Introduction to Cro

Building and consuming services in Perl 6

Jonathan Worthington | EDUMENT

What is Cro?

A set of libraries and tools for building distributed systems in Perl 6

Useful for both consuming existing services, building new services, and building entire systems of services

What is a distributed system?

One involving multiple processes that communicate with each other

These may be spread over many different machines, datacenters, countries, etc.

"First, do one thing well"

We're focusing first on HTTP services, since they're such a common choice

Also early work on ZeroMQ

Many ideas for the future

Built for Perl 6

Cro isn't a Perl 6 port of anything

It's a ground-up implementation, designed to feel natural to Perl 6 programmers and to make the most of what Perl 6 has to offer

Who?

Cro development is sponsored by Edument (we also provide Cro support and consulting)

Open Source (Artistic License), more than a dozen contributors so far

A look at Perl 6 Supplies

What is a Supply?

An asynchronous stream of values

Finite or infinite

If finite, may terminate naturally or exceptionally

Dual of iteration

Iterables <u>pull</u> values through a pipeline

Supplies <u>push</u> values through a pipeline

A Supply of timer ticks

We `tap` a `Supply` to start the flow of values, providing a handler

```
my $ticks = Supply.interval(0.5);
my $tap = $ticks.tap: {
    say now;
}
sleep 3;
$tap.close;
```

Syntactic relief

The react/whenever construct for processing asynchronous data

```
react {
    whenever Supply.interval(0.5) {
        say now;
    }
    whenever Promise.in(3) {
        done;
    }
}
```

Totally cheating HTTP client

```
my $socket = await IO::Socket::Async.connect:
    'moarvm.org', 80;
await $socket.print:
    "GET / HTTP/1.0\r\nHost: moarvm.org\r\n\r\n";
react {
    whenever $socket -> $chars {
        print $chars;
    }
}
```

Totally cheating HTTP server

```
react {
    whenever IO::Socket::Async.listen('0.0.0.0', 8080)
      -> $conn {
        whenever $conn {
            whenever $conn.print:
                    "HTTP/1.0 200 OK\r\n" ~
                    "Content-type: text/plain\r\n\r\n" ~
                    "Wow a HTTP response!\n" {
                $conn.close;
            }
        }
    }
}
```

The `supply` construct

Process one or more asynchronous streams, and emit values into a result stream

Automatic concurrency control (one message at a time), like `react`

```
class TimedOut is Exception {
    method message() { "Timed out" }
}

sub timeout(Supply() $s, Real() $seconds) {
    supply {
        sub refresh-timeout() {
            state $tap;
            $tap.?close;
            $tap = do whenever Promise.in($seconds) {
                die TimedOut.new;
            }
        }
        whenever $s -> $msg {
            refresh-timeout;
            emit $msg;
        }
        refresh-timeout; # Set initial timeout
    }
}
```

```
react {
    whenever IO::Socket::Async.listen('0.0.0.0', 8080)
      -> $conn {
        whenever timeout($conn, 10) {
            whenever $conn.print:
                    "HTTP/1.0 200 OK\r\n" ~
                    "Content-type: text/plain\r\n\r\n" ~
                    "Wow a HTTP response!\n" {
                $conn.close;
            }
            QUIT {
                when TimedOut {
                    $conn.close;
                }
            }
        }
    }
}
```

Reactive Pipelines in Cro

Cro is centered around pipelines - chains of `Supply` processors

These pipelines are made up of components that are composed together to form a client, server, message processor, etc.

As a really simple example, we'll built a TCP service that will apply ROT13 to anything it is sent, and send the result back to the client

