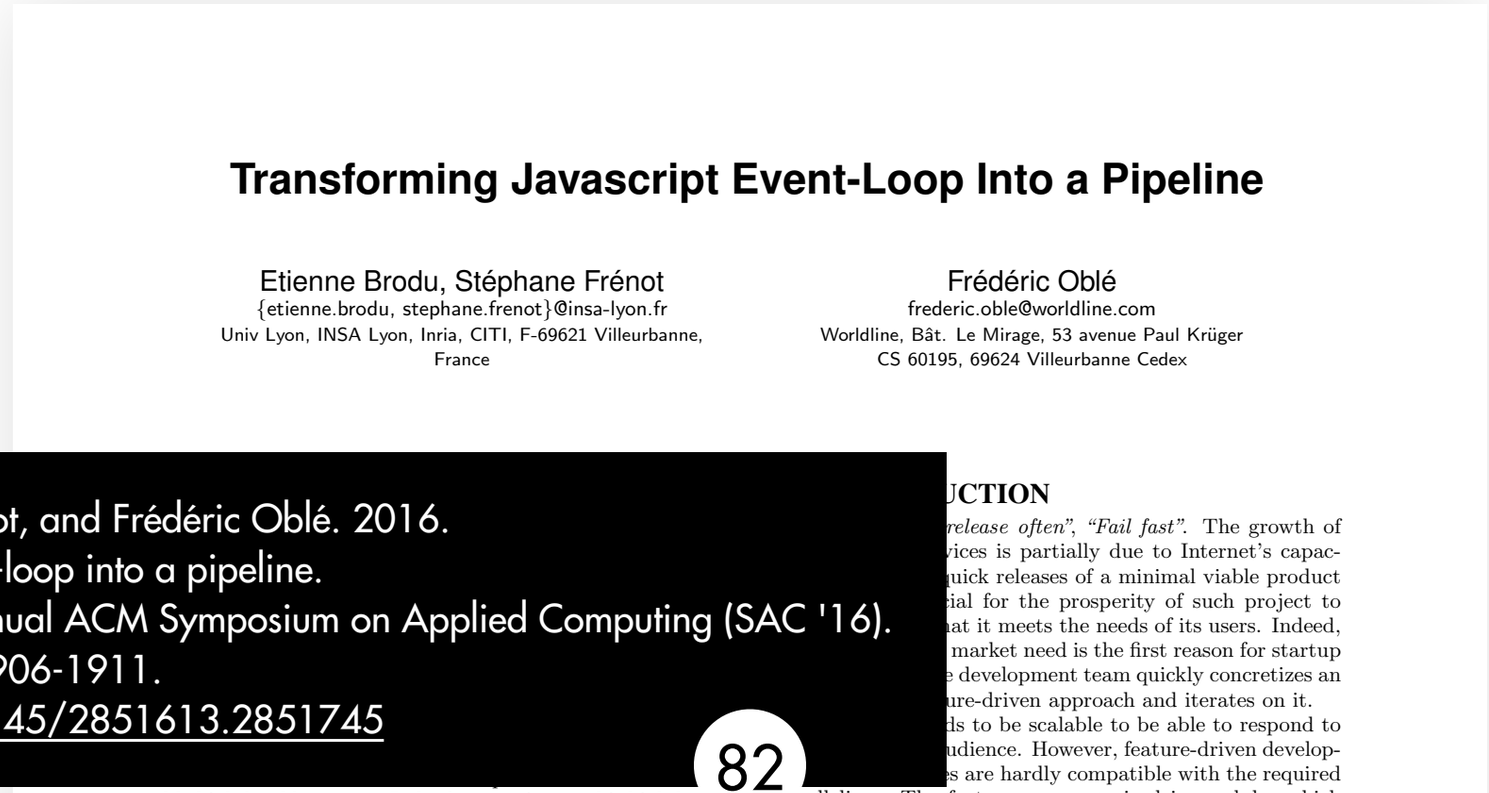


FLUXIONAL COMPILER

12K+ / 856K+

lines of code

GitHub `https://github.com/etnbrd/flx-compiler`



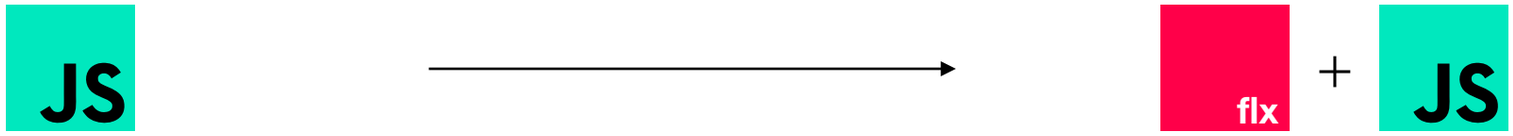
Transforming Javascript Event-Loop Into a Pipeline

Etienne Brodu, Stéphane Frénot
{etienne.brodu, stephane.frenot}@insa-lyon.fr
Univ Lyon, INSA Lyon, Inria, CITI, F-69621 Villeurbanne, France

Frédéric Oblé
frederic.oble@worldline.com
Worldline, Bât. Le Mirage, 53 avenue Paul Krüger
CS 60195, 69624 Villeurbanne Cedex

Etienne Brodu, Stéphane Frénot, and Frédéric Oblé. 2016.
Transforming JavaScript event-loop into a pipeline.
In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16).
ACM, New York, NY, USA, 1906-1911.
DOI: http://dx.doi.org/10.1145/2851613.2851745

