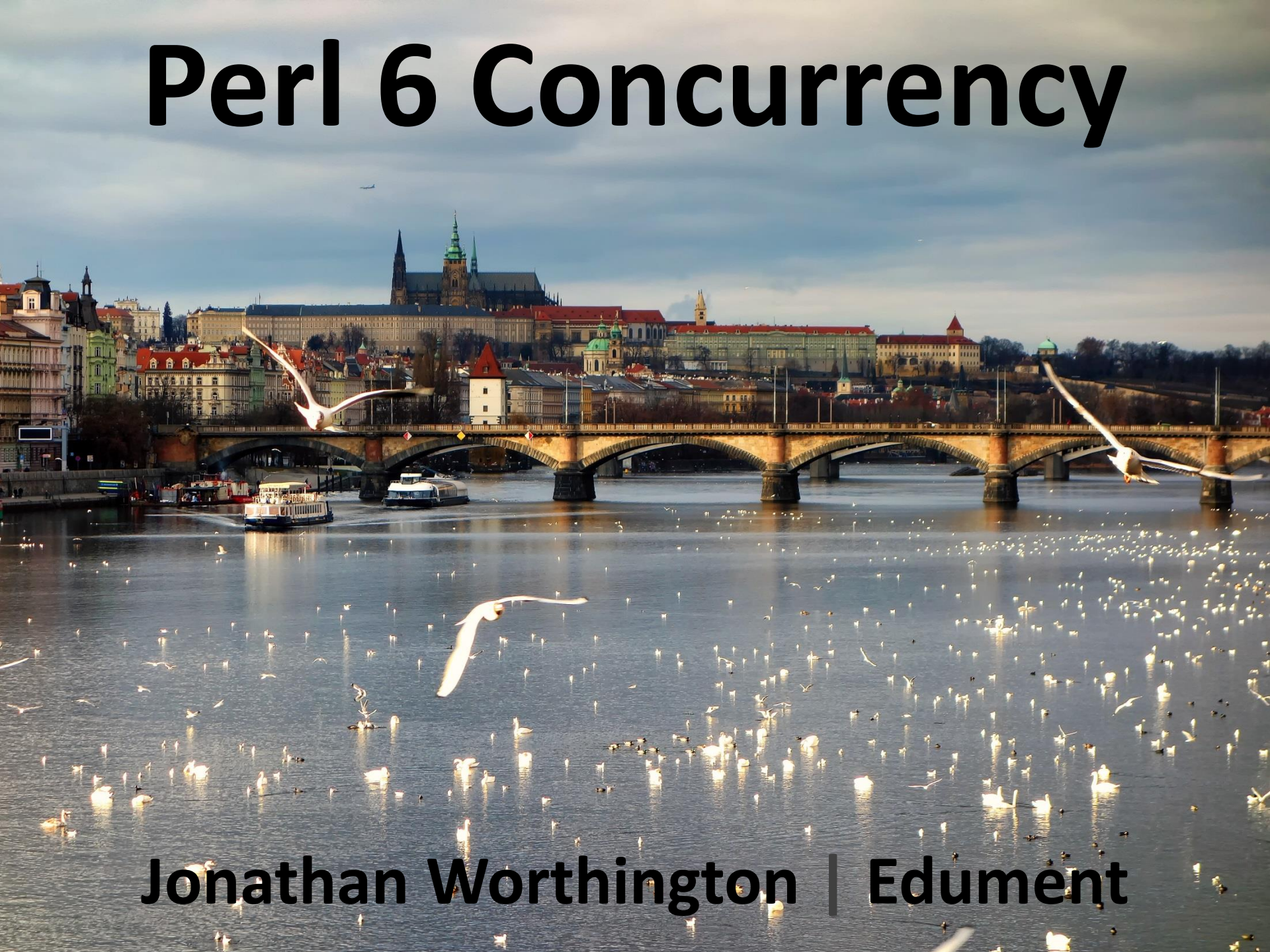
Perl 6 Concurrency



Jonathan Worthington | Edument

Hi. I'm Jonathan.

Hi. I'm Jonathan.

I do Perl 6 stuff....

Perl 6 concurrency designer

MoarVM founder and architect

Rakudo compiler developer

Hi. I'm Jonathan.

I do Perl 6 stuff....

Perl 6 concurrency designer

MoarVM founder and architect

Rakudo compiler developer

And I lead the Edument Prague office...

Developer tooling and compiler consultancy

Founder of Cro and Comma

Today...

The essence of Perl 6 concurrency

Key concepts and mechanisms

The essence of Perl 6 concurrency

Key concepts and mechanisms

The application of Perl 6 concurrency

A case-study from a production application

The essence of Perl 6 concurrency

Key concepts and mechanisms

The application of Perl 6 concurrency

A case-study from a production application

The future of Perl 6 concurrency

Where are we heading?

The

essence

of Perl 6 Concurrency

The essence of Perl 6 concurrency and parallelism reflects the essence of the Perl language family…

A Perlish language is

multi-paradigm

Because when we have a range of problem-solving tools, we can choose the most appropriate one for the problem at hand

Concurrency

Trying to get the right result when we have multiple, possibly competing, tasks with overlapping start/end times

We don't choose concurrency.

Concurrency chooses us.

Parallelism

Exploit multi-core hardware to do the same task, and deliver equivalent results, but in less wallclock time.

Concurrency is part of the

problem domain.

Parallelism is part of the

solution domain.

With concurrency, correctness

is domain specific.

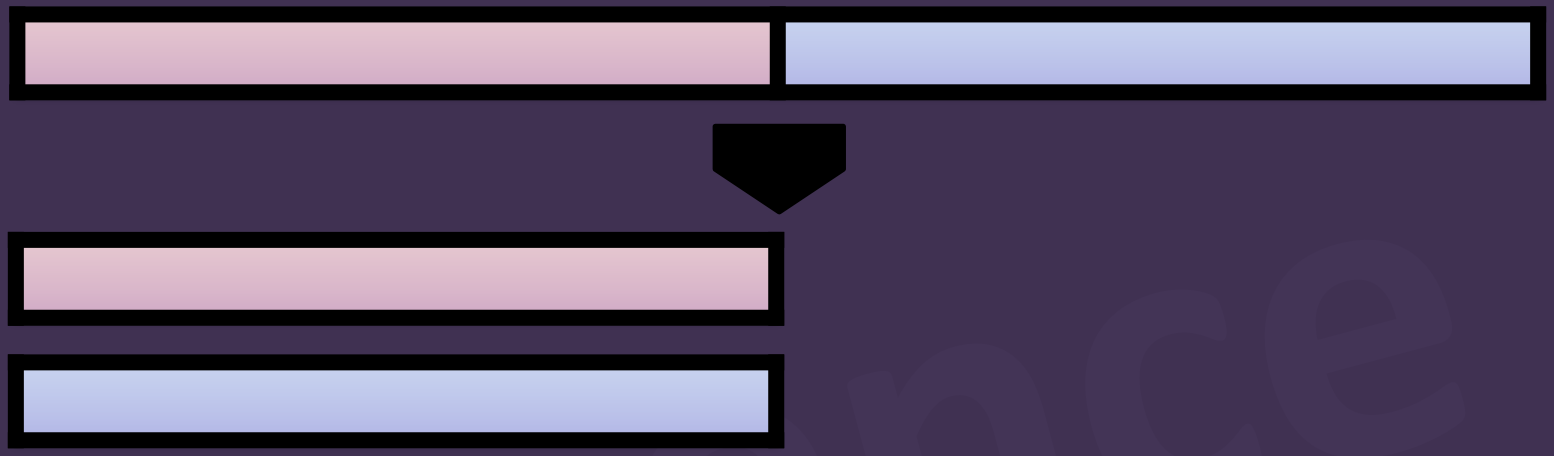
With parallelism, correctness

is just equivalence.

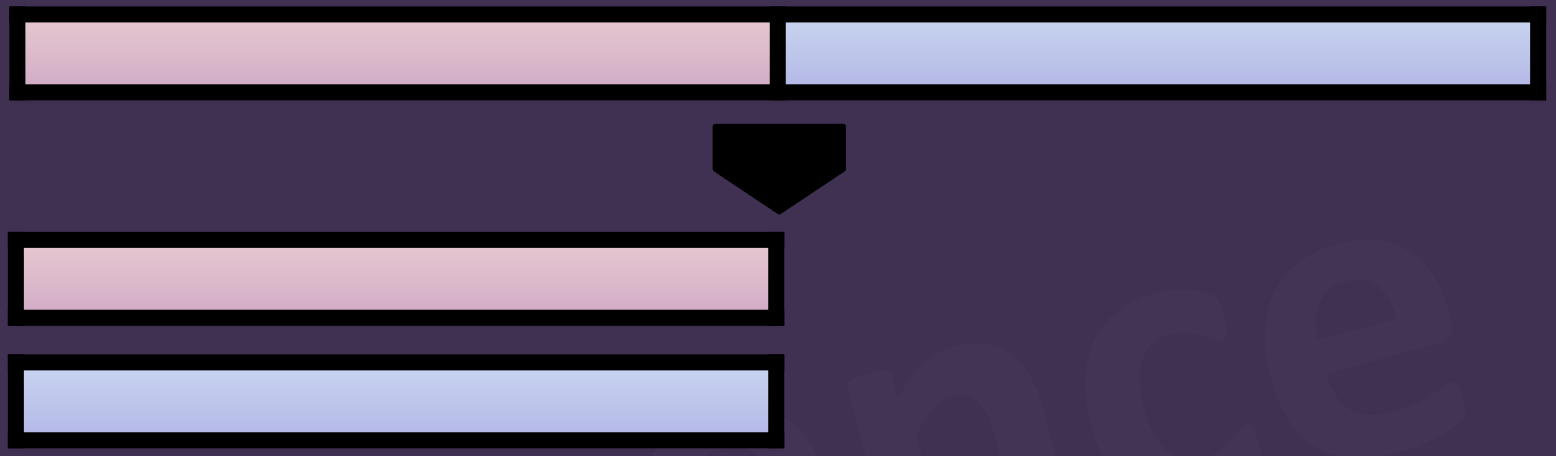
Concurrency and parallelism are best addressed by different tools.

In fact, there's different kinds of parallelism...

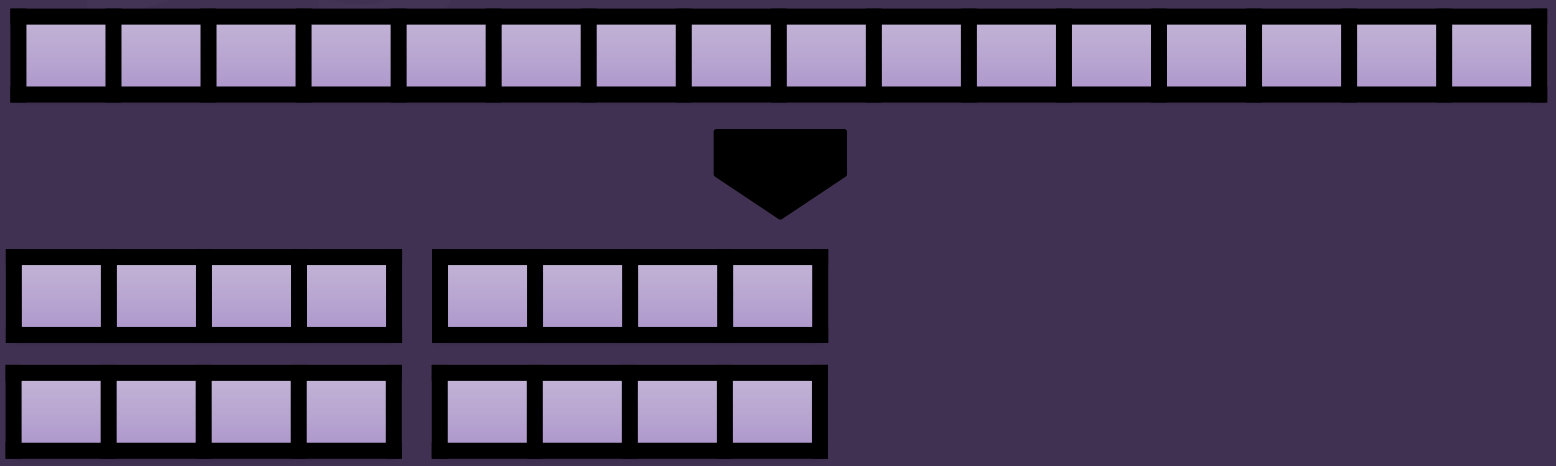
Task parallel



Task parallel

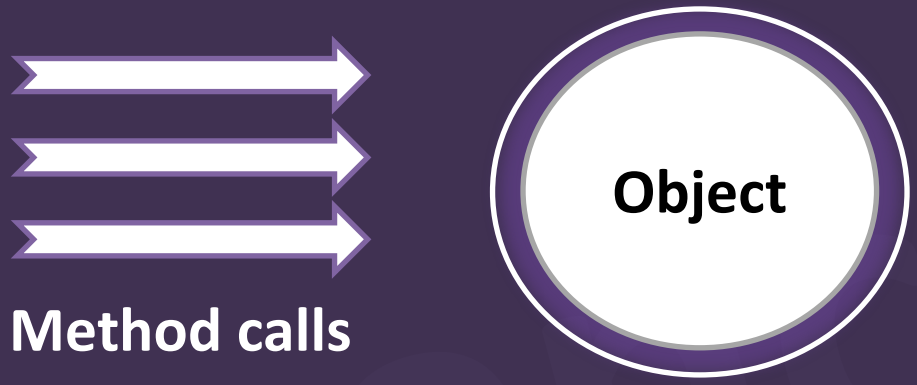


Data parallel

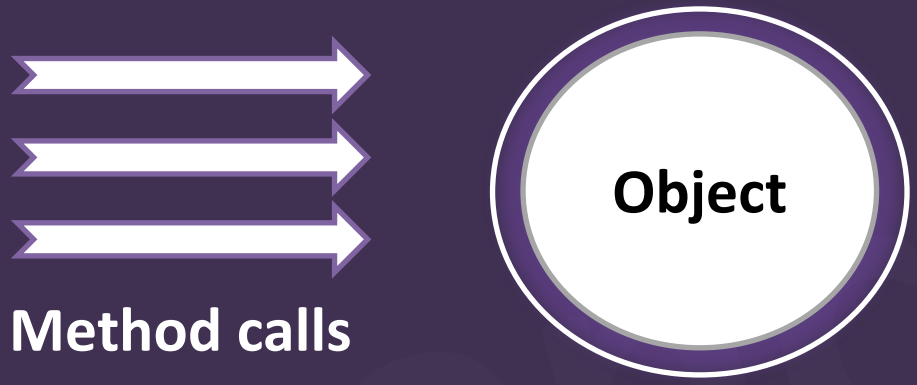


And different approaches to concurrency…

Concurrent objects



Concurrent objects



Event processing

Perl 6 provides for all of these, and more

A Perlish language makes the

easy things easy

Load and parse two files in parallel

```
my ($input-config, $app-config) = await
    start {
        load-yaml slurp $input-file
    },
    start {
        from-json $_ with slurp $*HOME.add('.fooconf')
    }
```

