

# **A Language-based Serverless Function Accelerator**

Emily Herbert

Arjun Guha

## What is serverless computing?

Approach to cloud computing...



without servers...



with servers

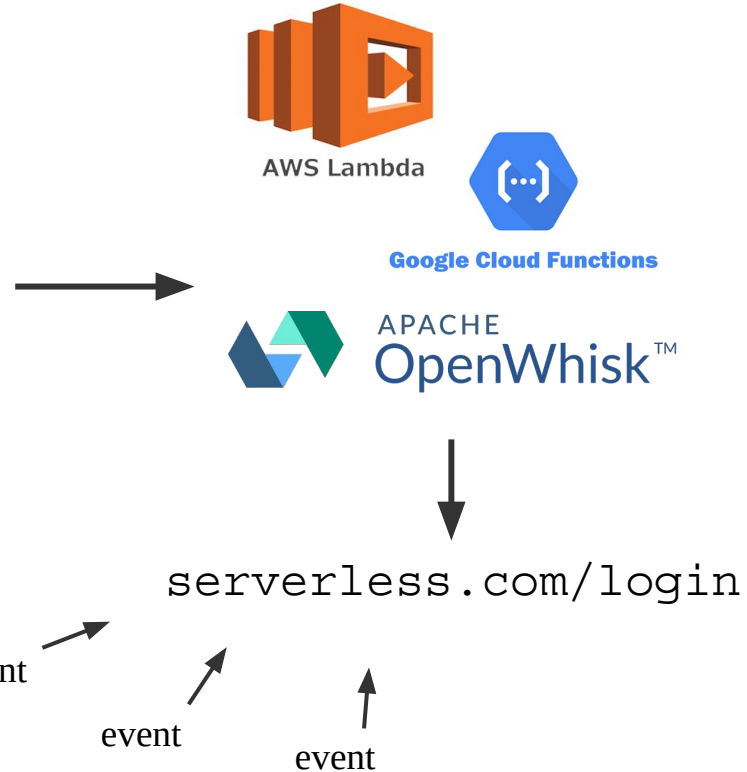


APACHE  
OpenWhisk™

## What is serverless computing?



```
1  function login(req) {  
2      function F(resp) {  
3          let u = req.body.username;  
4          let p = req.body.password;  
5          if(resp[u] === p) {  
6              respond('ok');  
7          } else {  
8              respond('error');  
9          }  
10     }  
11     get('passwords.json', F);  
12 }
```



## Problems with serverless computing

- JavaScript is ill-suited for serverless computing
  - Can consume a significant amount of time and memory
  - Require an operating system sandbox
- These sandboxes incur slowdowns <sup>1</sup>

<sup>1</sup> Mohammad Shahrade, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In IEEE/ACM International Symposium on Microarchitecture (MICRO)

## Rust as an alternative

- Boucher et al. present a serverless platform that runs functions written in Rust<sup>2</sup>
- Leverages Rust's language-level guarantees to run multiple serverless functions in one process

Microservices		Latency (μs)		Throughput
Resident?	Isolation	Median	99%	(M invoc/s)
Warm-start	Process	8.7	27.3	0.29
	Language	1.2	2.0	5.4
Cold-start	Process	2845.8	15976.0	–
	Language	38.7	42.2	–

**Table 1: Microservice invocation performance**

<sup>2</sup> Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the “Micro” back in microservices. In USENIX Annual Technical Conference (ATC).

## Rust as an alternative

- Rust is difficult to learn
- Rust's safety alone is not strong enough for serverless computing
  - CPU monopolization
  - deadlocks
  - memory leaks
  - ...

## Containerless

- Serverless function accelerator
- Seeks to improve serverless computing performance
- Uses **language-based isolation** instead of container-based isolation

## Containerless

- **Transforms JavaScript code to Rust code**  
by means of a **traced-based intermediate representation**
- Employs the Rust type system to ensure memory-safety  
(language-based isolation)
- Runs serverless functions using the new language-based isolation

JavaScript  $\longrightarrow$  trace IR  $\longrightarrow$  Rust



## Containerless

- Why use a IR?
- Compiling directly would suffer from **impedance mismatch**
  - Dynamic types v. static types
  - Garbage collection v. explicit memory management
  - Pointer aliasing
  - ...