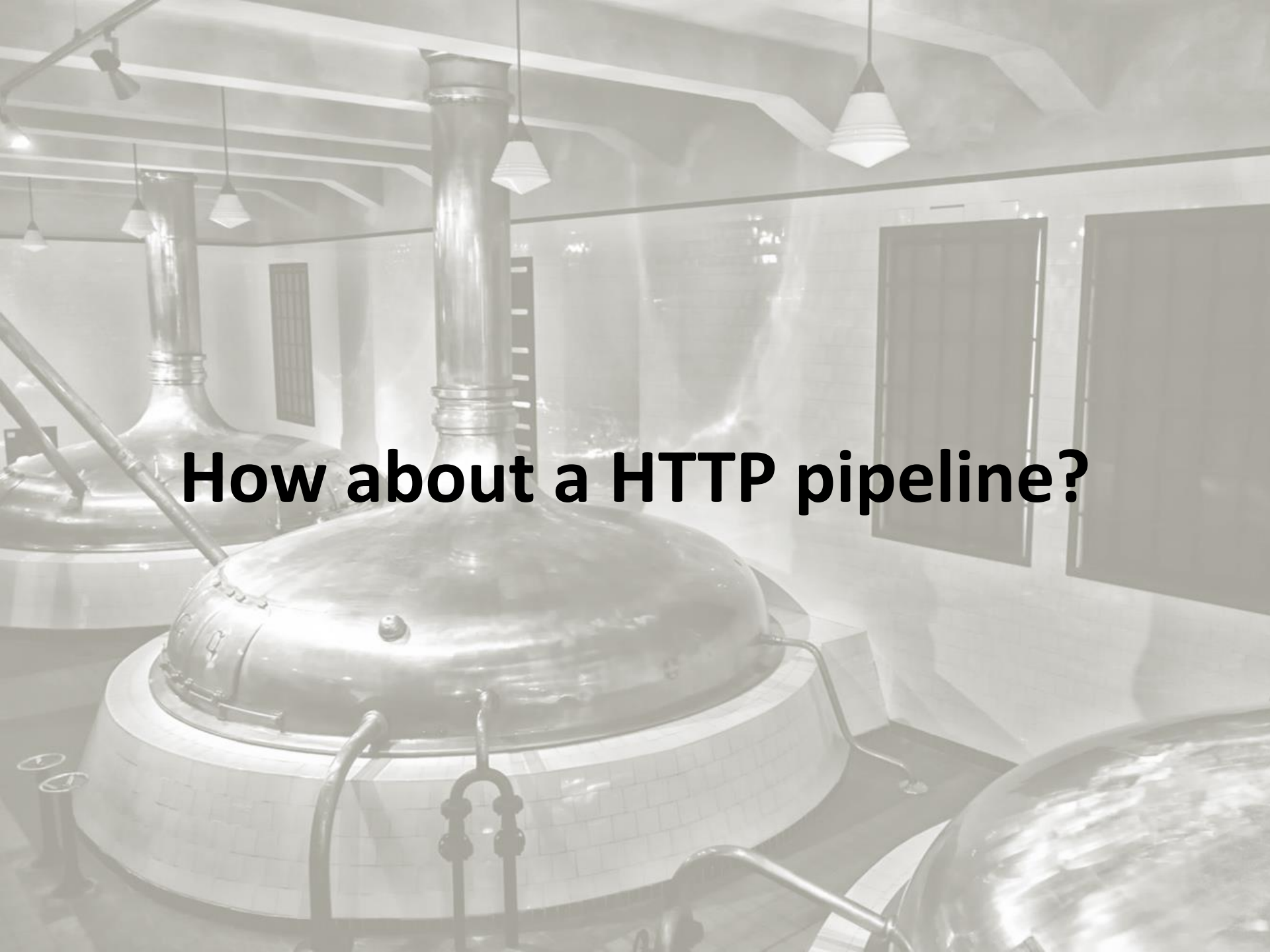
First, write a transform

```
use Cro;
use Cro::TCP;

class Rot13 does Cro::Transform {
    method consumes() { Cro::TCP::Message }
    method produces() { Cro::TCP::Message }
    method transformer(Supply $messages --> Supply) {
        supply {
            whenever $messages {
                emit Cro::TCP::Message.new: data =>
                    .data.decode('latin-1')
                    .trans('a..mn..z' => 'n..za..m', :ii)
                    .encode('latin-1')
            }
        }
    }
}
```

Compose it into a service and run it

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;

$rot13.start;
react whenever signal(SIGINT) { $rot13.stop; done }
```

Connection management and response sending provided automatically in composition



How about a HTTP pipeline?

Again, we write a transform...

```
use Cro::HTTP::Request;
use Cro::HTTP::Response;