Parallel search for 100 palindromic primes

```
.say for (1..*)
    .hyper(batch => 512, degree => 6)
    .grep(-> $n { $n.is-prime && $n eq $n.flip })
    .head(100);
```

Acquire a lock around all method calls

```
monitor Cache {
    has %!entries;
    method add(Str $key, Any $value --> Nil) {
        %!entries{$key} = $value;
    }
    method lookup(Str $key --> Any) {
        %!entries{$key} // fail "No entry '$key'"
    }
}
```

Re-run a script whenever it changes

```
react {
    my $current-proc;
    whenever $script.watch.unique(:as(*.path), :expires(1)) {
        .kill with $current-proc;
        $current-proc = Proc::Async.new($*EXECUTABLE, $script);
        my $done = $current-proc.start;
        whenever $done {
            $current-proc = Nil;
        }
    }
}
```

A Perlish language makes the

hard things possible

Perl 6 provides access too…

OS-level threads
Locks
Atomic operations

Don't use them!*

Don't use them!*

* Unless you're implementing new concurrency or parallelism paradigms and data structures in Perl 6 ☺

A Perlish language offers

whip-up-ability

When we "whip up a solution", we're typically taking existing components, which we then

wire together

And wiring things together depends on them having a

common interface

Promise

A single, asynchronously produced, value

Supply

A stream of asynchronously produced values

A `Supply` can be…

Network packets
WebSocket messages
File system notifications
Child process output
UI events
Timer ticks
Domain events

A Perlish language will

torture the language implementer for the sake of the language user

M : N

M : N

High-level tasks

OS-level threads

Many ~Core-count

M : N

High-level tasks

OS-level threads

`await`

Suspend the current high-level task until the thing it needs is available

async?

No async!

No need to refactor all of the callers in order to use `await`!
Just save the whole stack.

A Perlish language helps us to

do the right thing

What does the `supply/whenever` syntax give us?

Thanks to using it, this code will work robustly…

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

Unsubscription, however things end

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

Unsubscription, however things end

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If the data source completes…

Unsubscription, however things end

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If the data source completes, cancel the timeout

Unsubscription, however things end

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If we hit the timeout…

Unsubscription, however things end

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If we hit the timeout, close the data source

Automatic exception propagation

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

Automatic exception propagation

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If the data source crashes…

Automatic exception propagation

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If the data source crashes, cancel the timeout, convey the exception

Automatic cleanup upon downstream close

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

Automatic cleanup upon downstream close

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If our consumer unsubscribes…

Automatic cleanup upon downstream close

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

If our consumer unsubscribes, close the data source and cancel the timeout

Automatic concurrency control

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

Automatic concurrency control

```
sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        whenever $source {
            emit $_;
            LAST done;
        }
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
```

We'll only ever be in one whenever block at a time

Even if we remembered all of these, it'd be a huge amount of boilerplate

Instead, we just Do The Right Thing

The

application

of Perl 6 Concurrency

eA*sii*

A tool to assist insurance or reinsurance undertakings with calculation of the European regulatory standard formula (Solvency II, Pillar I) and associated reporting to the supervisory authority via XBRL (Solvency II, Pillar III).

EasiiLang

A pure, functional, non-Turing Complete language

The entire calculation forms a DAG, so can see the path from input to result

Syntax inspired by Perl 6

EasiiLang was easy...

Parsed by a Perl 6 grammar

Produces a tree, which is walked to evaluate the expression

Perl 6 is good at this stuff. But, that's not the focus for today...

Architecture

Backend

Exposes a HTTP API (using Cro)

Versioned Input Storage (uses a SQLite database)

Live Dataset (in-memory reactive calculation)

Architecture

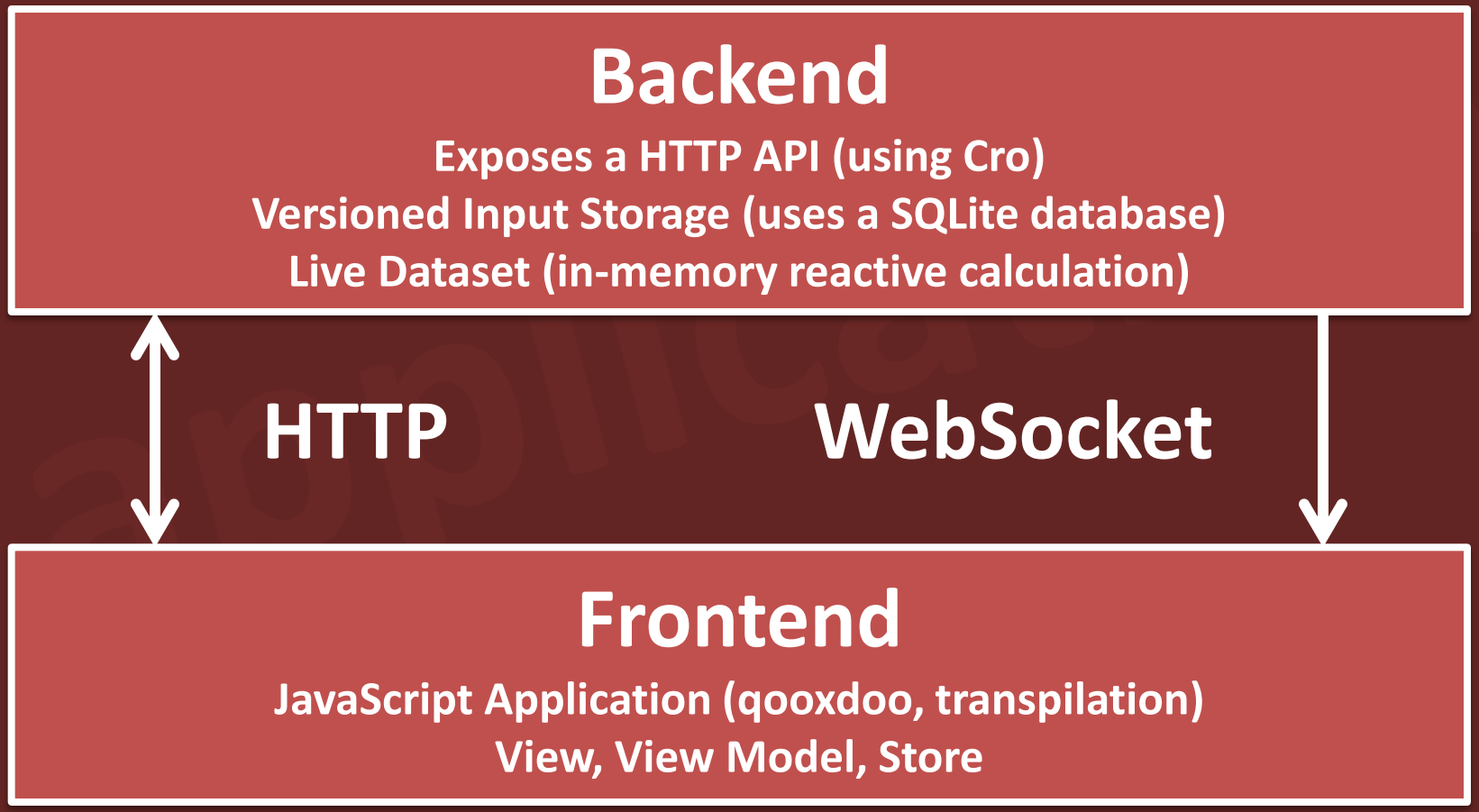
Backend

Exposes a HTTP API (using Cro)
Versioned Input Storage (uses a SQLite database)
Live Dataset (in-memory reactive calculation)

Frontend

JavaScript Application (qooxdoo, transpilation)
View, View Model, Store

Architecture



Cro

Libraries for building distributed systems; currently mostly used for building HTTP applications

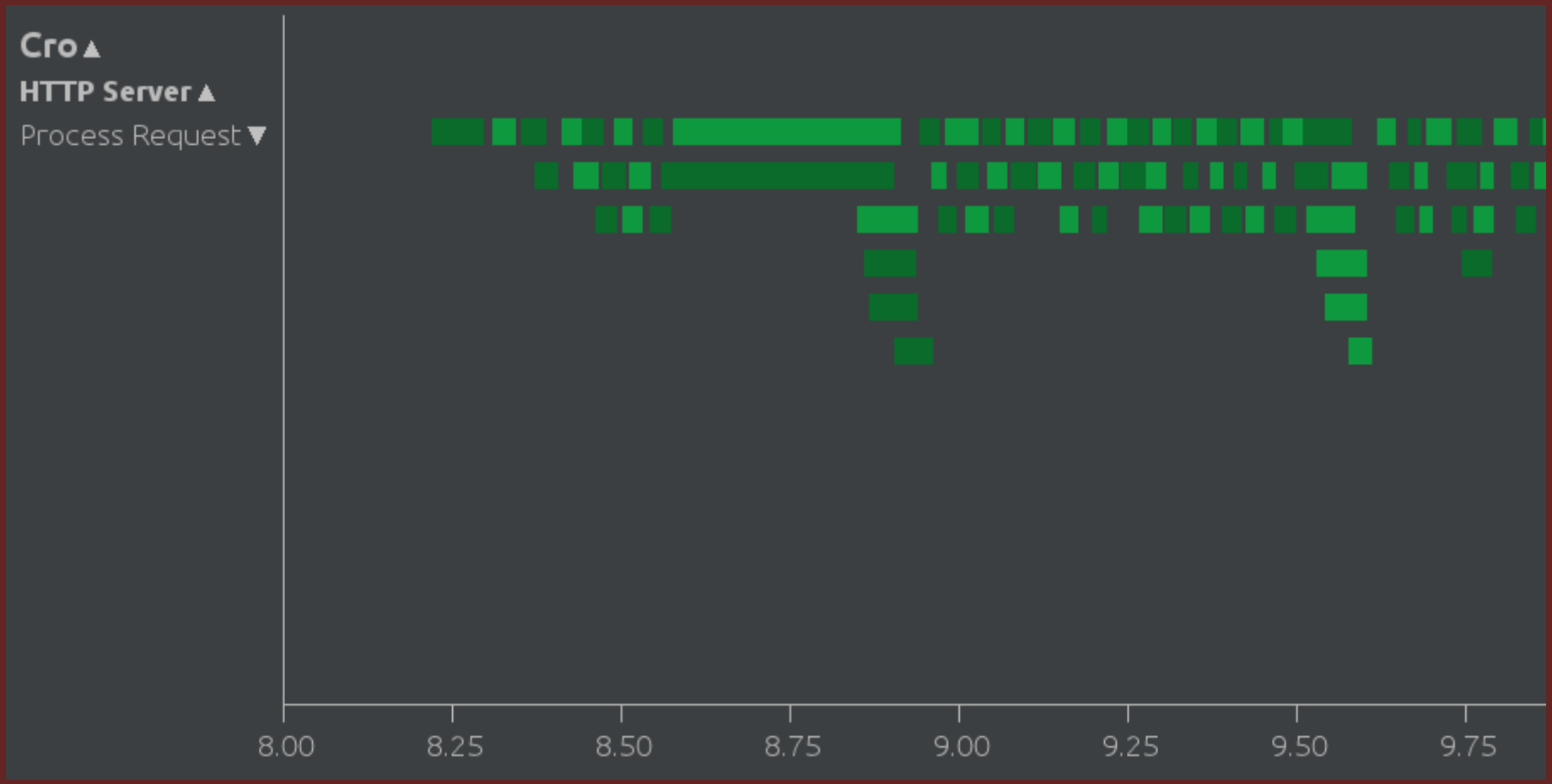
Request and response processing pipeline is a set of steps connected using `Supply` ➔ asynchronous

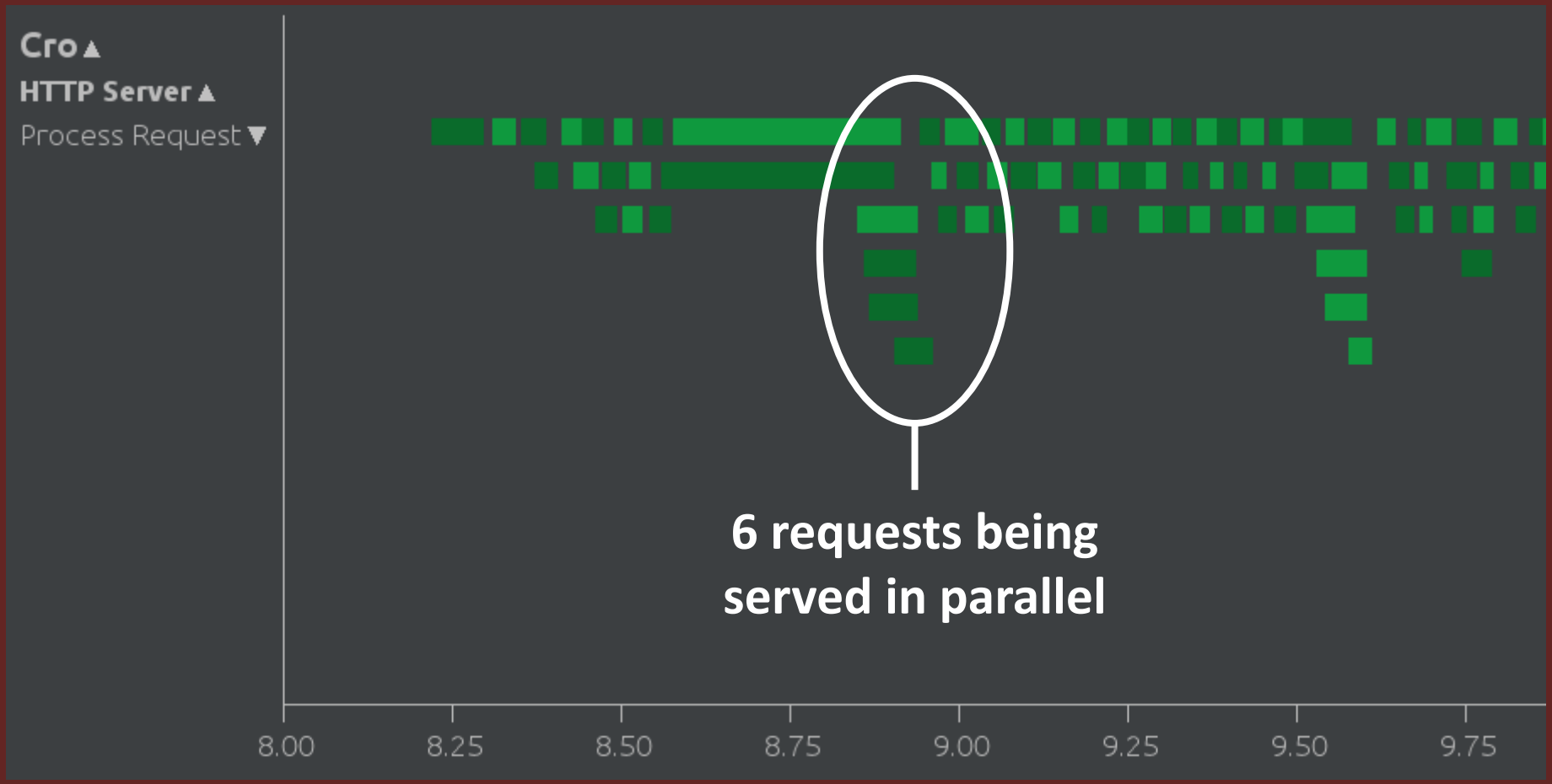
Cro has...