## Containerless

- Domain specific
- Utilizes common features of serverless functions
  - idempotent
  - short-lived
- Not a general purpose JavaScript to Rust compiler

## Components

Three general components:


1. JavaScript to IR
2. IR to Rust
3. invoker


## JavaScript to IR

- IR is **trace-tree built over multiple executions of the function**
- Similar to an execution trace, but a tree

## JavaScript to IR

Key features:

1. ~~Functions~~
2. Closures (**closure**)
3. Unknown behavior ()
4. Callbacks (*cb*) and events (**event**)

<b>Events</b>	
<i>ev</i> ::=	'listen'   'get'   'post'   ...
<b>Callbacks</b>	
<i>cb</i> ::=	callback( $x_1 \dots x_n$ ) <i>blk</i>
<b>l-values</b>	
<i>lval</i> ::=	<i>x</i> Variable
	<i>t.f</i> Field
	<i>*t.x</i> Variable in closure
<b>Blocks</b>	
<i>blk</i> ::=	{ <i>t</i> <sub>1</sub> ; ...; <i>t</i> <sub><i>n</i></sub> }
<b>Operators</b>	
<i>op</i> ::=	+   -   *   ...
<b>Trace trees</b>	
<i>t</i> ::=	<i>c</i> Constant
	<i>x</i> Variable
	<i>t.f</i> Read field
	<i>t</i> <sub>1</sub> <i>op</i> <i>t</i> <sub>2</sub> Binary operation
	if ( <i>t</i> <sub>1</sub> ) <i>blk</i> <sub>1</sub> else <i>blk</i> <sub>2</sub> Conditionals
	while ( <i>t</i> <sub>1</sub> ) <i>blk</i> Loops
	let <i>x</i> = <i>t</i> ; Variable declaration
	<i>lval</i> = <i>t</i> ; Assignment and mutation
	<i>blk</i> Block
	{ <i>f</i> <sub>1</sub> : <i>t</i> <sub>1</sub> , ... , <i>f</i> <sub><i>n</i></sub> : <i>t</i> <sub><i>n</i></sub> } Object literal
	 Unknown behavior
	event( <i>ev</i> , <i>t</i> <sub><i>a</i></sub> , <i>t</i> <sub><i>c</i></sub> , <i>cb</i> ) Event handler
	respond( <i>x</i> ) Response
	closure( & <i>x</i> <sub>1</sub> , ... , & <i>x</i> <sub><i>n</i></sub> ) Closure object
	& <i>t.x</i> Read from closure

## JavaScript to IR

### 1. Instrument function with trace-building runtime statements

```
1  let c = require('containerless');
2
3  function main(req) {
4    function F(resp) {
5      let u = req.body.username;
6      let p = req.body.password;
7      if (resp[u] === p) {
8        c.respond('ok');
9      } else {
10       c.respond('error');
11     }
12   }
13   c.get('passwords.json', F);
14 }
15
16 c.listen(main);
```

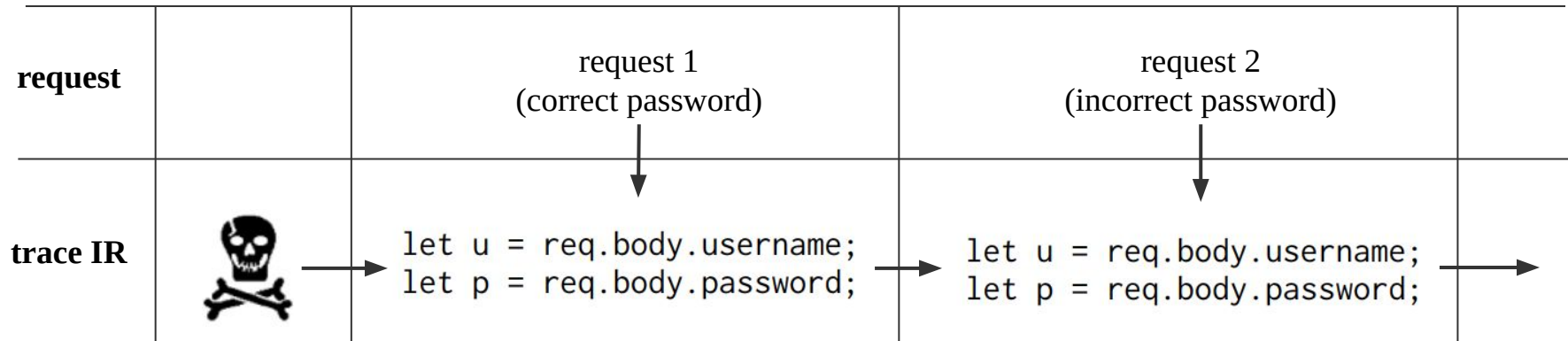


```
1  let c = require('containerless');
2  let t = require('containerless/tracing');
3
4  function main(req) {
5    let [_req] = t.popArgs();
6    function F(resp) {
7      let [_resp] = t.popArgs();
8      let _clos = t.popClosure();
9      t.let('req', t.getClos(_clos, 'req'));
10     let u = req.body.username;
11     t.let('u', t.get(t.get(t.id('req'), 'body'), 'username'));
12     let p = req.body.password;
13     t.let('p', t.get(t.get(t.id('req'), 'body'), 'password'));
14     t.if(t.eq(t.vget(_resp, t.id('u')), t.id('p')));
15     if (resp[u] === p) {
16       t.ifTrue();
17       t.pushArgs(t.str('ok'));
18       c.respond('ok');
19       t.popResult();
20     } else {
21       t.ifFalse();
22       t.pushArgs(t.str('error'));
23       c.respond('error');
24       t.popResult();
25     }
26     t.exitIf();
27     t.exitFunction(t.undefined);
28   }
29   t.let('F', t.closure({ 'req': _req }));
30   t.pushArgs([t.str('passwords.json'), t.id('F')]);
31   c.get('passwords.json', F);
32   t.popResult();
33 }
34
35 c.listen(main);
```

