

Liquefy the cloud

worldline
e-payment services

INSA
LYON

citi
lab

Inria
INVENTORS FOR THE DIGITAL WORLD

Etienne Brodu, Stéphane Frenot, Frédéric Oblé, Fabien Cellier

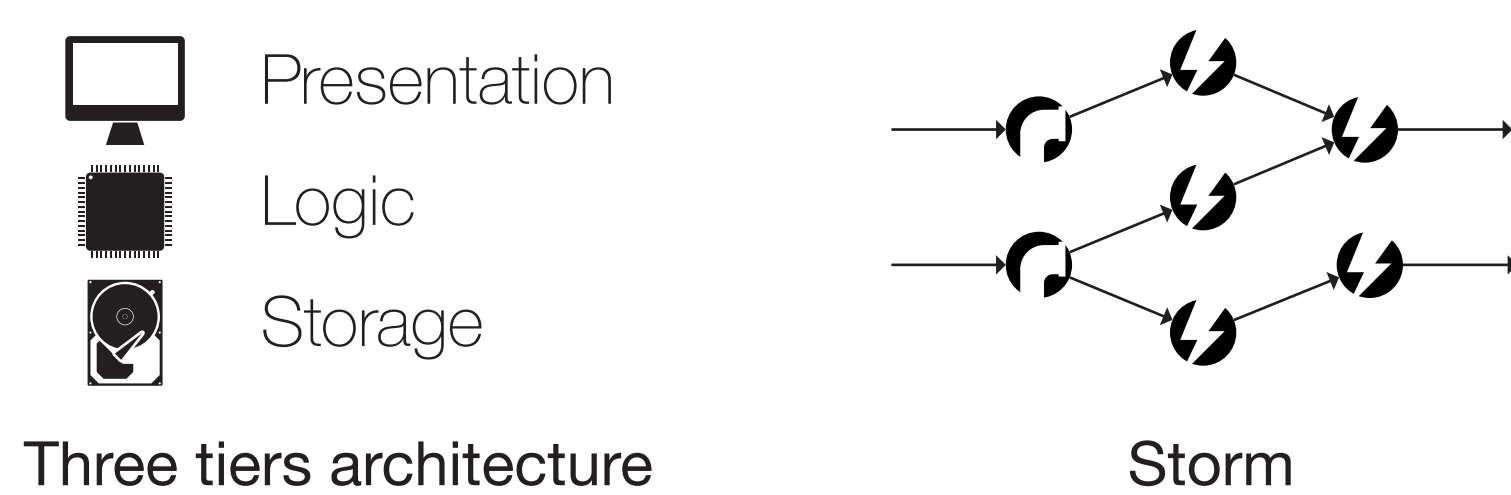
etienne.brodu@insa-lyon.fr, stephane.frenot@insa-lyon.fr, frederic.oble@worldline.com, fabien.cellier@worldline.com

How to abstract web services' usage variation from developpement ?

A popular web service might grow from thousands to millions of users in a matters of days.

To react to such variation of usage, they have to be scalable.

The classical approaches - the **three tiers** architecture, frameworks like **storm** or **languages** like Erlang - allow developer to split web services into well defined parts in order to be scalable.



Instead we want to **automatically** split a web service into **stateless parts**, and make them communicate by **volatile data streams**.

The persistence is decoupled from the logic and managed into-the messaging system.

Statelessness and volatility assure the web service to be **frictionless***, and the association with this fine decomposition, make it **scalable**.

***Frictionless** : independent from the hardware allowing it to be moved at runtime without relocation of ressources.

In a context of **data-stream oriented** web services written in **javascript**, we want

- + to create a **frictionless*** execution model and
- + to **transpile** standard web services into this model

Example : a visit counter

// simple javascript

```
var count = {};  
app.get('/:id', function(req, res){  
  return router(req.params.id, res);  
});  
  
function router(userId, res) {  
  data = count[userId] || 0;  
  return process(userId, data, res);  
}  
  
function process(userId, inputData, res) {  
  value = inputData + 1;  
  response = 'Count(' + userId + ') : ' + value;  
  return store(userId, value, response, res);  
}  
  
function store(userId, value, outputData, res){  
  count[userId] = value;  
  return send(outputData, res);  
}  
  
function send(outputData, res) {  
  return res.send(outputData);  
}  
  
app.listen(port);
```