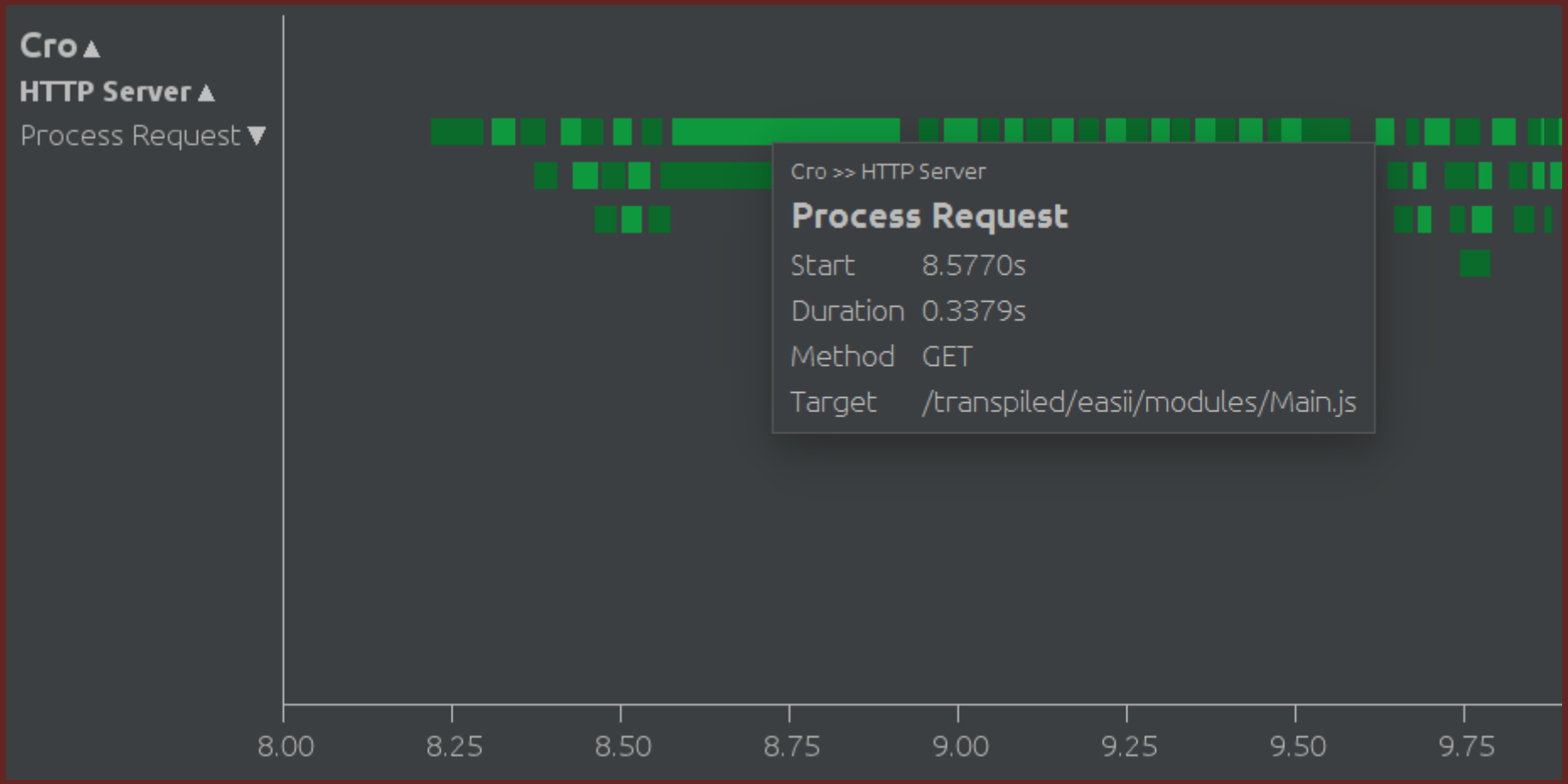
WebSocket support

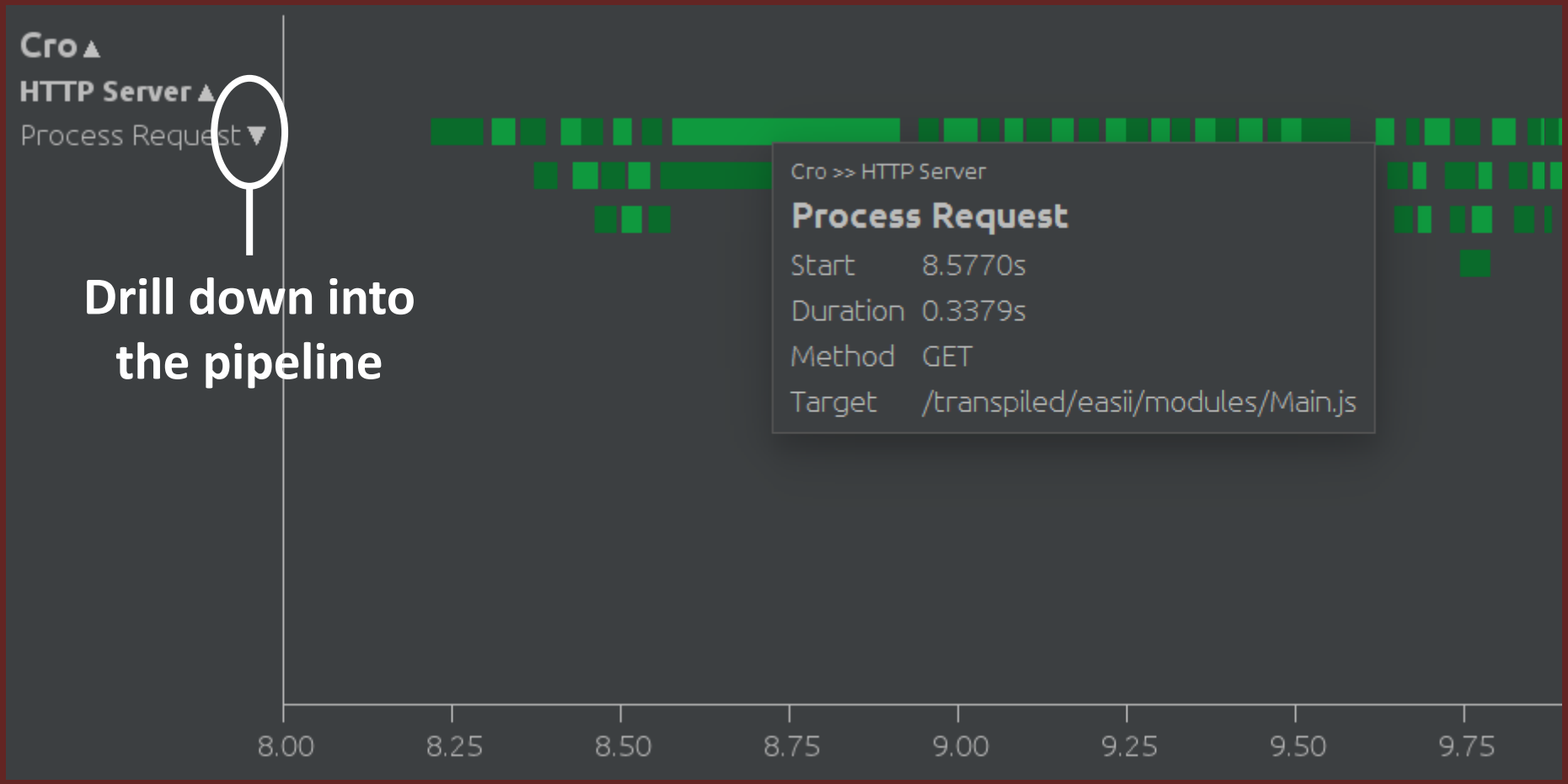
Reactive middleware

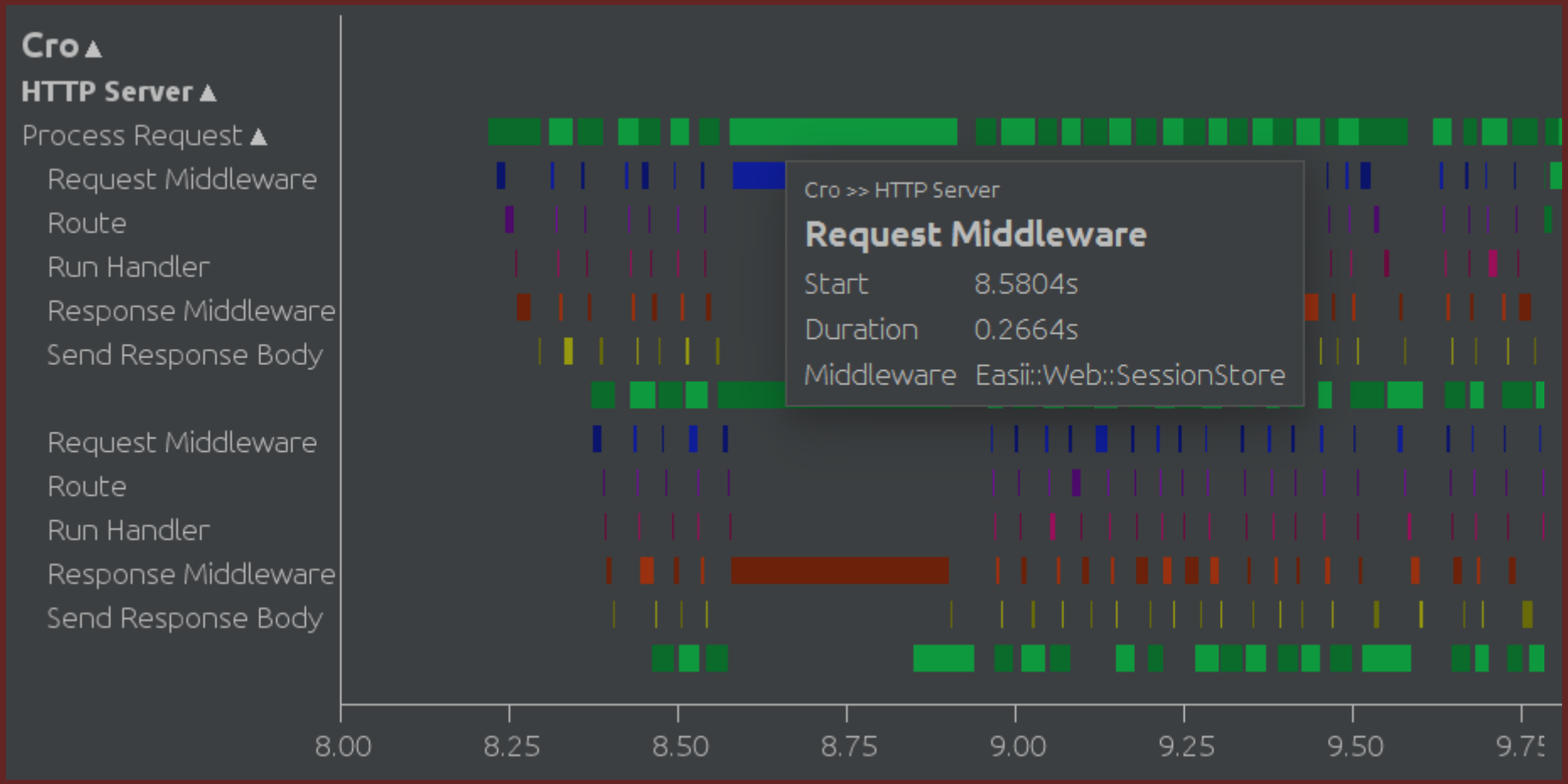
`Log::Timeline` integration, to allow tools to trace the request pipeline