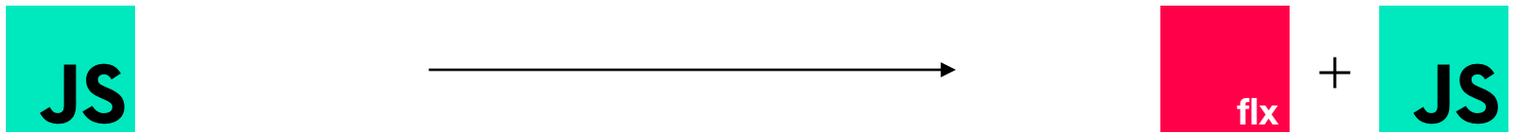
FLUXIONAL COMPILER




```
var app = require('express')(),
    fs = require('fs'),
    count = 0;

app.get('/', function handler(req, res){
  fs.readFile(__filename,
    function reply(err, data){
      count += 1;
      res.send(err || template(count, data));
    }
  );
});

app.listen(8080);
```

```
flx main & grp_res
>> handler [res]
  var app = require('express')(),
      fs = require('fs'),
      count = 0;
  app.get('/', >> handler);
  app.listen(8080);


flx handler
-> reply [res]
  function handler(req, res) {
    fs.readFile(__filename , -> reply)
  }


flx reply & grp_res {count, template}
-> null
  function reply(error , data) {
    count += 1;
    res.send(err || template(count, data));
```

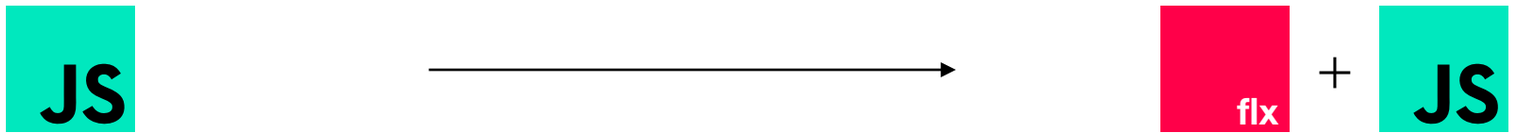
FLUXIONAL COMPILER




```
var app = require('express')(),
    fs = require('fs'),
    count = 0;

app.get('/', function handler(req, res){
  fs.readFile(__filename,
    function reply(err, data){
      count += 1;
      res.send(err || template(count, data));
    }
  );
});

app.listen(8080);
```

```
flx main & grp_res
>> handler [res]
  var app = require('express')(),
      fs = require('fs'),
      count = 0;
  app.get('/', >> handler);
  app.listen(8080);


flx handler
-> reply [res]
  function handler(req, res) {
    fs.readFile(__filename , -> reply)
  }


flx reply & grp_res {count, template}
-> null
  function reply(error , data) {
    count += 1;
    res.send(err || template(count, data));
```

FLUXIONAL COMPILER




```
var app = require('express')(),
    fs = require('fs'),
    count = 0;

app.get('/', function handler(req, res){
  fs.readFile(__filename,
    function reply(err, data){
      count += 1;
      res.send(err || template(count, data));
    }
  );
});

app.listen(8080);
```

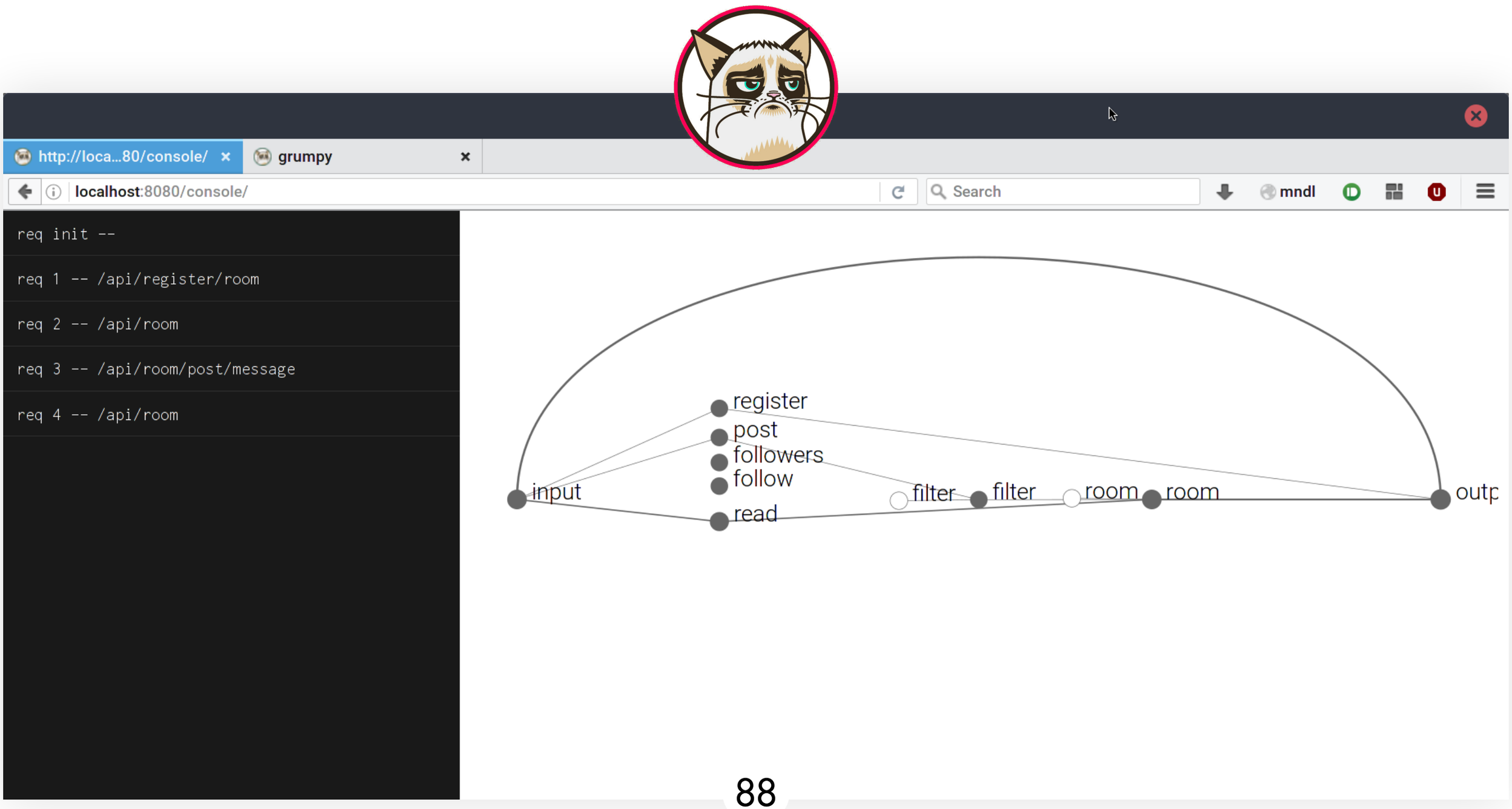
```
flx main & grp_res
>> handler [res]
  var app = require('express')(),
      fs = require('fs'),
      count = 0;
  app.get('/', >> handler);
  app.listen(8080);


flx handler
-> reply [res]
  function handler(req, res) {
    fs.readFile(__filename , -> reply)
  }


flx reply & grp_res {count, template}
-> null
  function reply(error , data) {
    count += 1;
    res.send(err || template(count, data));
```