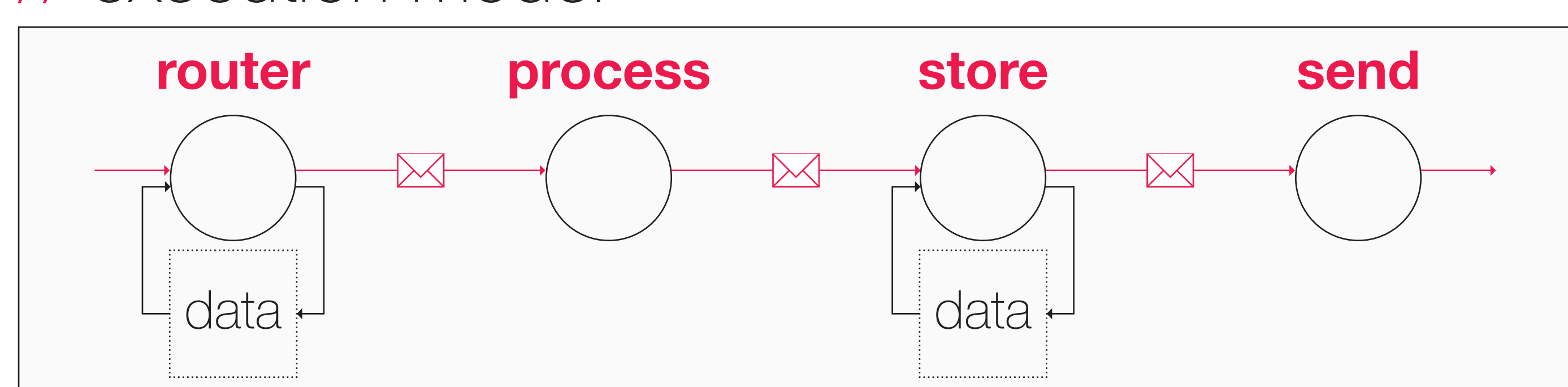
Extraction

// frictionless model

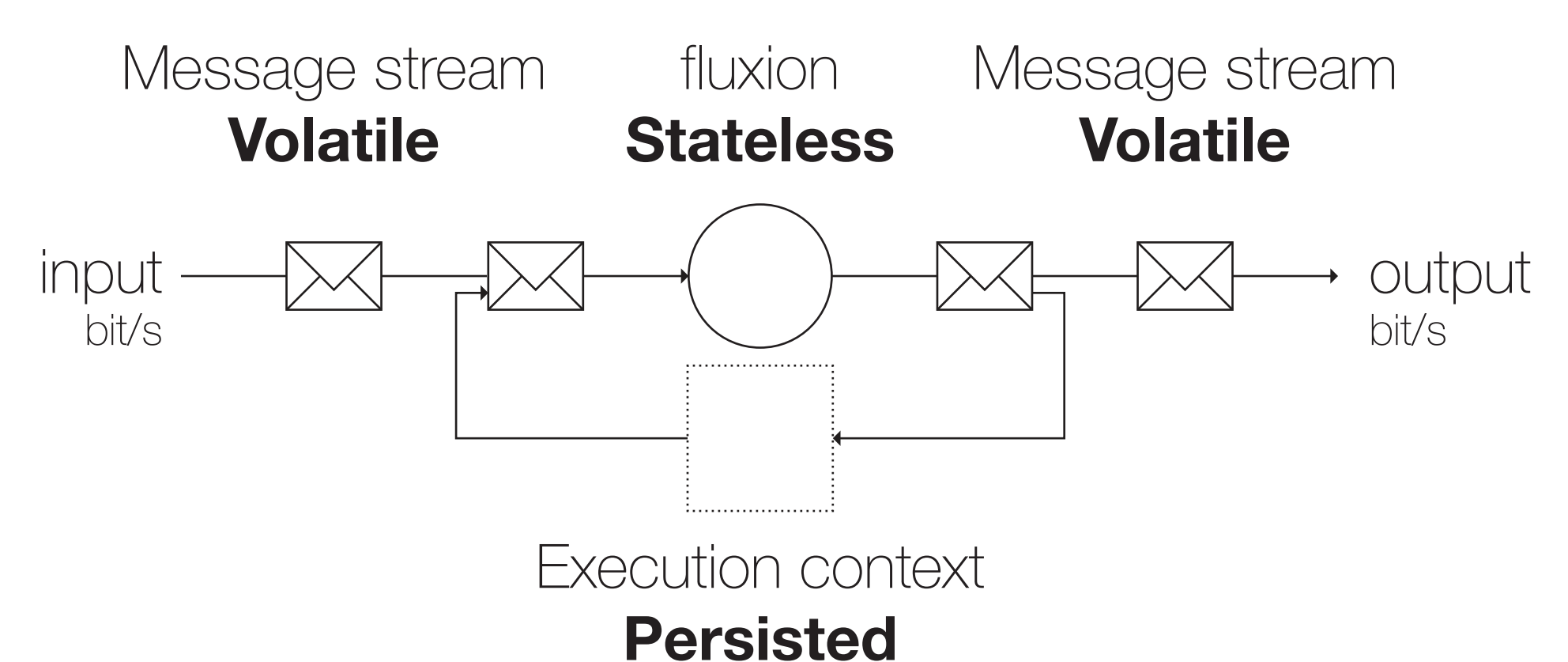
```
input >> router  
router >> process | count  
process >> store  
store >> send | count  
send >> output
```

Execution

// execution model



Execution model



Our **execution model** is composed of :

Fluxion, stateless parts, **listens** for, **modifies** and **sends** messages to other fluxions.

Execution context are persisted **memory states** needed by fluxions.

Messaging system keeps tracks of fluxions, and delivers volatile message streams.

- + Binds context execution and messages.
- + Moves fluxions and contexts to balance load.

Progression & Objectives

We aim to transpile any javascript web service into one which can adapt dynamically to load.

Without the development constraints imposed by other approaches.

DONE a javascript library to express and execute a fluxional program written in javascript.

TODO enhance this library with the automatic migration of fluxions, and create a langage with a compiler to javascript.