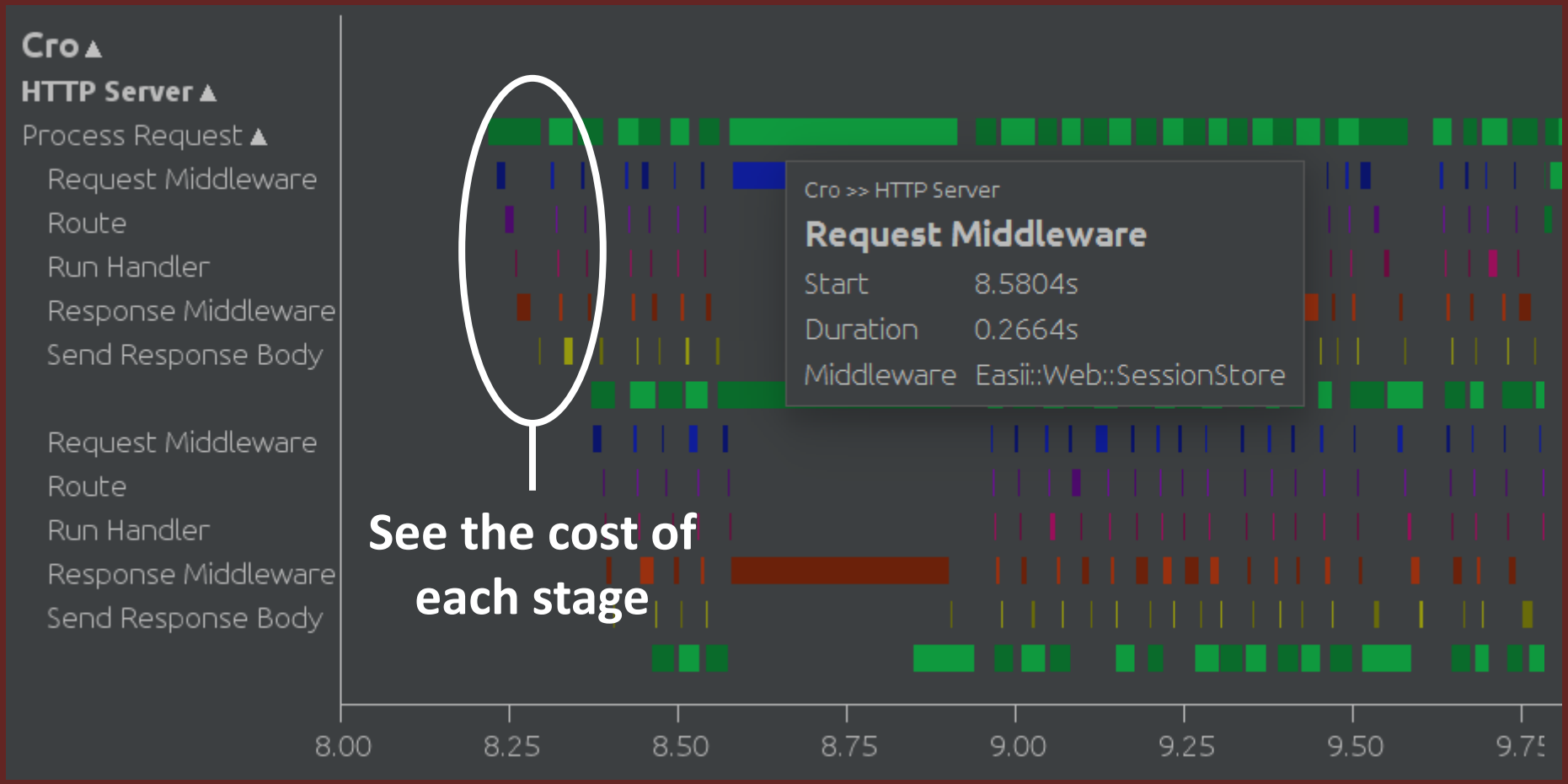

Cro▲
HTTP Server▲
Process Request▼
8.00
8.25
8.50
8.75
9.00
9.25
9.50
9.75


Cro▲
HTTP Server▲
Process Request▼
6 requests being
served in parallel
8.00
8.25
8.50
8.75
9.00
9.25
9.50
9.75


Cro ▲
HTTP Server ▲
Process Request ▼
Cro >> HTTP Server
Process Request
Start 8.5770s
Duration 0.3379s
Method GET
Target /transpiled/easii/modules/Main.js
8.00
8.25
8.50
8.75
9.00
9.25
9.50
9.75


Cro ▲
HTTP Server ▲
Process Request ▼
Drill down into the pipeline
Cro >> HTTP Server
Process Request
Start 8.5770s
Duration 0.3379s
Method GET
Target /transpiled/easii/modules/Main.js
8.00
8.25
8.50
8.75
9.00
9.25
9.50
9.75


Cro ▲
HTTP Server ▲
Process Request ▲
Request Middleware
Route
Run Handler
Response Middleware
Send Response Body
Request Middleware
Route
Run Handler
Response Middleware
Send Response Body
Cro >> HTTP Server
Request Middleware
Start 8.5804s
Duration 0.2664s
Middleware Easii::Web::SessionStore
8.00
8.25
8.50
8.75
9.00
9.25
9.50


Cro ▲
HTTP Server ▲
Process Request ▲
Request Middleware
Route
Run Handler
Response Middleware
Send Response Body
Request Middleware
Route
Run Handler
Response Middleware
Send Response Body
Cro >> HTTP Server
Request Middleware
Start 8.5804s
Duration 0.2664s
Middleware Easii::Web::SessionStore
See the cost of each stage
8.00
8.25
8.50
8.75
9.00
9.25
9.50

Log::Timeline

Can use it to do application-level logging also

Doing this helped us to understand the application behavior, and guided our use of parallelism

The model

Developed by mathematicians based on European regulations

Loaded at application startup

During model development, reloaded when the model is changed

The current model

350+ modules
A YAML file for each. Totals over 100,000 lines of YAML.

Nearly 4,000 formulas
25,000 lines of EasiiLang between the modules

64 Excel Documents...
Based on the legal requirement of the European supervisory authority

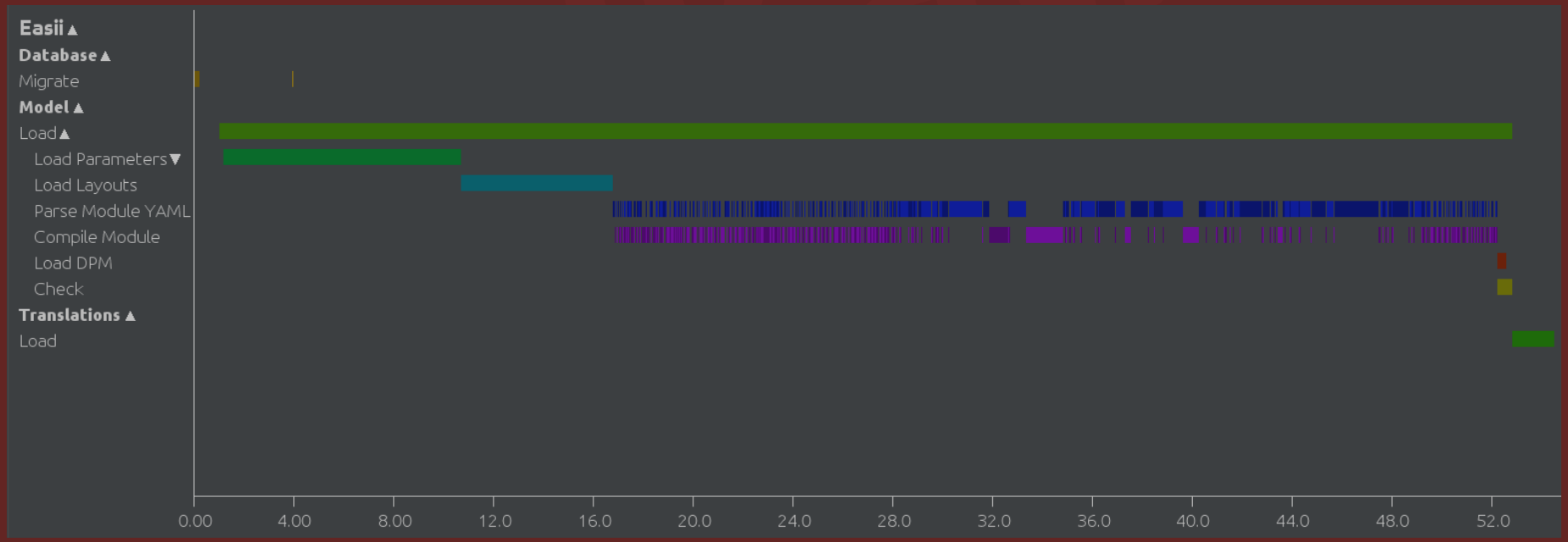
...cached as 7 MB of JSON
Since reading from Excel every time we load the model is too slow

5,500+ lines of CSV
Containing parameters, such as country-specific data

2000+ translation keys
And many more to come, written in `.po` files

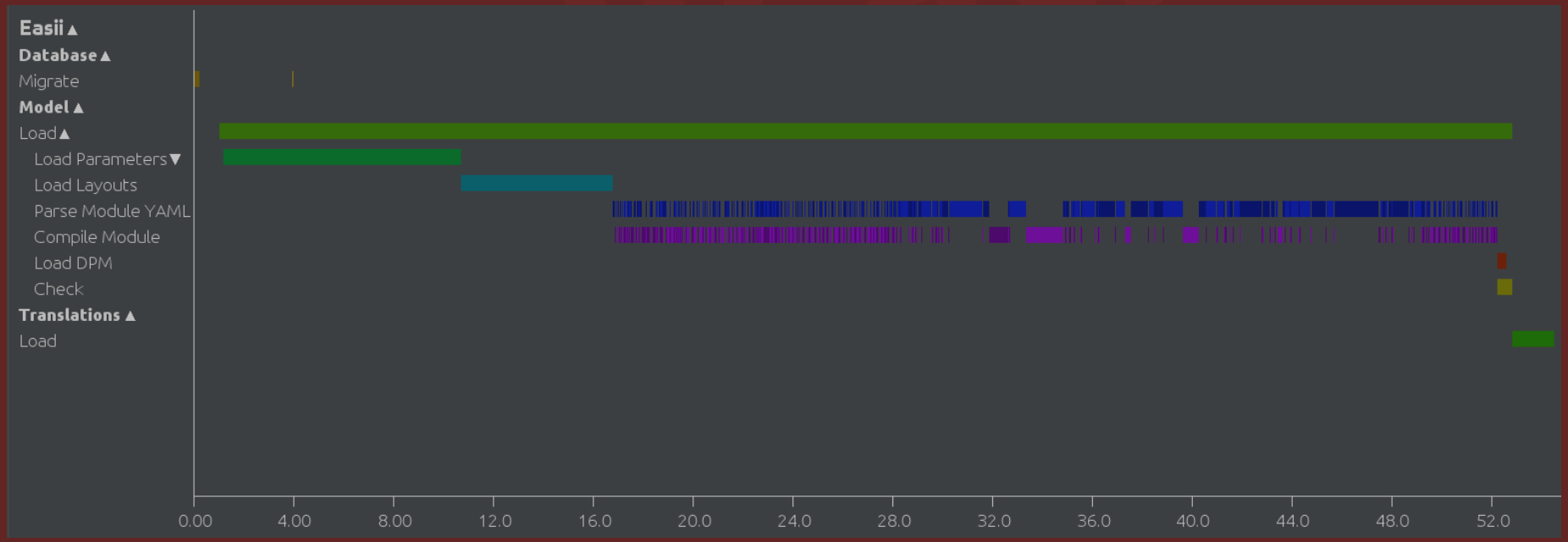
Model loading

As the model grew, model reloads became long enough to be annoying



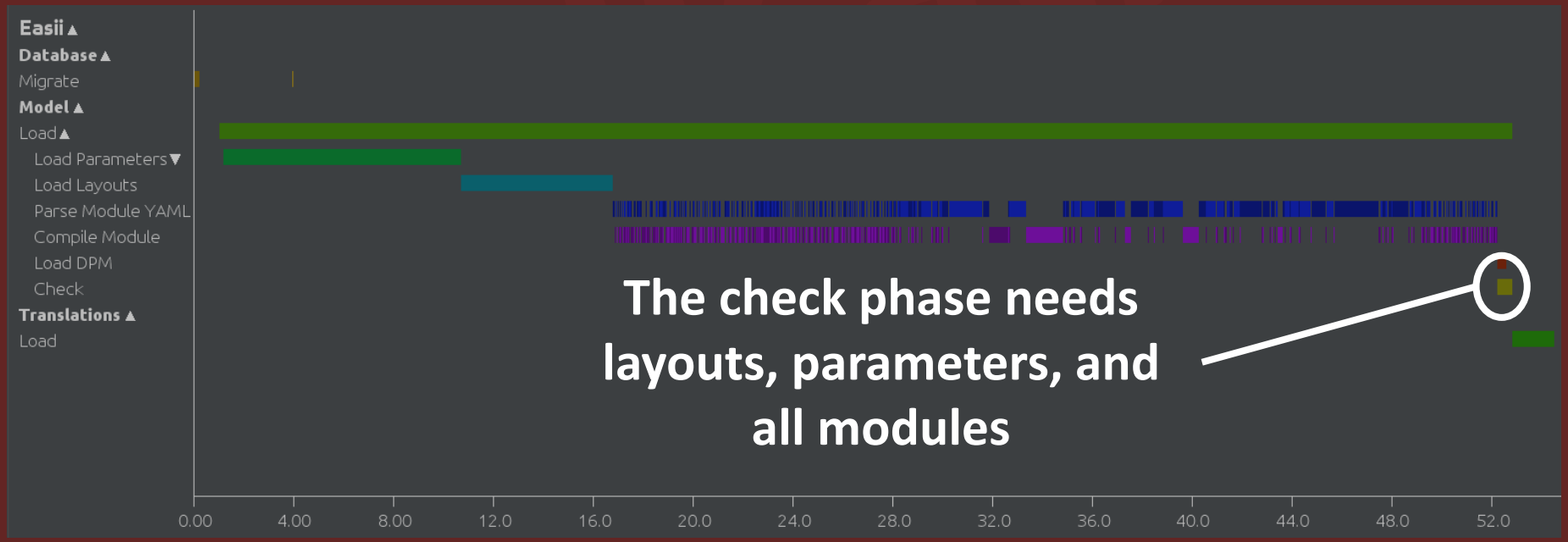
Can we parallelize?