Evaluation

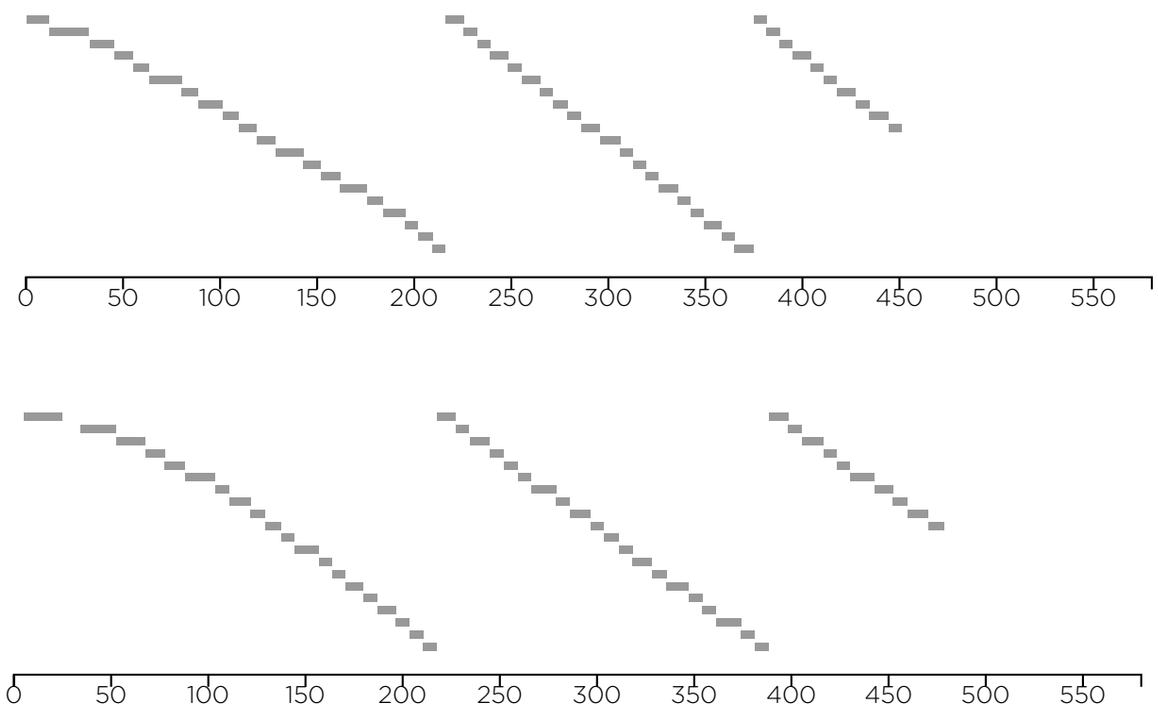
EVALUATIONS

1. FLUXIONAL EXECUTION MODEL
2. COMPILERS
   1. DUE COMPILER
   2. FLUXIONAL COMPILER

GRUMPY


http://loca...80/console/
grumpy
localhost:8080/console/
Search
mndl
req init --
req 1 -- /api/register/room
req 2 -- /api/room
req 3 -- /api/room/post/message
req 4 -- /api/room
register
post
followers
follow
read
input
filter
filter
room
room
outp