## JavaScript to IR

2. Execute function multiple times to build a trace tree
  - Linked with library
  - Builds incrementally
  - Tree fragments are merged

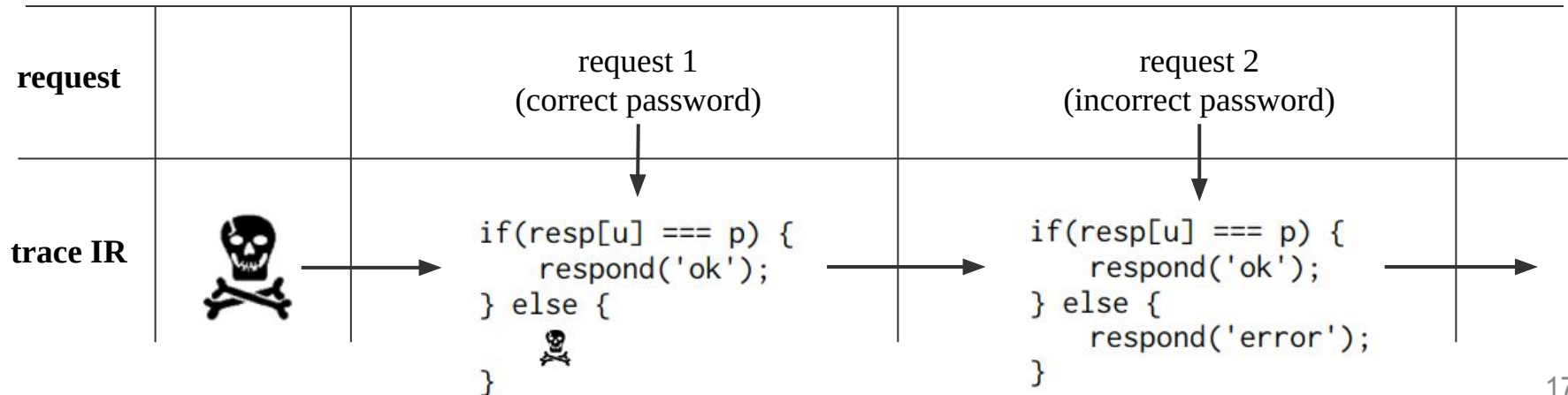
```
10      let u = req.body.username;  
11      t.let('u', t.get(t.get(t.id('req')), 'body'), 'username'));  
12      let p = req.body.password;  
13      t.let('p', t.get(t.get(t.id('req')), 'body'), 'password'));
```





```

14 t.if(t.eq(t.vget(_resp, t.id('u')), t.id('p')));
15 if (resp[u] === p) {
16     t.ifTrue();
17     t.pushArgs(t.str('ok'));
18     c.respond('ok');
19     t.popResult();
20 } else {
21     t.ifFalse();
22     t.pushArgs(t.str('error'));
23     c.respond('error');
24     t.popResult();
25 }
26 t.exitIf();
    
```



```


4  function main(req) {
5      let [_req] = t.popArgs();
6      function F(resp) {
7          let [_resp] = t.popArgs();
8          let _clos = t.popClosure();
9          t.let('req', t.getClos(_clos, 'req'));

```

```

27      t.exitFunction(t.undefine);
28      }
29      t.let('F', t.closure({ 'req': _req }));
30      t.pushArgs([t.str('passwords.json'), t.id('F')]);
31      c.get('passwords.json', F);
32      t.popResult();
33      }
34
35      c.listen(main);

```

request		request 1 (correct password)	request 2 (incorrect password)
trace IR		<p style="text-align: center;">↓</p> <pre> event('listen', [], closure(), callback (clos, req) {   event('get', ['passwords.json'], closure(&amp;req),     callback(clos, resp) {       let req = *(clos.req);       // body     }); }); </pre>	<p style="text-align: center;">↓</p> <pre> event('listen', [], closure event('get', ['passwords.   callback(clos, resp) {     let req = *(clos.req);     // body   }); }); </pre>

```
1 let c = require('containerless');
2 let t = require('containerless/tracing');
3
4 function main(req) {
5   let [_req] = t.popArgs();
6   function F(resp) {
7     let [_resp] = t.popArgs();
8     let _clos = t.popClosure();
9     t.let('req', t.getClos(_clos, 'req'));
10    let u = req.body.username;
11    t.let('u', t.get(t.get(t.id('req'), 'body'), 'username'));
12    let p = req.body.password;
13    t.let('p', t.get(t.get(t.id('req'), 'body'), 'password'));
14    t.if(t.eq(t.vget(_resp, t.id('u')), t.id('p')));
15    if (resp[u] === p) {
16      t.isTrue();
17      t.pushArgs(t.str('ok'));
18      c.respond('ok');
19      t.popResult();
20    } else {
21      t.ifFalse();
22      t.pushArgs(t.str('error'));
23      c.respond('error');
24      t.popResult();
25    }
26    t.exitIf();
27    t.exitFunction(t.undefined);
28  }
29  t.let('F', t.closure({ 'req': _req }));
30  t.pushArgs([t.str('passwords.json'), t.id('F')]);
31  c.get('passwords.json', F);
32  t.popResult();
33 }
34
35 c.listen(main);
```