First, need to consider the data dependencies of model loading



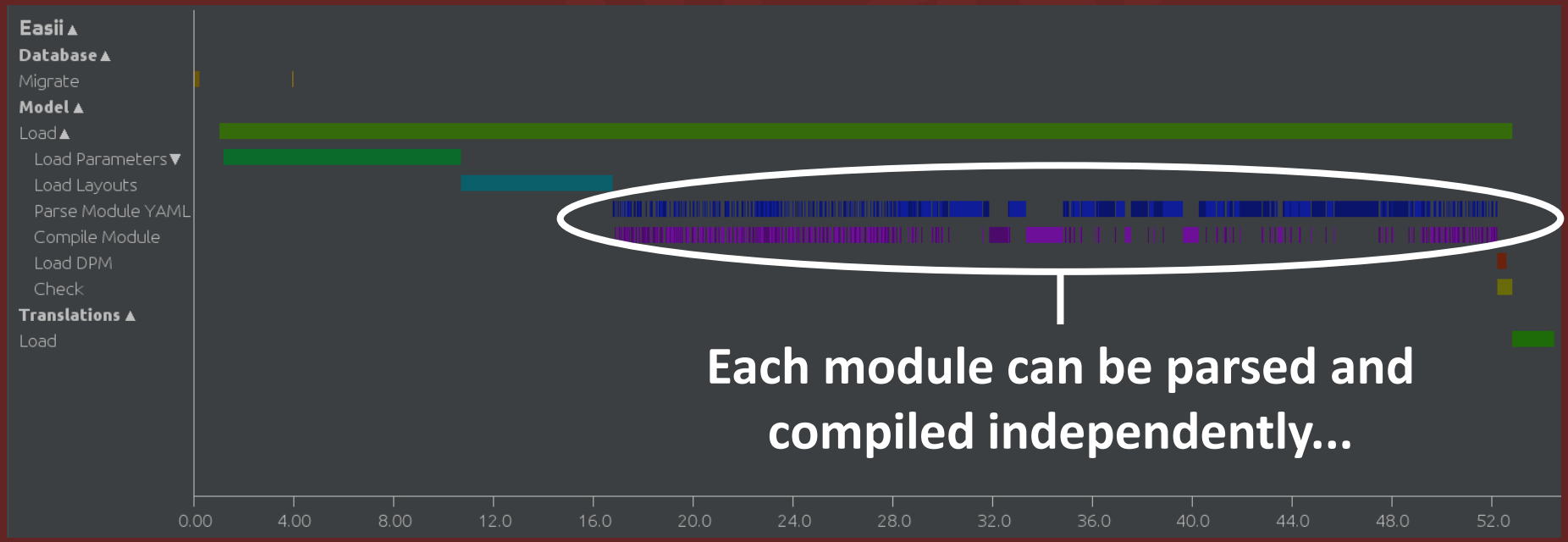
Can we parallelize?

First, need to consider the data dependencies of model loading



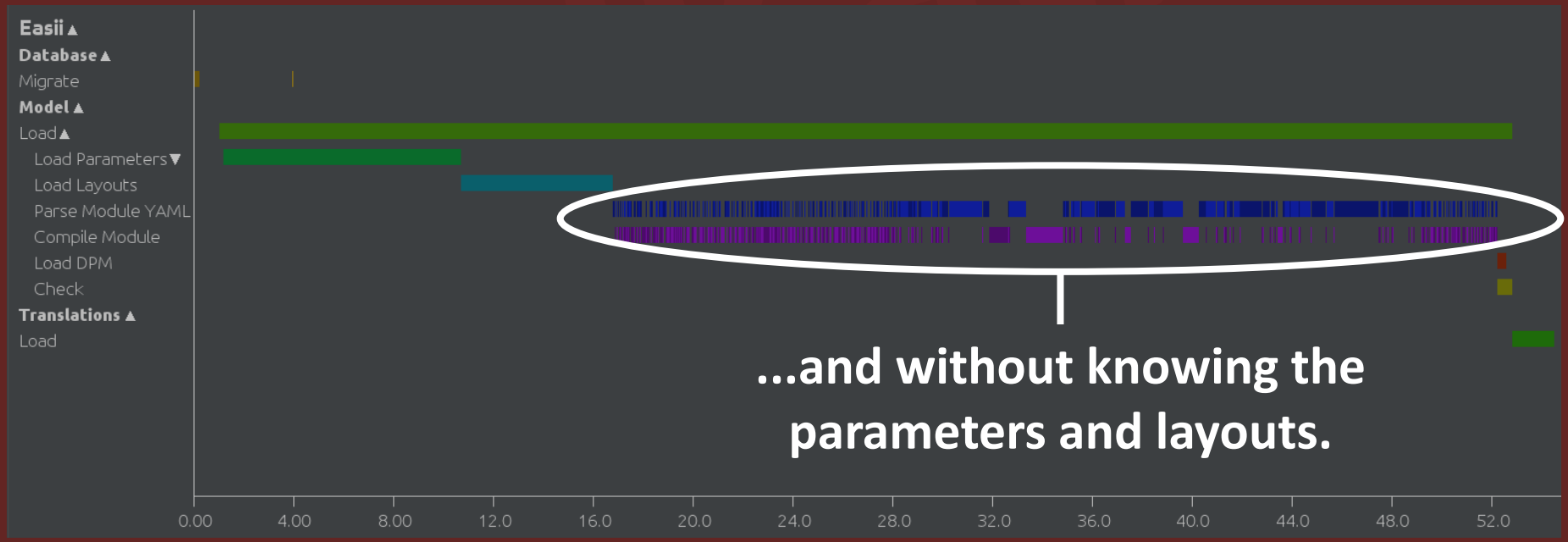
Can we parallelize?

First, need to consider the data dependencies of model loading



Can we parallelize?

First, need to consider the data dependencies of model loading



Data parallelism

When we apply the same operation to many data items

Parallelism comes from partitioning the data - into items or batches - and spreading them over worker threads

Module loading

```
my @modules = @files
    .grep(/ \.(yaml|yml) $/)
    .map(-> $file {
        my $yaml = Easii::Log::ParseModuleYAML.log: $task, :file(~$file), -> {
            self!load-yaml($file, $schema, $problems)
        }
        with $yaml {
            Easii::Log::CompileModule.log: $task, :file(~$file), -> {
                Easii::Model::Module.new(parsed => $yaml,
                        source => $file.basename)
            }
        }
    });
```

Module loading

(Log::Timeline use omitted for simplicity)

```
my @modules = @files
    .grep(/ \.(yaml|yml) $/)
    .map(-> $file {
        my $yaml = self!load-yaml($file, $schema, $problems) ;
        with $yaml {
            Easii::Model::Module.new(parsed => $yaml,
                    source => $file.basename)
        }
    });
```

Load in parallel

```
my @modules = @files
    .grep(/ \.(yaml|yml) $/)
    .race(batch => 1, degree => 6)
    .map(-> $file {
        my $yaml = self!load-yaml($file, $schema, $problems) ;
        with $yaml {
            Easii::Model::Module.new(parsed => $yaml,
                    source => $file.basename)
        }
    });
```

But wait...

```
my @modules = @files
    .grep(/ \.(yaml|yml) $/)
    .race(batch => 1, degree => 6)
    .map(-> $file {
        my $yaml = self!load-yaml($file, $schema, $problems) ;
        with $yaml {
            Easii::Model::Module.new(parsed => $yaml,
                    source => $file.basename)
        }
    });
```

This problem collector may be used concurrently

Not safe

```
my class Problems {
    has @.errors;
    method add-error($error--> Nil) {
        @!errors.push($error);
    }
}
```

Potential race on this

Make it a monitor

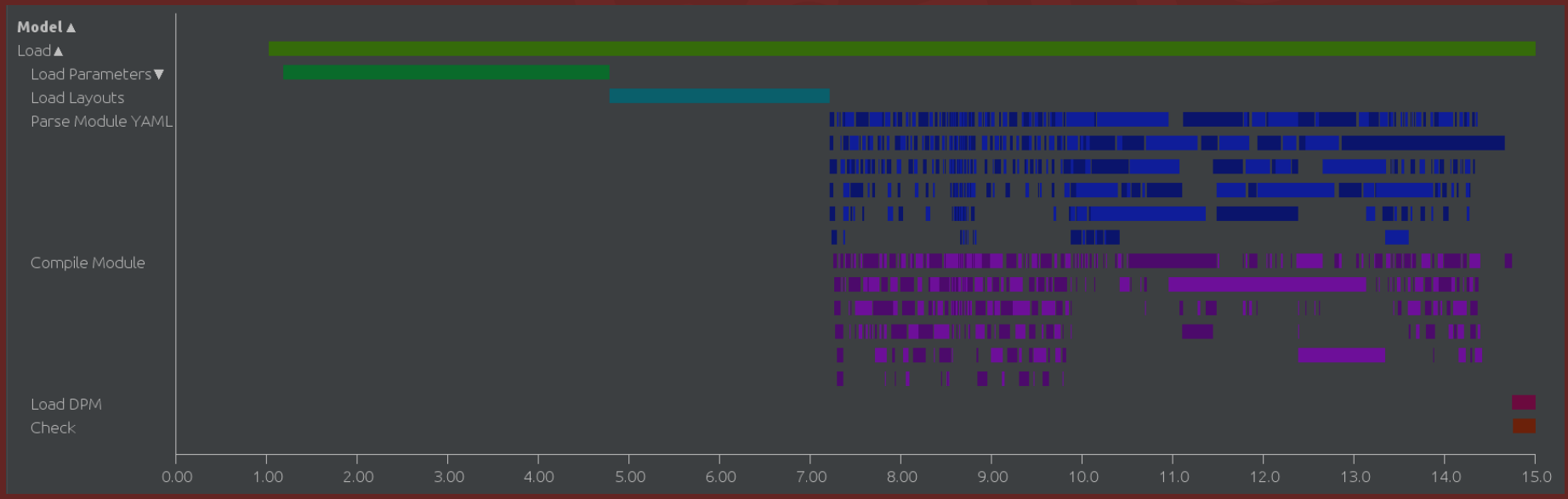
Acquires a lock automatically.

```
use OO::Monitors;

my monitor Problems {
    has @.errors;
    method add-error($error--> Nil) {
        @!errors.push($error);
    }
}
```

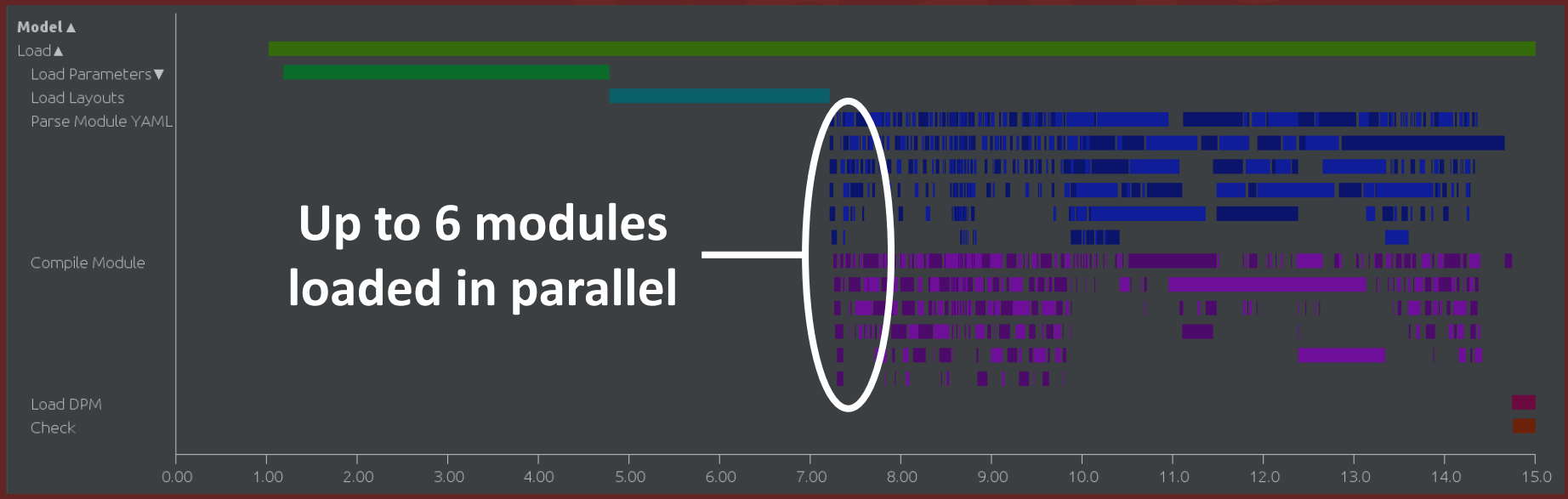
Huge improvement!

Takes 30% of the time it used to



Huge improvement!

Takes 30% of the time it used to



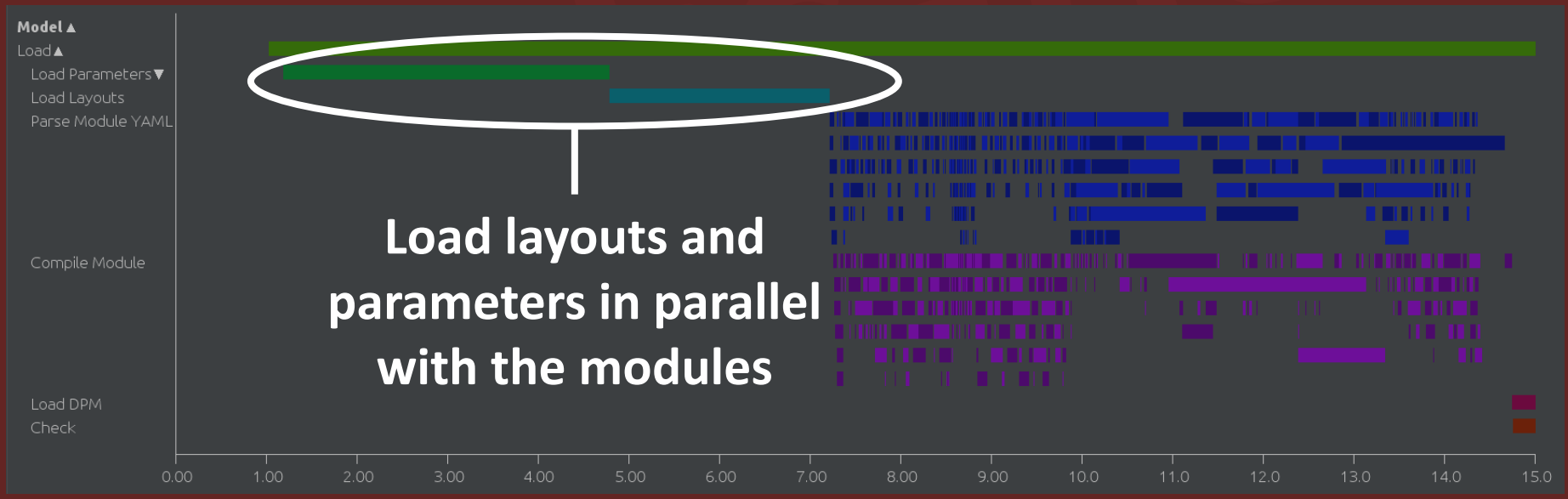
Task parallelism

Identify different, independent, tasks that we could do in parallel

Have different threads do them

Task parallelism?

There's an opportunity!