FLUXIONAL EXECUTION MODEL



VANILLA

max	min	average	total	transition
20ms	6ms	9ms	451ms	0.058ms

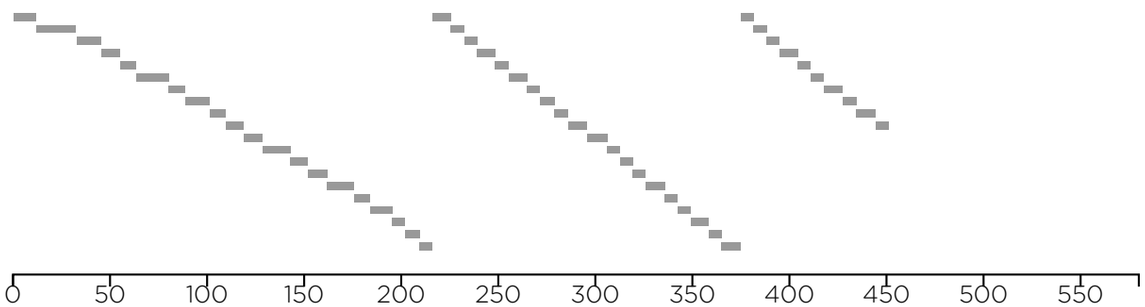
FLUXIONS

max	min	average	total	transition
20ms	7ms	9ms	473ms	0.31ms

OVERHEAD

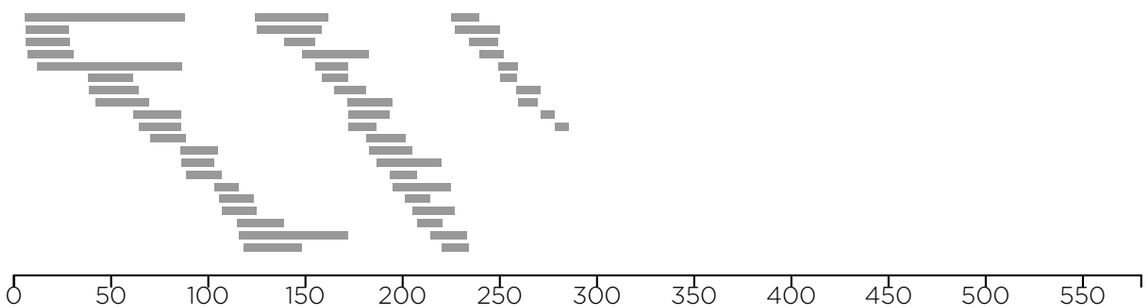
×5.3

FLUXIONAL EXECUTION MODEL



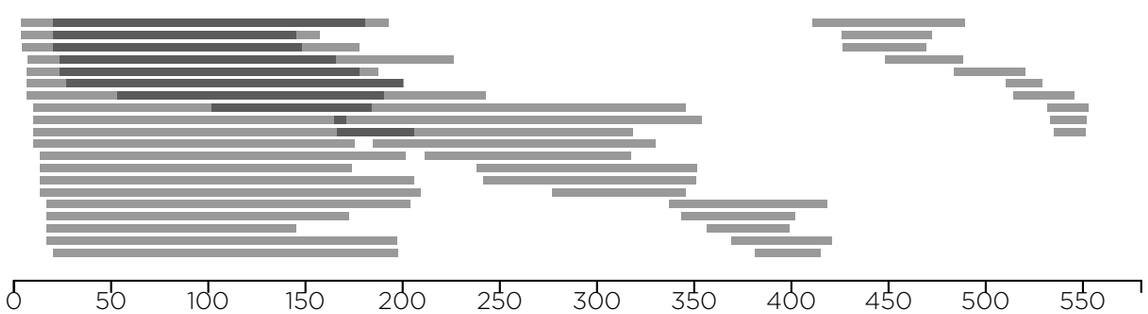
VANILLA

max	min	average	total
20ms	6ms	9ms	451ms



5 WORKERS PARALLEL

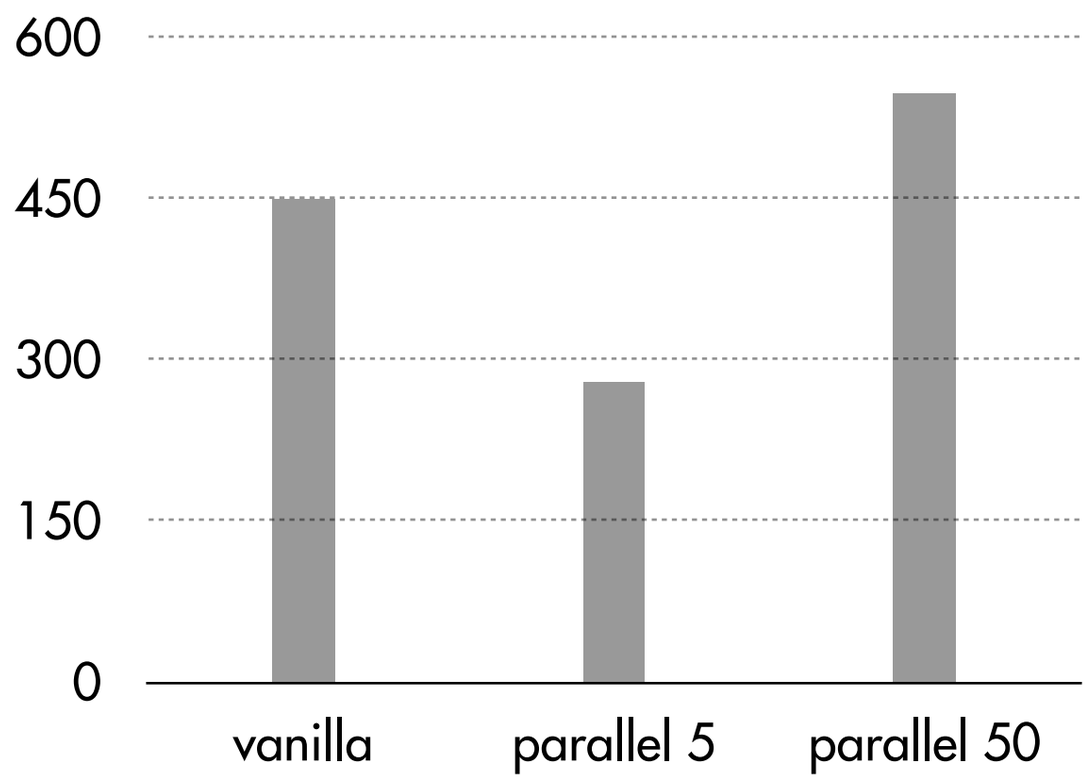
max	min	average	total
82ms	7ms	22ms	280ms