### 3. Produce trace IR!

```
1 event('listen', [], closure(), callback (clos, req) {
2   event('get', ['passwords.json'],
3     closure(&req, callback(clos, resp) {
4     let req = *(clos.req);
5     let u = req.body.username;
6     let p = req.body.password;
7     if (resp[u] === p) {
8       respond('ok');
9     } else {
10      respond('error');
11    }
12  });
13 });
```

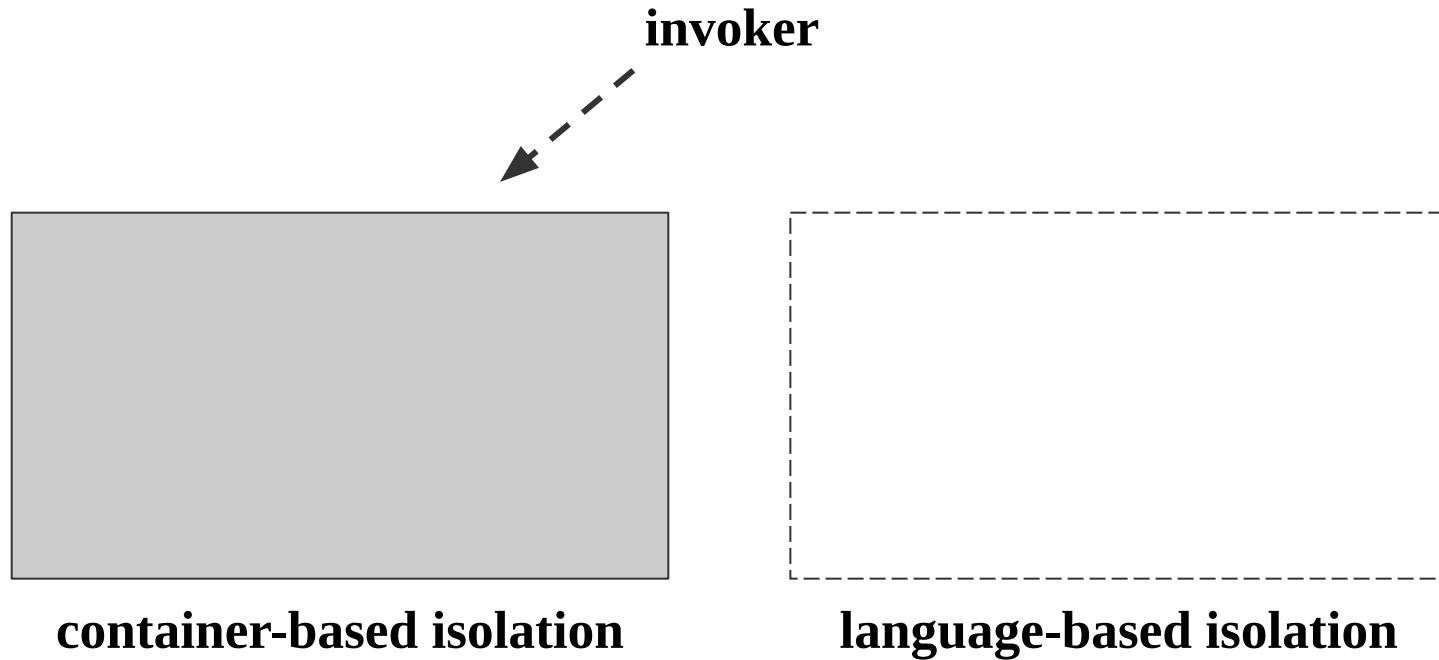
## IR to Rust

1. Transform callbacks in the trace IR to a state machine
2. Impose CPU and memory limits on the program
3. Inject all values into a **dynamic type**
4. Use **arena allocation** to resolve Rust lifetimes
5. Produce Rust code!