Load asynchronously

```
my $modules-load = start @files
    .grep(/ \.(yaml|yml) $/)
    .race(batch => 1, degree => 6)
    .map(-> $file {
        my $yaml = self!load-yaml($file, $schema, $problems) ;
        with $yaml {
            Easii::Model::Module.new(parsed => $yaml,
                    source => $file.basename)
        }
    })
    .eager;
```

Load asynchronously

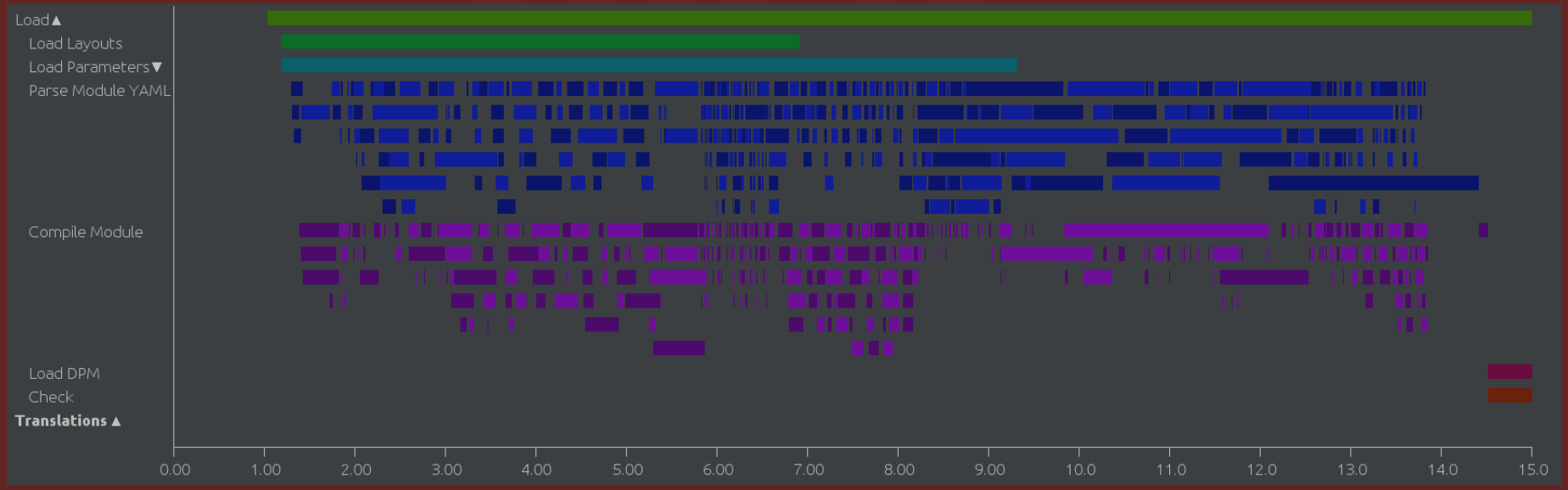
```
my $parameter-load = start self!load-parameters(
    $parameters-path, $problems);
my $layout-load = start self!load-layouts(
    $layouts-dir, $cache-dir, $problems);
```

Load asynchronously

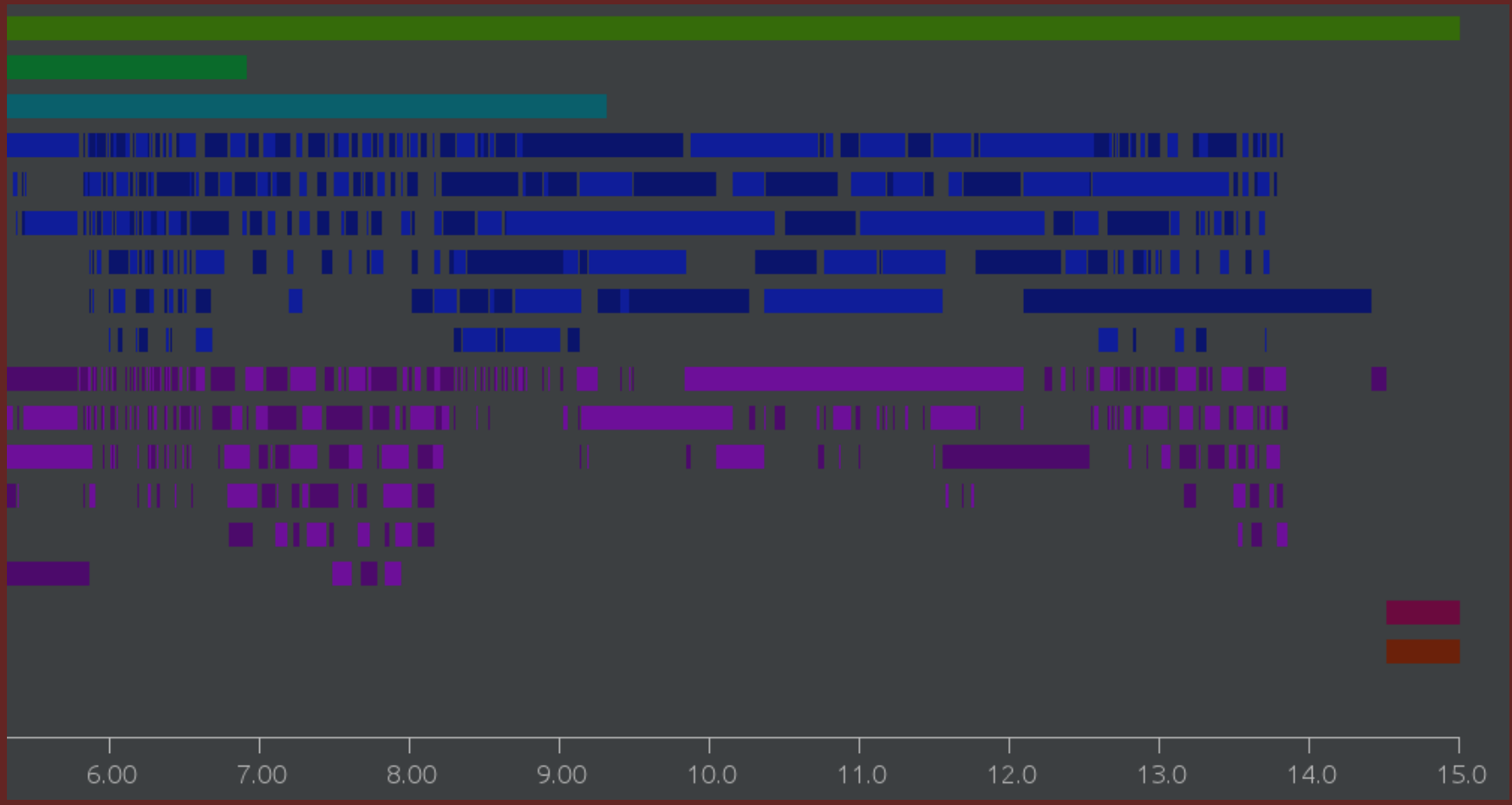
```
self.bless:
    modules => await($modules-load),
    parameters => await($parameter-load),
    layouts => await($layout-load),
    dpm => self!load-dpm($dpm-dir, $cache-dir),
    load-errors => $problems.errors
```

An improvement?

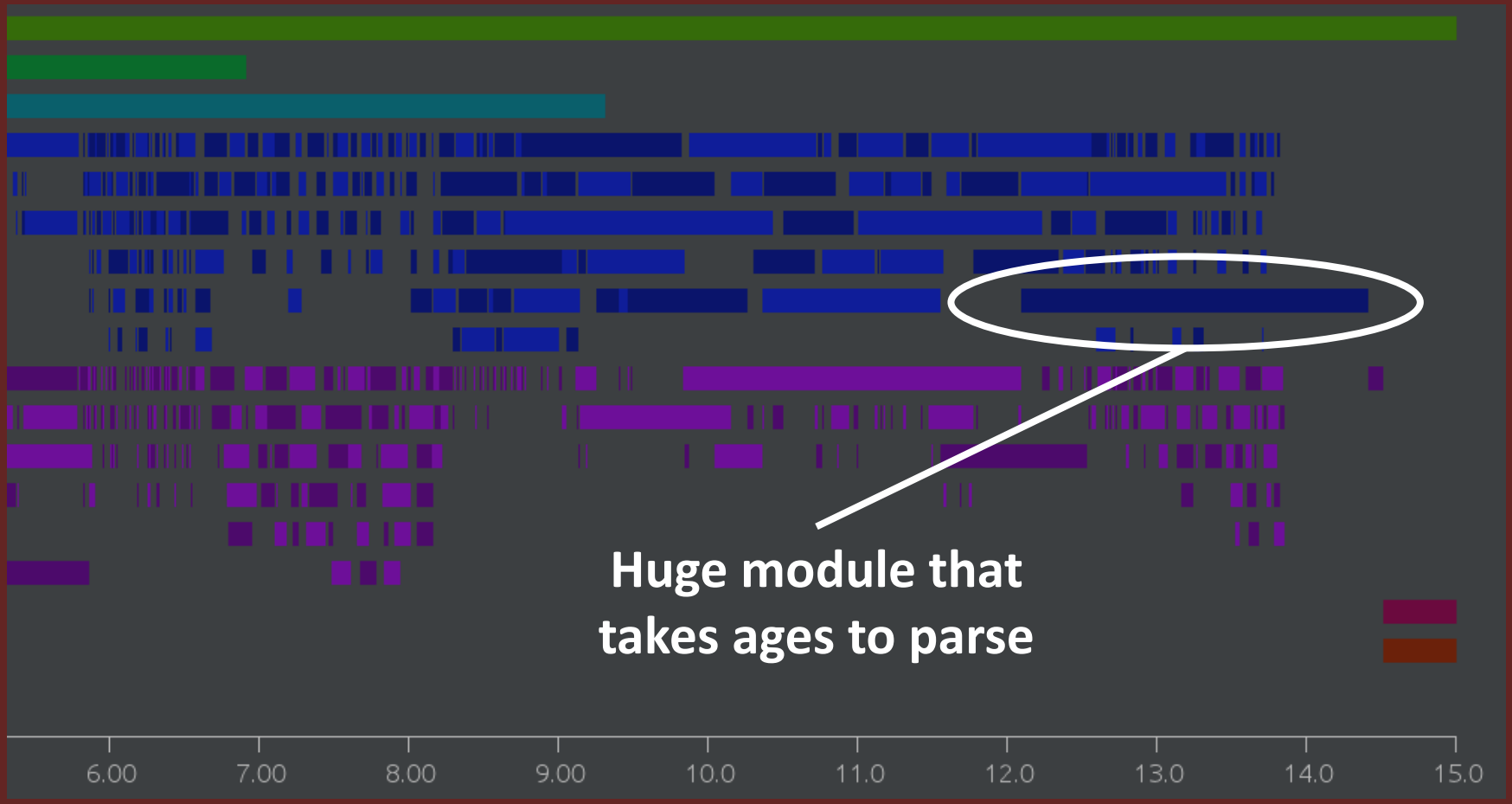
A little, though not that much more, due to resource contention



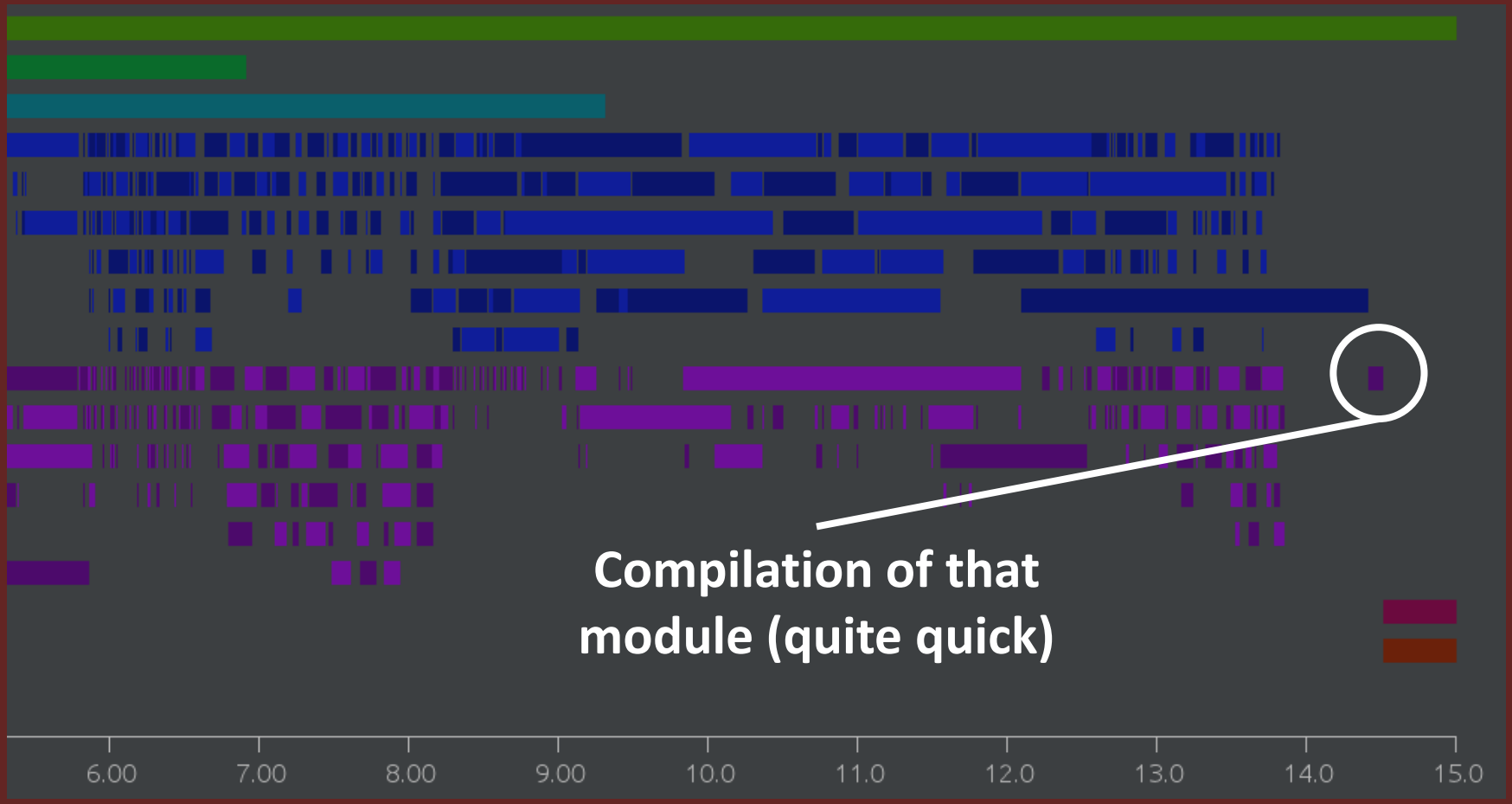
Looking closer...