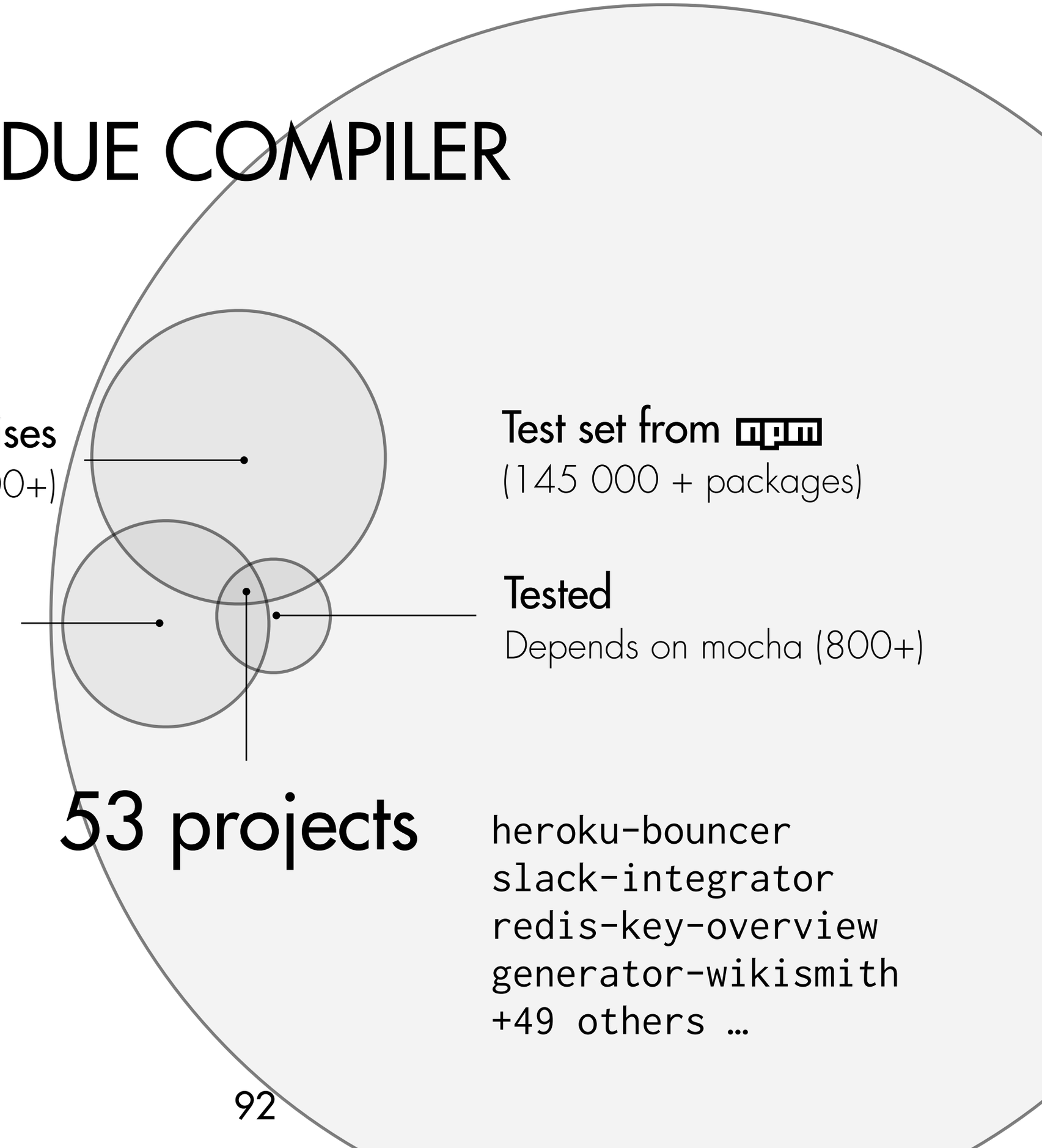
50 WORKERS PARALLEL

max	min	average	total
244ms	16ms	127ms	549ms

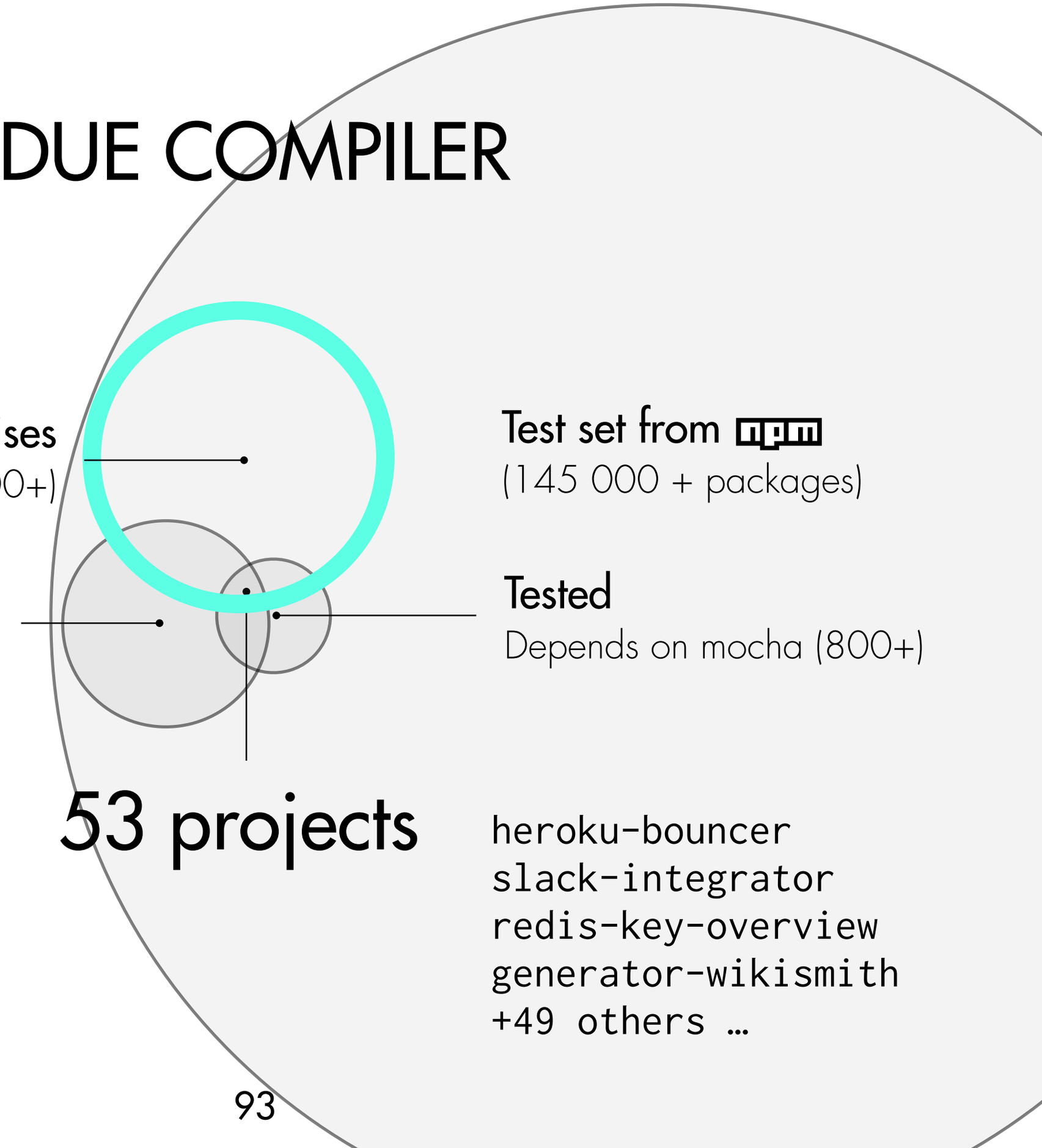
FLUXIONAL EXECUTION MODEL



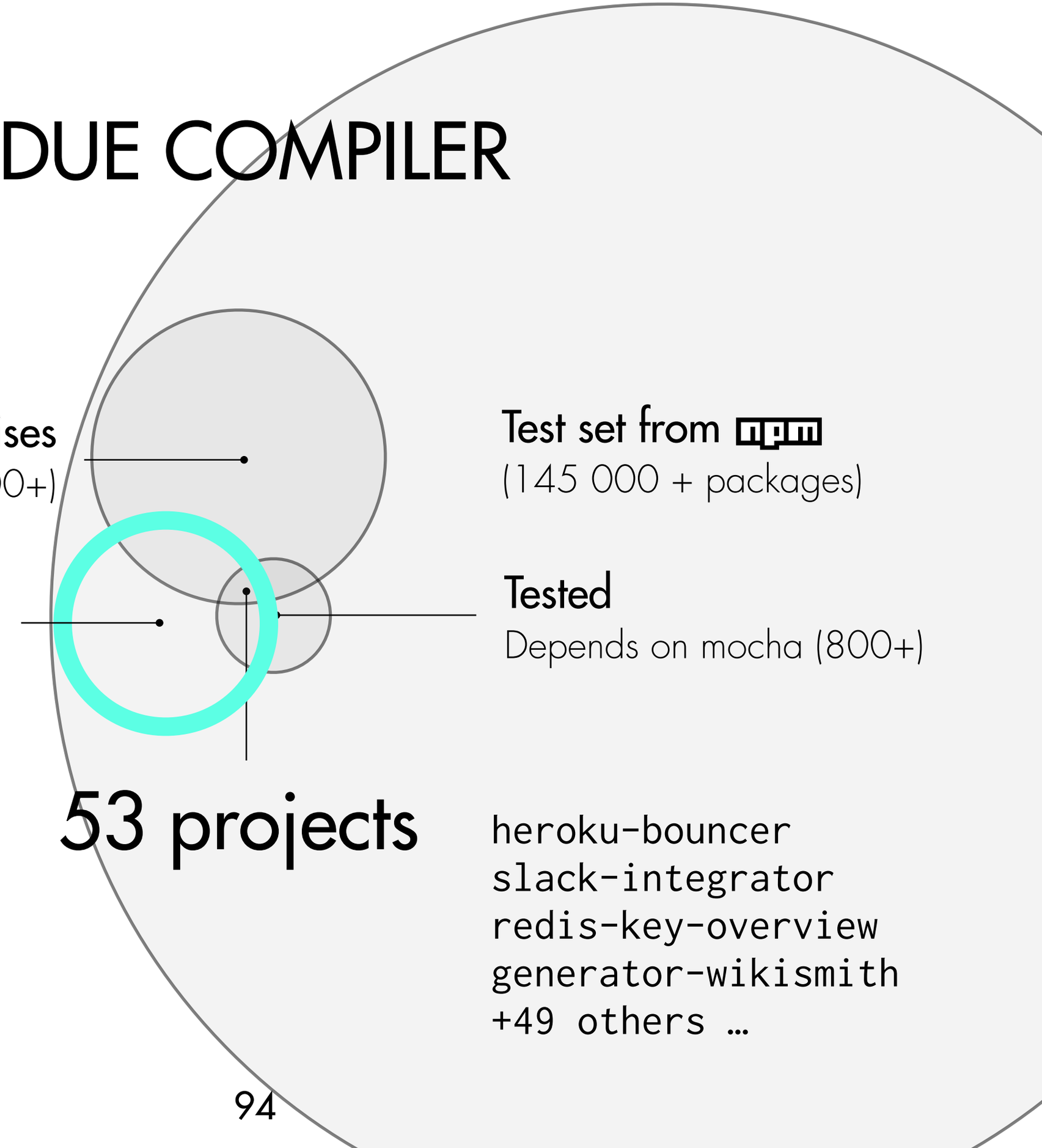
DUE COMPILER



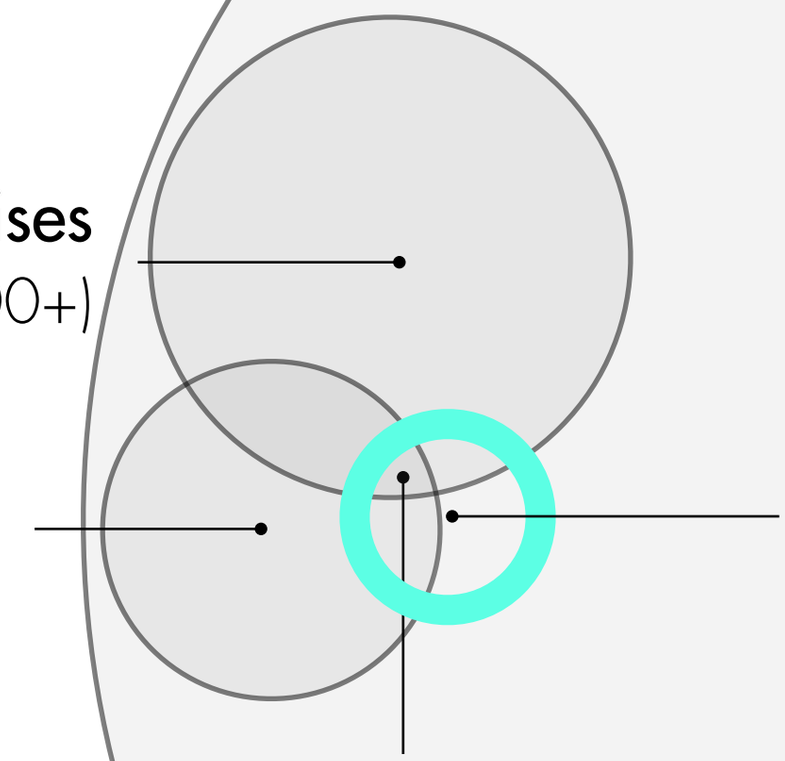
DUE COMPILER



DUE COMPILER



DUE COMPILER



Don't use Promises
Depends on Q or Async (15 000+)

Test set from npm
(145 000 + packages)