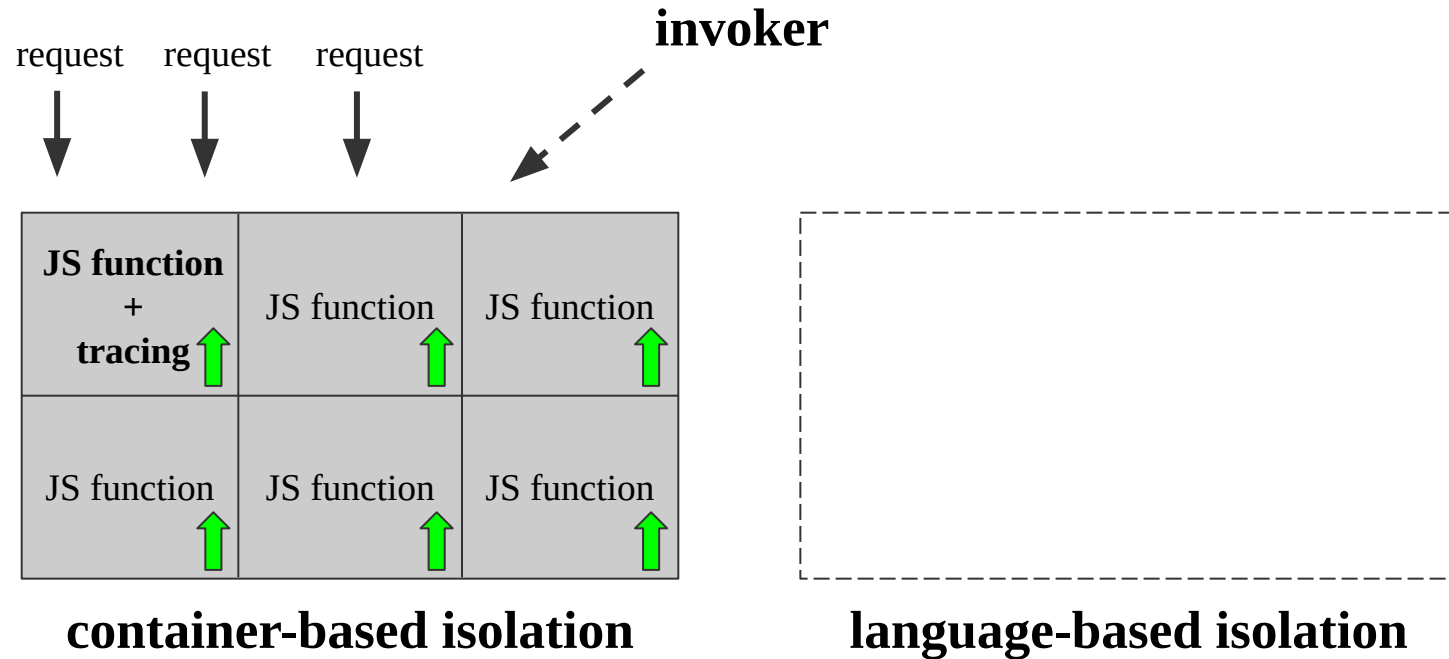


this is ok because  
of **idempotency**

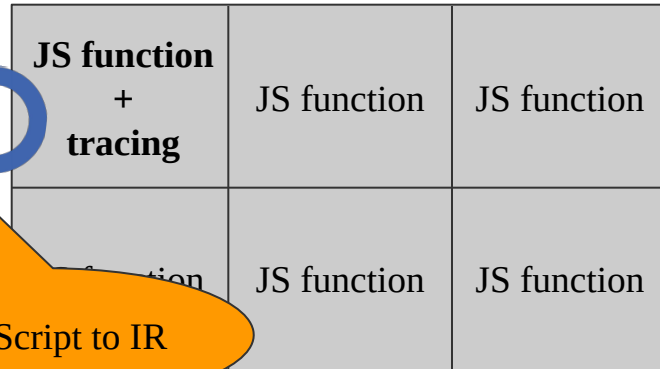
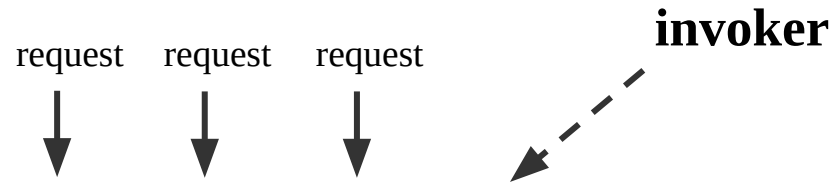
## Invoker