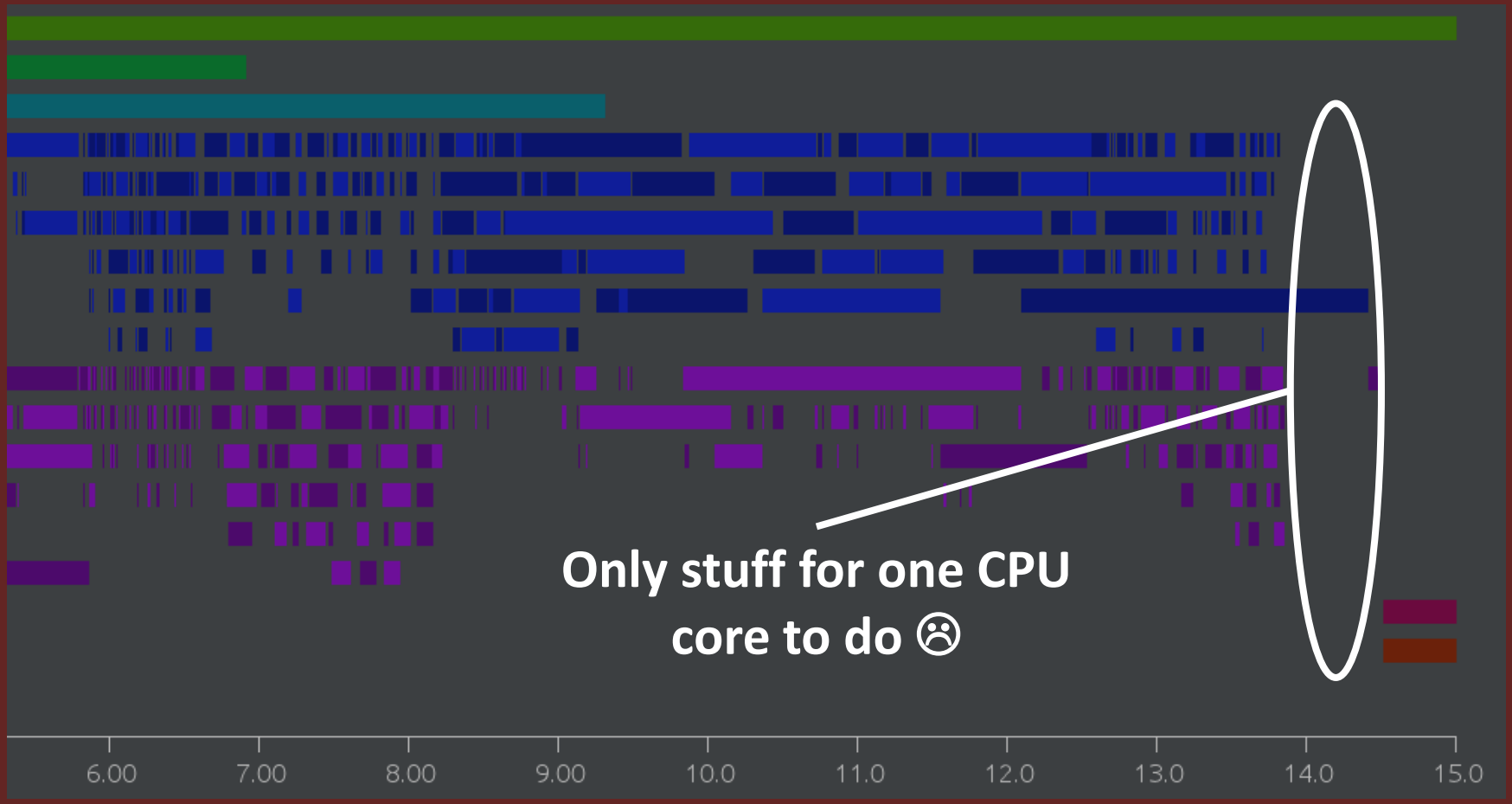
Looking closer...



Looking closer...



Looking closer...

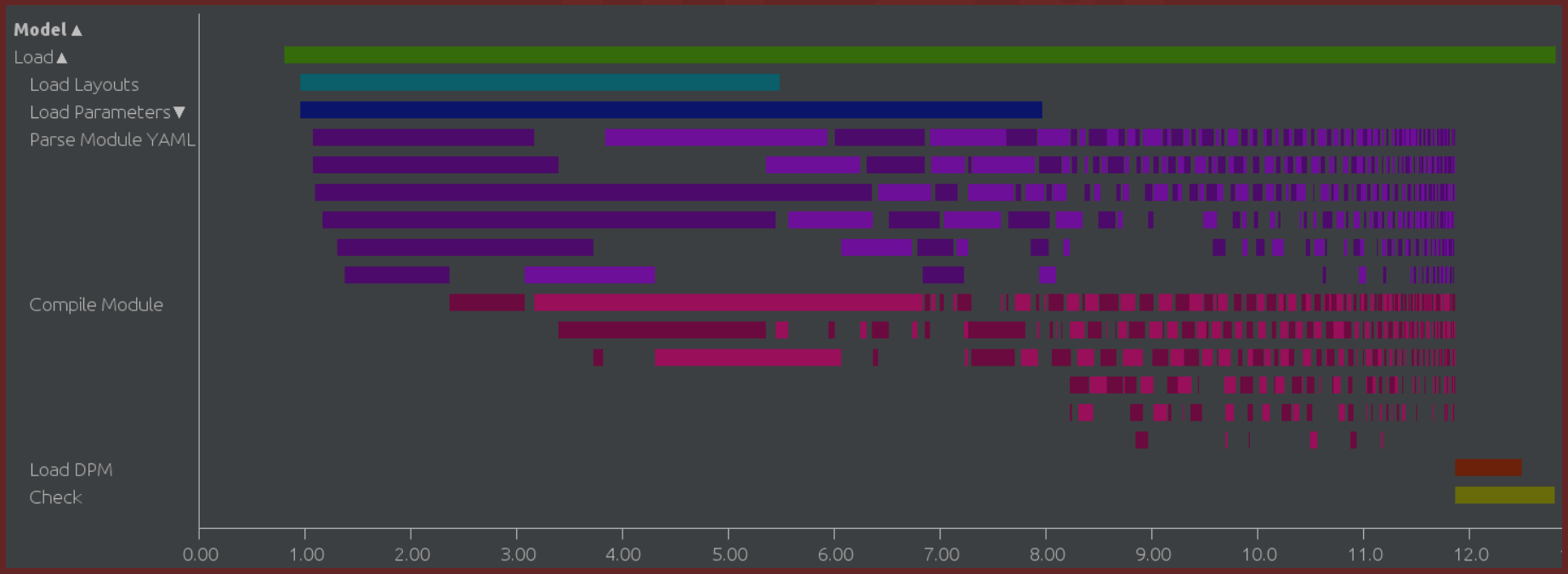


Do the big files first

```
my $modules-load = start @files
    .grep(/ \.(yaml|yml) $/)
    .sort(-*.s)
    .race(batch => 1, degree => 6)
    .map(-> $file { ... })
    .eager;
```

It helps!

Model loading in around 20% of the original time - with few code changes!



Concurrency too

Parallelism gave us an easy speedup

However, implementing e*A*s*ii* was also greatly aided by Perl 6's concurrency support - of note, for live calculations

Datasets

A set of inputs, either entered manually, uploaded, or sometimes derived from other inputs

Current test customer dataset has 250,000 inputs (and each input has a change history, for audit purposes)