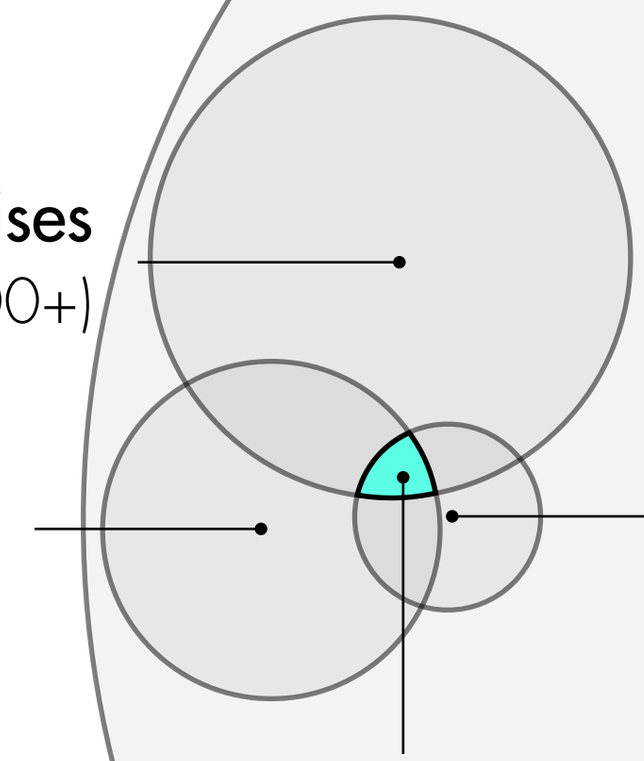
Web Application
Depends on express (4 000+)

Tested
Depends on mocha (800+)

53 projects

```
heroku-bouncer
slack-integrator
redis-key-overview
generator-wikismith
+49 others …
```

DUE COMPILER



Don't use Promises
Depends on Q or Async (15 000+)

Test set from npm
(145 000 + packages)

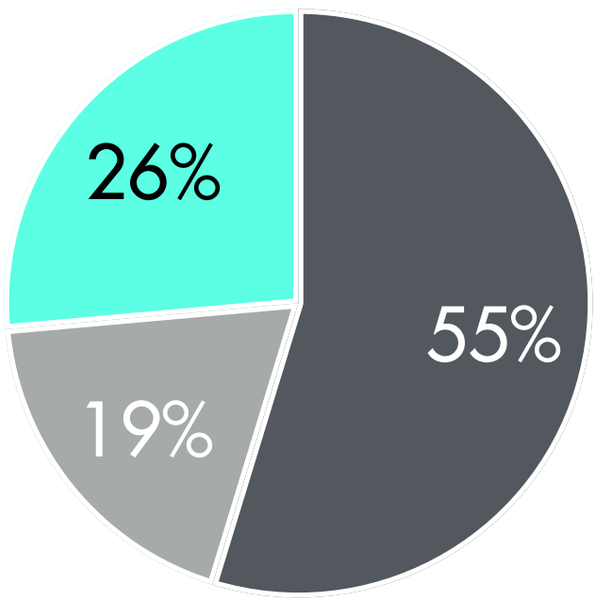
Web Application
Depends on express (4 000+)

Tested
Depends on mocha (800+)

53 projects

```
heroku-bouncer
slack-integrator
redis-key-overview
generator-wikismith
+49 others …
```

DUE COMPILER



- 29 no detected continuations
- 10 eval or with statements
- 14 successfully compiled

FLUXIONAL COMPILER

Tested on a small real application,
selected from previous test set.
`gifsockets-server`

ANALYZER STEP
1 fluxion successfully identified.