## Invoker



## Invoker

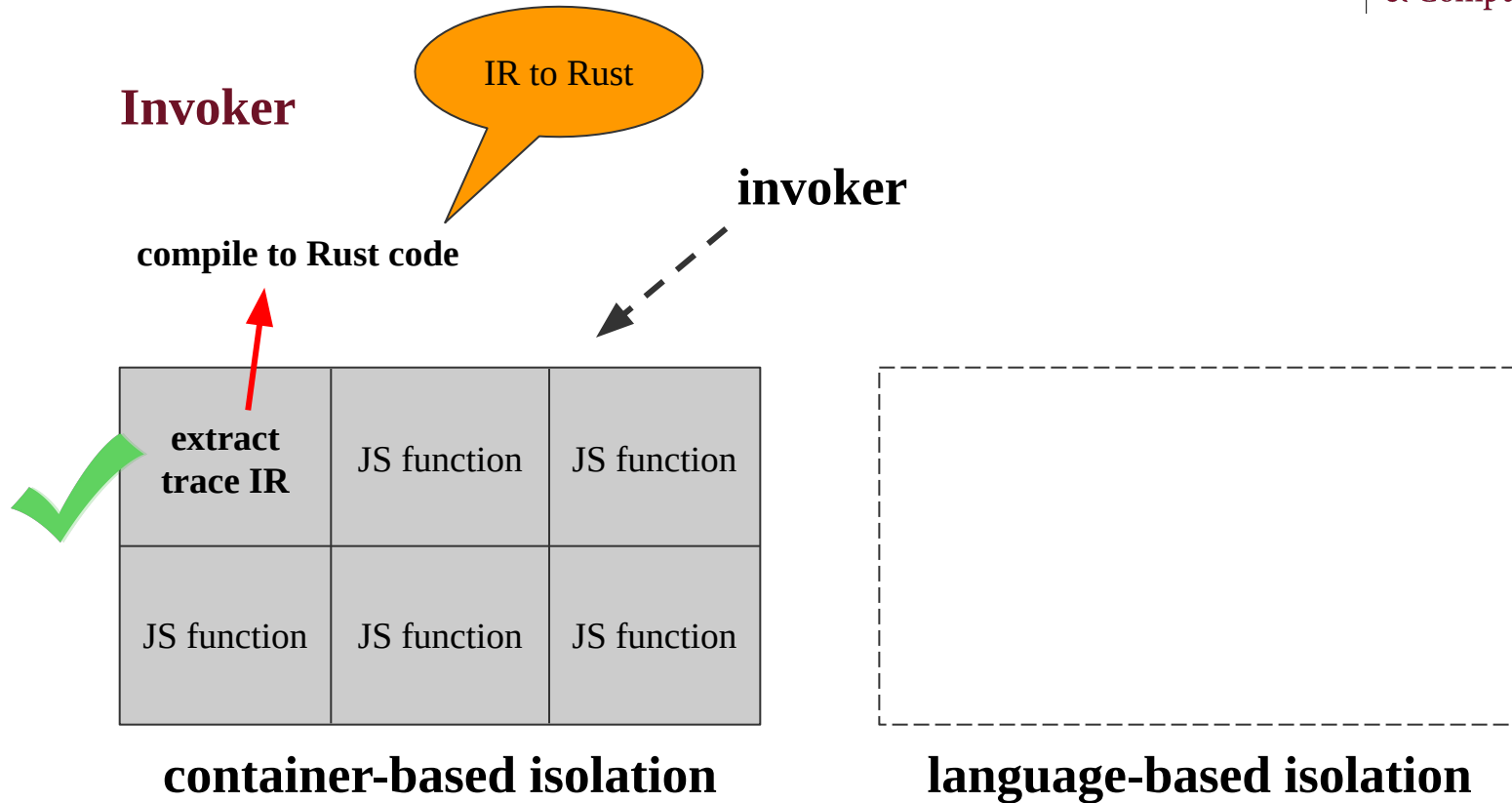


JavaScript to IR

**container-based isolation**

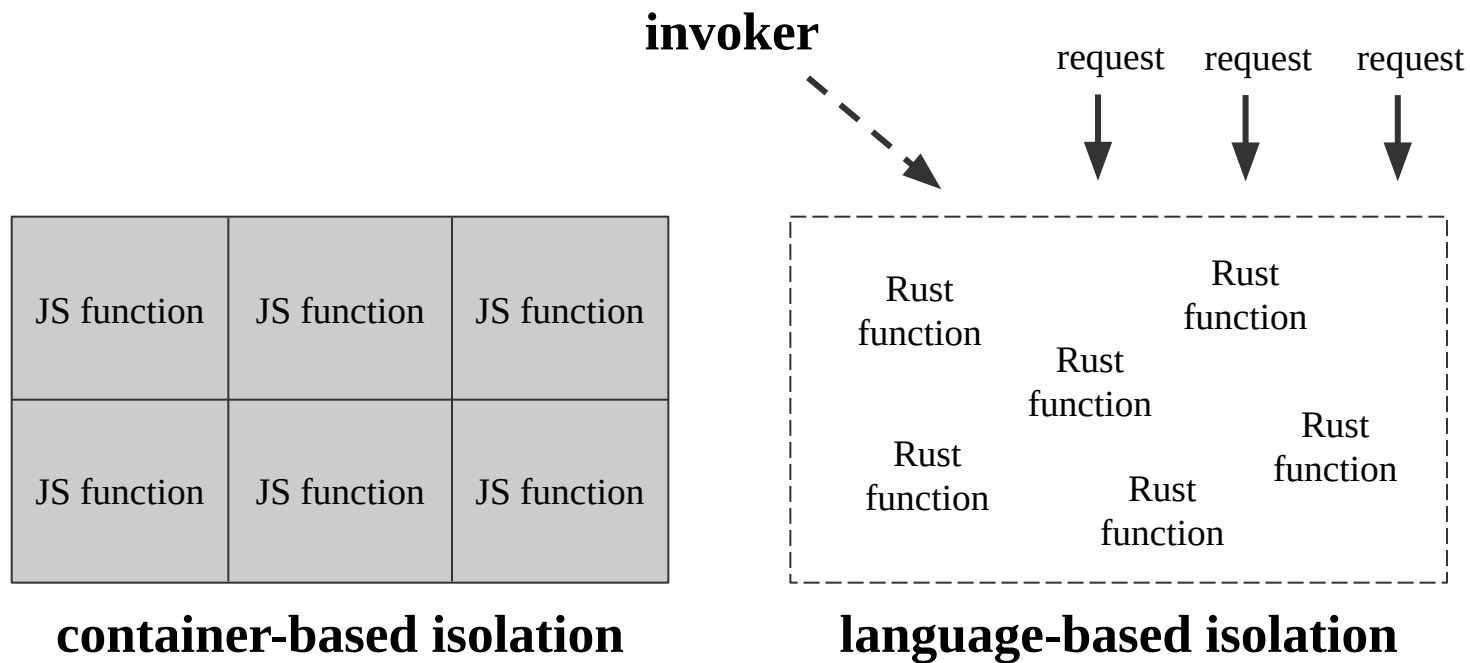


**language-based isolation**

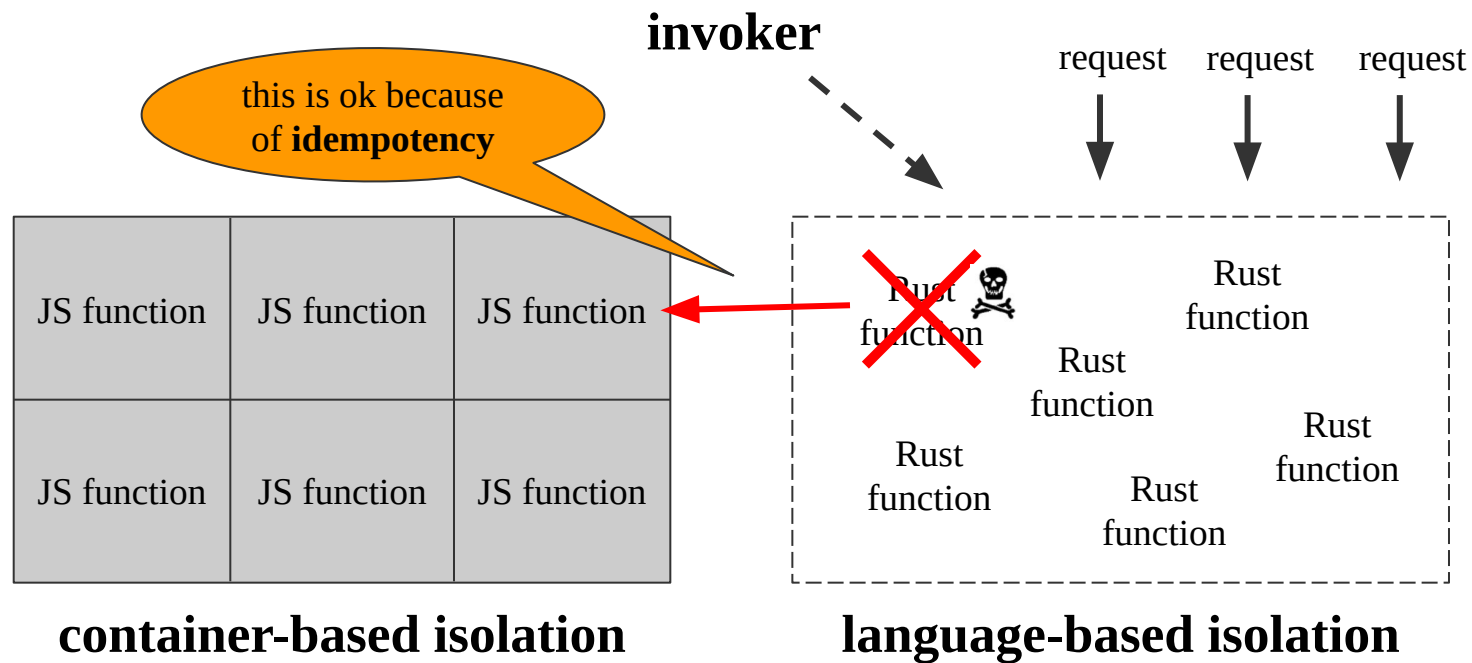




## Invoker



## Invoker

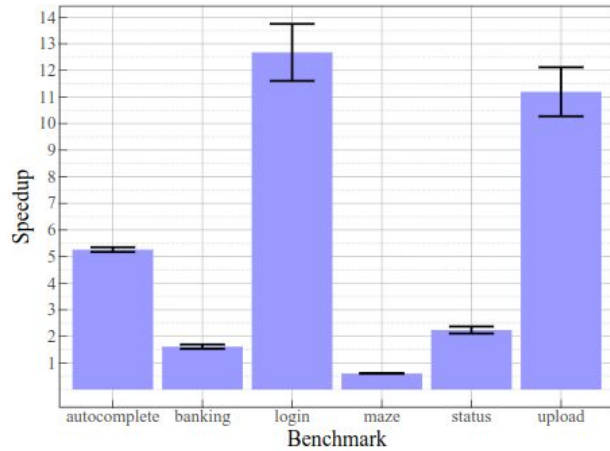