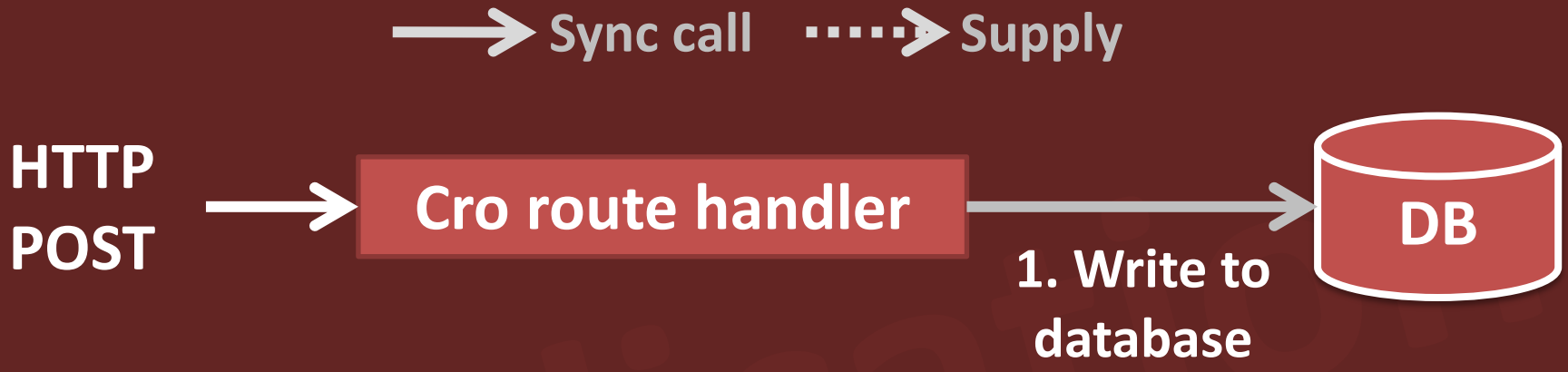
Processing inputs

Processing inputs

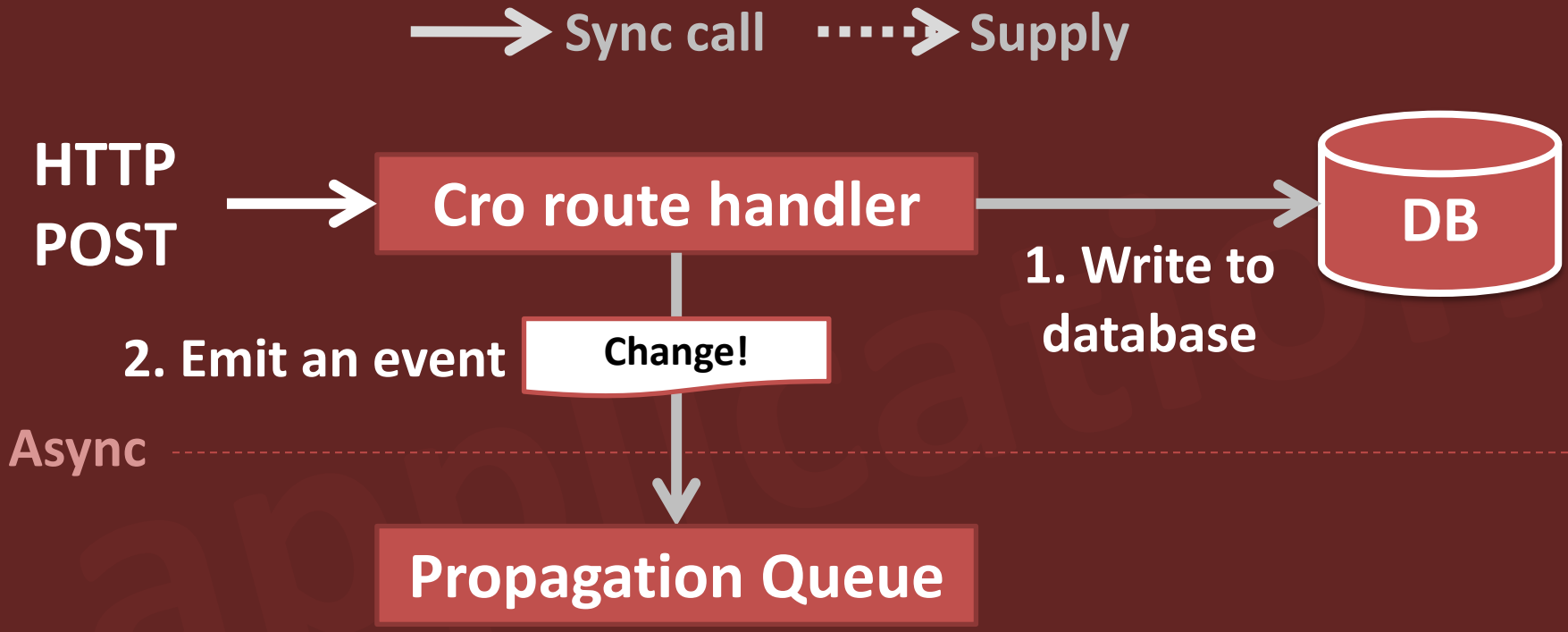
HTTP POST

Cro route handler

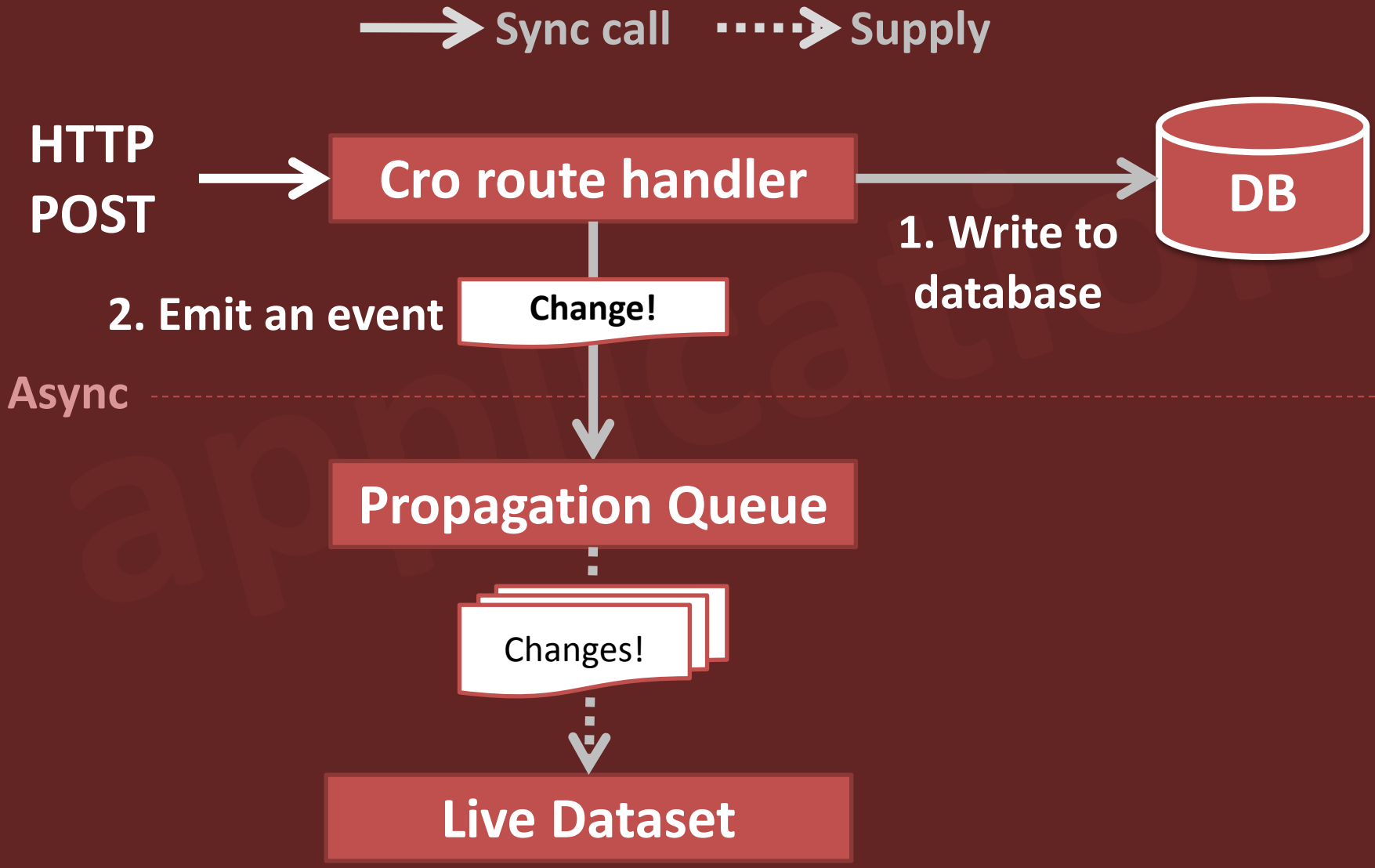
Processing inputs



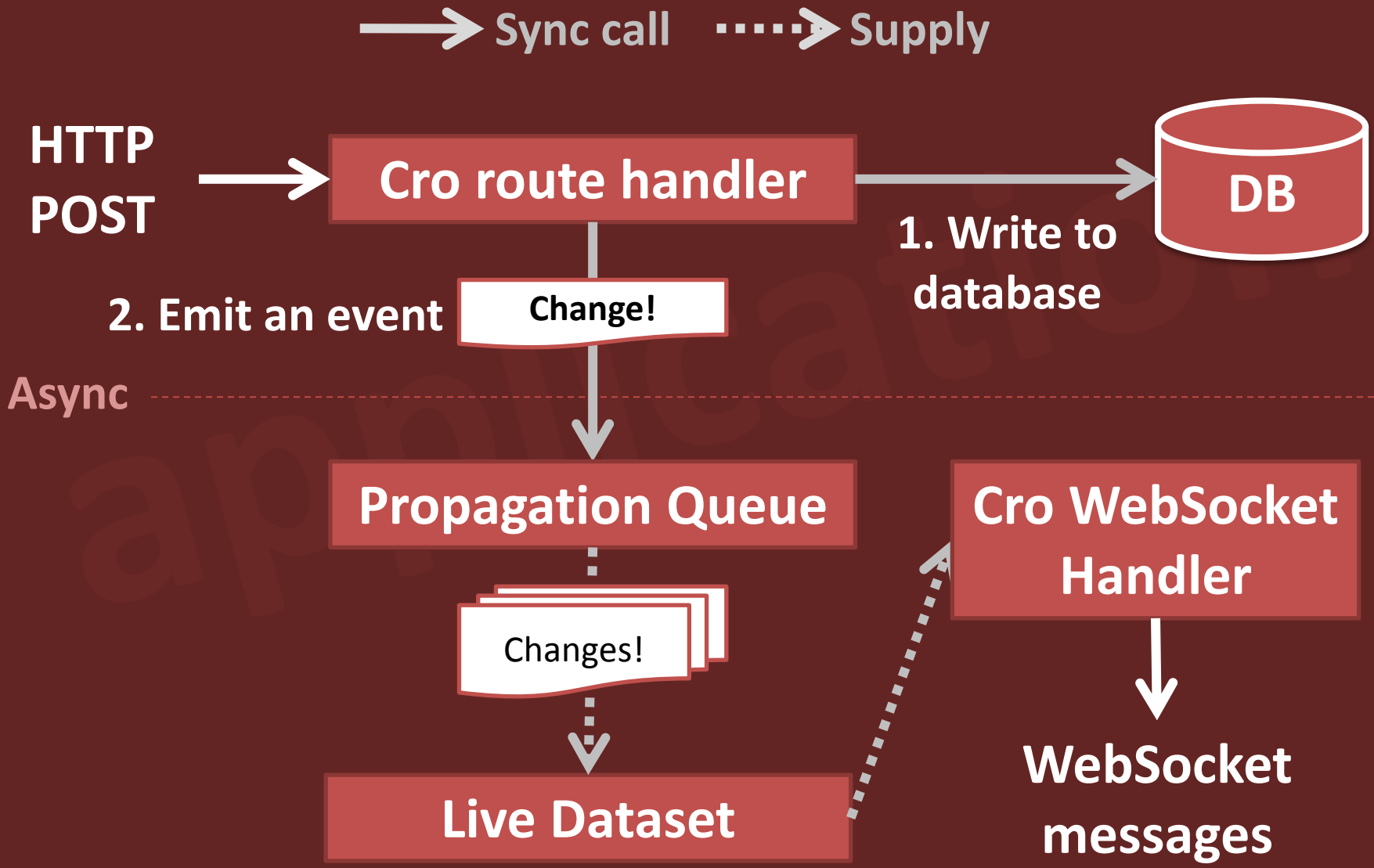
Processing inputs



Processing inputs

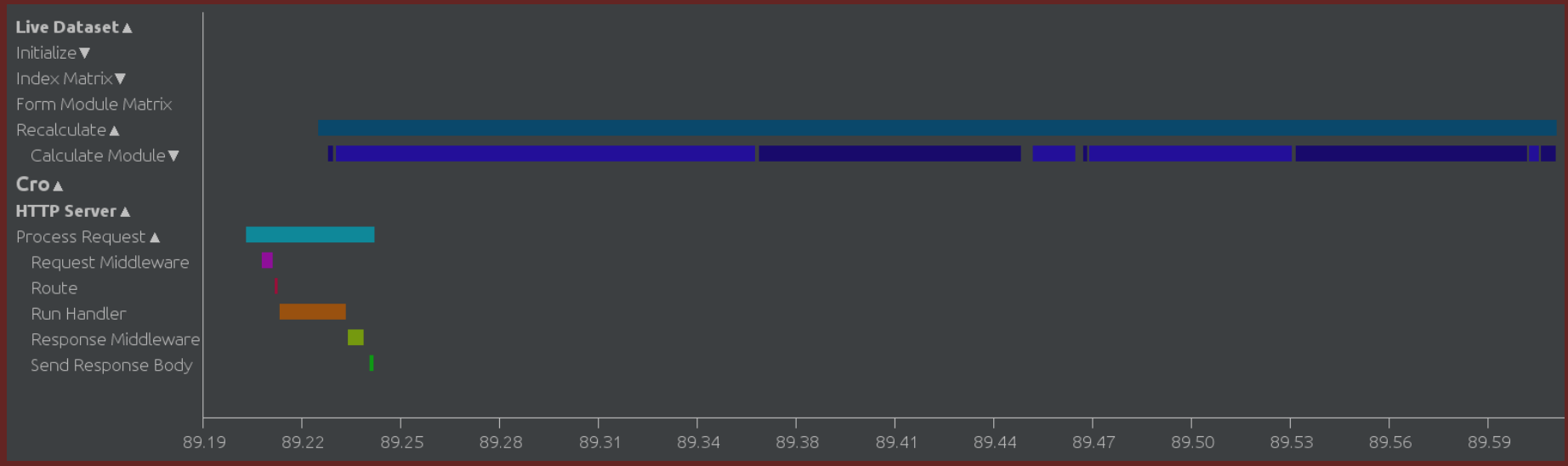


Processing inputs



Processing timeline

HTTP request completes quickly, recalculation runs in the background



Live dataset setup

```
class Easii::LiveDataset {
    has Int $.dataset is required;
    has Supply $.input-source is required;
    has Supplier $.changes .= new;
}
```

Live dataset setup

```
class Easii::LiveDataset {
    has Int $.dataset is required;
    has Supply $.input-source is required;
    has Supplier $!changes .= new;
    submethod TWEAK(:%initial-inputs) {
        start react {
            my $matching-input = $!input-source
                    .grep(*.dataset == $!dataset);
            whenever $matching-input {
                # Perform reclculation...
            }
        }
    }
}
```

Live dataset changes

If there recalculation determines there are changes to a module, emit an event containing them

```
if %formula-changes {
    $!version++;
    $!changes.emit: Easii::LiveDataset::Change.new:
            :$!version, :$module-key, :%formula-changes;
}
```

WebSocket

```
get -> LoggedIn $user, 'easii', 'setupWebsocket', Int :$dataset {

}
```

WebSocket

```
get -> LoggedIn $user, 'easii', 'setupWebsocket', Int :$dataset {
    $app.with-current: $user.customer, -> $state {

    }
}
```

WebSocket

```
get -> LoggedIn $user, 'easii', 'setupWebsocket', Int :$dataset {
    $app.with-current: $user.customer, -> $state {
        web-socket :json, -> $incoming {
            supply {

            }
        }
    }
}
```

WebSocket

```
get -> LoggedIn $user, 'easii', 'setupWebsocket', Int :$dataset {
    $app.with-current: $user.customer, -> $state {
        web-socket :json, -> $incoming {
            supply {
                my $live-dataset = $state.get-live-dataset($dataset);
                whenever $live-dataset.changes -> $change {

                }
            }
        }
    }
}
```

WebSocket

```
get -> LoggedIn $user, 'easii', 'setupWebsocket', Int :$dataset {
    $app.with-current: $user.customer, -> $state {
        web-socket :json, -> $incoming {
            supply {
                my $live-dataset = $state.get-live-dataset($dataset);
                whenever $live-dataset.changes -> $change {
                    my $change-set = $change.for-json;
                    emit $change-set;
                }
            }
        }
    }
}
```

We also...

Data-parallelize formula calculation in modules with many instances

Use a `Channel` to send code to the live dataset for evaluation, the concurrency control meaning we don't evaluate it when recalculating

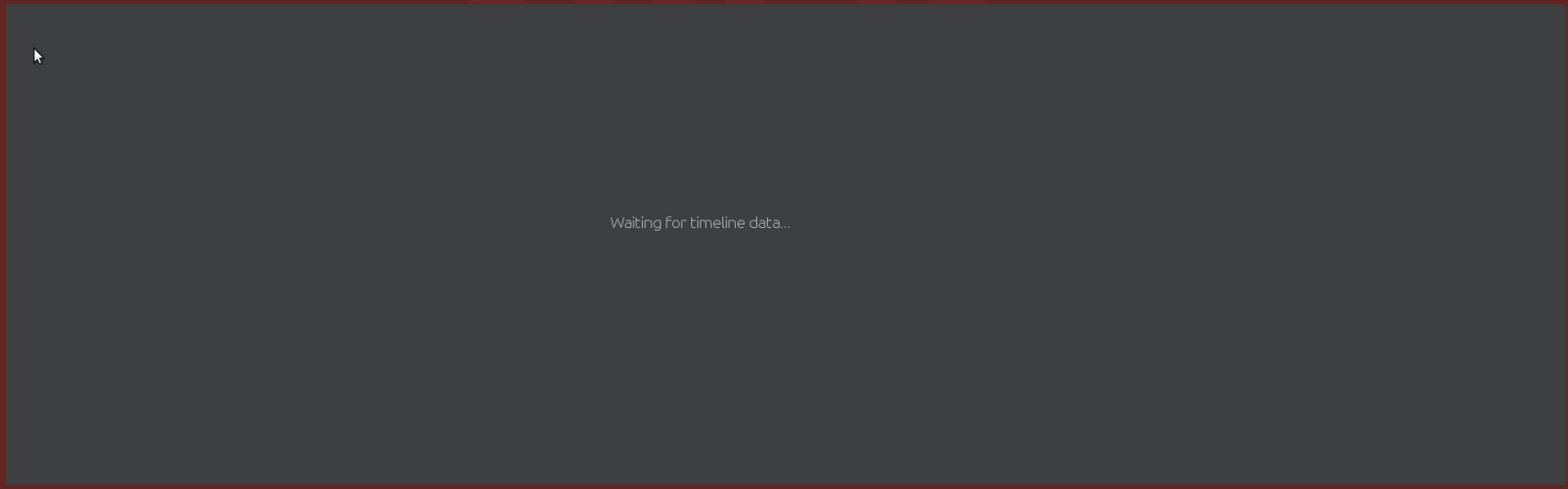
Perl 6: good choice

Perl 6's concurrency features helped us to deliver on the reactive aspects of the application

Meanwhile, the parallelism gave us a bunch of easy performance gains

Lesson: tools are good

Tooling to visualize what's going on in a concurrent/parallel system is a huge win



The

future

of Perl 6 Concurrency

We have a good story - but something is missing

Something, perhaps, that will turn out to be a differentiator

Safety

There's smart folks who feel the future is static proofs

There's others who argue to bind as late as possible

"We need to write tests to assert correctness anyway"

"It's easier to debug a concrete situation than a theoretical type error"

But what if the failure is a data race that happens

1 time in 10,000?

What we kind of need is...

future

What we kind of need is...

....ummm....

What we kind of need is...

....ummm....

reliable failure!

We need a Perl-ish solution.

That's a research problem.

But it's one I believe we should take on.

Why?

So we can make getting the easy things right easier

So we can make getting the hard things right possibler

So we can make the whipped up concurrent or parallel program do the right thing

Because we torment the language implementer for the sake of the language user

These are the things that define a Perlish language

These are the things that define a Perlish library

Being easy to get in to

Whipping up ideas together

Trying to do the right thing

Realizing others are trying to do the right thing

That, to me, seems like a way to be a Perl community

Thank you!

@ jonathan@edument.cz

W jnthn.net

 jnthnwrthngtn

 jnthn

 cro.services

 commaide.com