PIPELINER STEP
Compilation result executes as expected
after manual modifications.

https://github.com/twolfson/gifsockets-server

FLUXIONAL COMPILER

Raises two problems

- ✕ Statical detection of variable aliasing
- ✕ Statical serialisation of closures

FLUXIONAL COMPILER

Statically detecting variable aliasing

one level indirection

```
var a = {
  modified: false
};


var b = a;

b.modified = true;
a.modified === true;
```

two level indirection

```
var a = {
  modified: false
};

// i comes from user input
var b = handlers[i](a);

b.modified = true;
a.modified === true;
```

FLUXIONAL COMPILER

Statically detecting variable aliasing

one level indirection

```
var a = {
  modified: false
};

var b = a;

b.modified = true;
a.modified === true;
```

two level indirection

```
var a = {
  modified: false
};

// i comes from user input
var b = handlers[i](a);

b.modified = true;
a.modified === true;
```

FLUXIONAL COMPILER

Statically detecting variable aliasing

one level indirection

```
var a = {
  modified: false
};


var b = a;

b.modified = true;
a.modified === true;
```

two level indirection

```
var a = {
  modified: false
};

// i comes from user input
var b = handlers[i](a);

b.modified = true;
a.modified === true;
```

FLUXIONAL COMPILER

Statically detecting variable aliasing

one level indirection

```
var a = {
  modified: false
};


var b = a;

b.modified = true;
a.modified === true;
```

two level indirection

```
var a = {
  modified: false
};

// i comes from user input
var b = handlers[i](a);

b.modified = true;
a.modified === true;
```

FLUXIONAL COMPILER

Statically detecting variable aliasing

one level indirection

```
var a = {
  modified: false
};


var b = a;

b.modified = true;
a.modified === true;
```

two level indirection

```
var a = {
  modified: false
};

// i comes from user input
var b = handlers[i](a);

b.modified = true;
a.modified === true;
```

FLUXIONAL COMPILER

Statically detecting variable aliasing

one level indirection

```
var a = {
  modified: false
};

var b = a;

b.modified = true;
a.modified === true;
```

two level indirection

```
var a = {
  modified: false
};

// i comes from user input
var b = handlers[i](a);

b.modified = true;
a.modified === true;
```

FLUXIONAL COMPILER

Statically detecting variable aliasing

one level indirection

```
var a = {
  modified: false
};


var b = a;

b.modified = true;
a.modified === true;
```

two level indirection

```
var a = {
  modified: false
};

// i comes from user input
var b = handlers[i](a);

b.modified = true;
a.modified === true;
```

Conclusion

1

THE PIVOT DILEMMA

The risk of failing in the transition from development productivity to execution efficiency.

2

PROPOSITION

An intermediate language for dual representation

3

IMPLEMENTATIONS

Fluxional execution model

Due and Fluxional compilers

1 FLUXIONAL EXECUTION MODEL

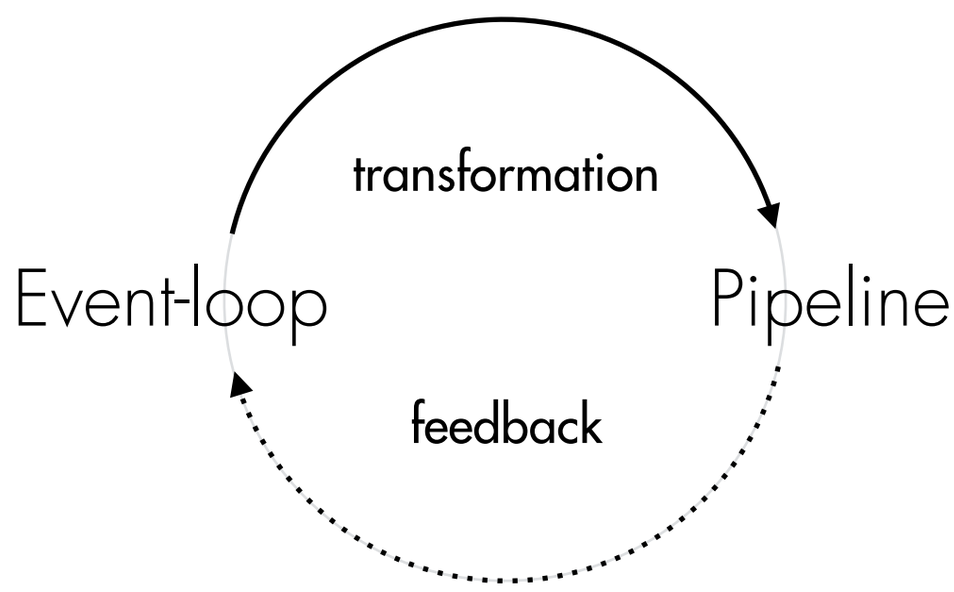
Compatible with both programming models

2 FLUXIONAL COMPILER

From one programming model to the other

1. RUPTURE POINTS
2. INDEPENDENCE

development productivity



execution efficiency

PERSPECTIVES

1

JUST IN TIME COMPILER

Dynamic analysis to replace static analysis.

PERSPECTIVES

2

CATEGORISATION OF STREAM

Qualification of the streams in term of policy and bandwidth to adjust the system.

PERSPECTIVES

3

LIVE MIGRATION OF FLUXIONS

The dependencies of fluxions are detailed so as to be able to migrate them to cope with traffic evolutions and spikes.

Thank you.

Questions.