## Containerless

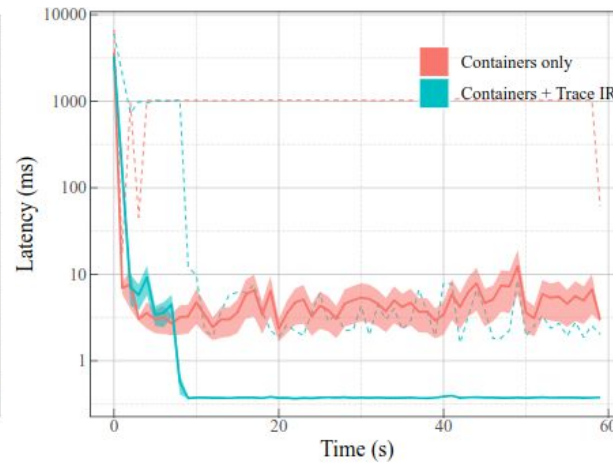
Three general components:

1. JavaScript to IR  $\longrightarrow$  Eliminates functions, etc.
2. IR to Rust  $\longrightarrow$  Dynamic type, arena allocation, etc.
3. invoker  $\longrightarrow$  Manages language-based isolation

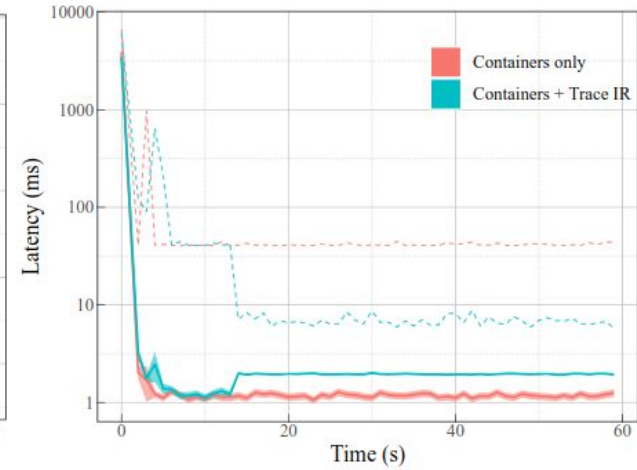
## Latency



all benchmarks

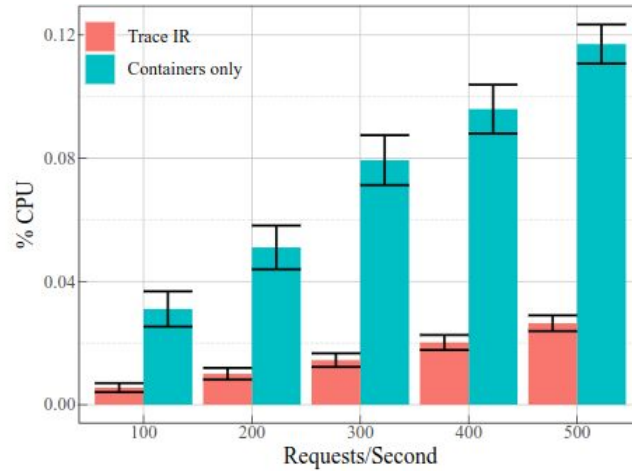


authorize benchmark

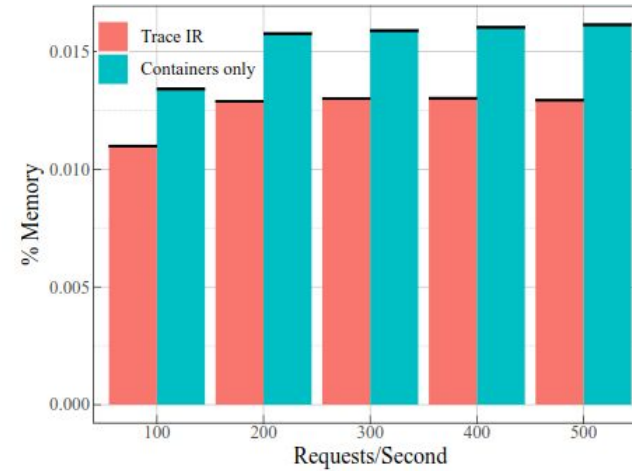


maze benchmark

## Utilization



CPU utilization



memory utilization

**Thanks!**