class MyApp does Cro::Transform {
    method consumes() { Cro::HTTP::Request }
    method produces() { Cro::HTTP::Response }
    method transformer(Supply $reqs) {
        supply whenever $reqs -> $request {
            my $res = Cro::HTTP::Response.new(
                :$request, :200status);
            $res.append-header('Content-type',
                               'text/plain');
            $res.set-body("Hello from Cro\n");
            emit $res;
        }
    }
}
```

...and compose it into a service

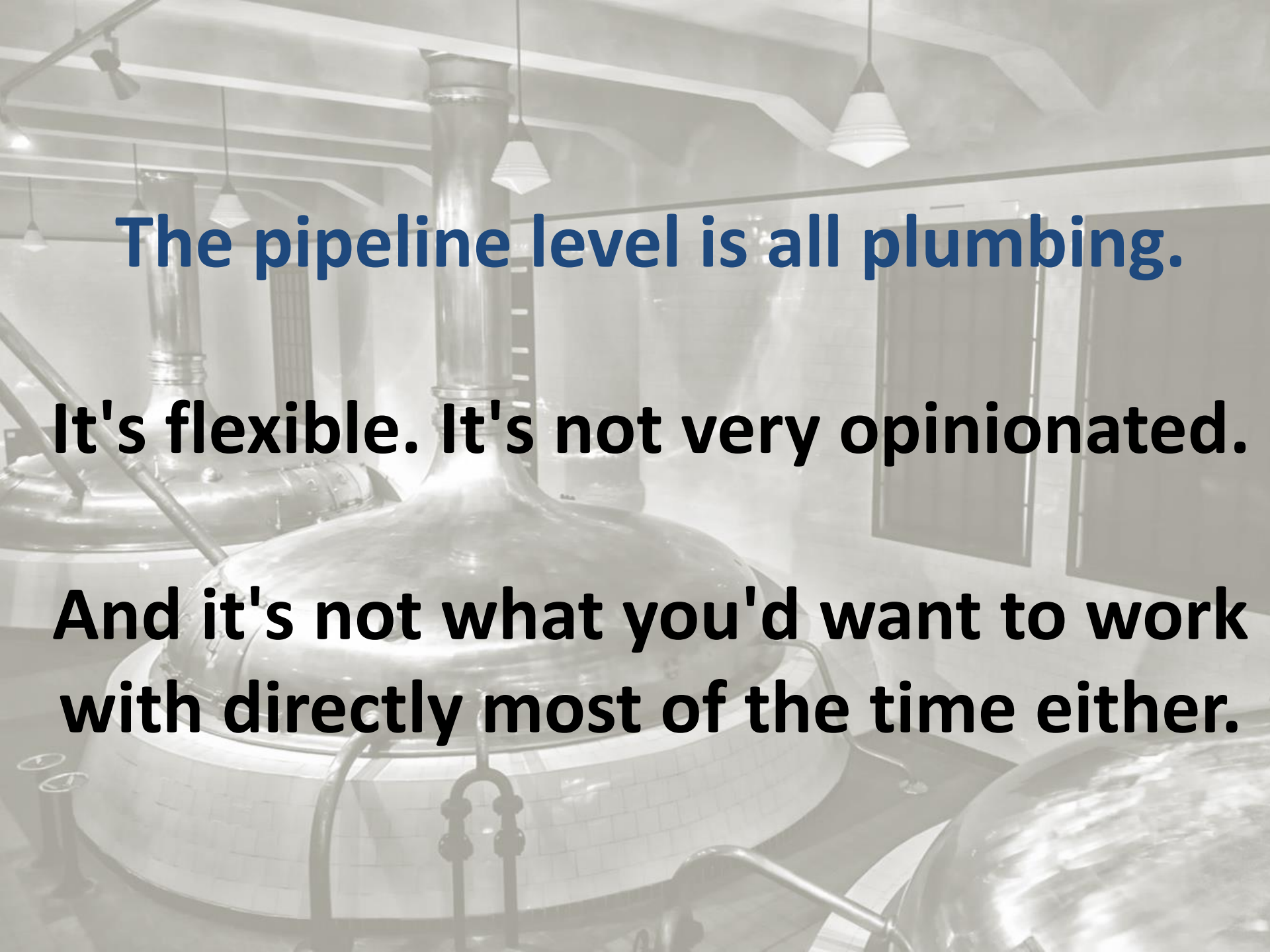
```
use Cro
use Cro::TCP;
use Cro::HTTP::RequestParser;
use Cro::HTTP::ResponseSerializer;

my Cro::Service $http-hello = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Cro::HTTP::RequestParser.new,
    MyApp,
    Cro::HTTP::ResponseSerializer.new;

$http-hello.start;
react whenever signal(SIGINT) { $http-hello.stop; done; }
```

Want logging? Just add it!

```
use Cro
use Cro::TCP;
use Cro::HTTP::RequestParser;
use Cro::HTTP::ResponseSerializer;
use Cro::HTTP::Log::File;

my Cro::Service $http-hello = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Cro::HTTP::RequestParser.new,
    MyApp,
    Cro::HTTP::Log::File.new,
    Cro::HTTP::ResponseSerializer.new;

$http-hello.start;
react whenever signal(SIGINT) { $http-hello.stop; done; }
```

Many web libraries and frameworks have a concept of middleware

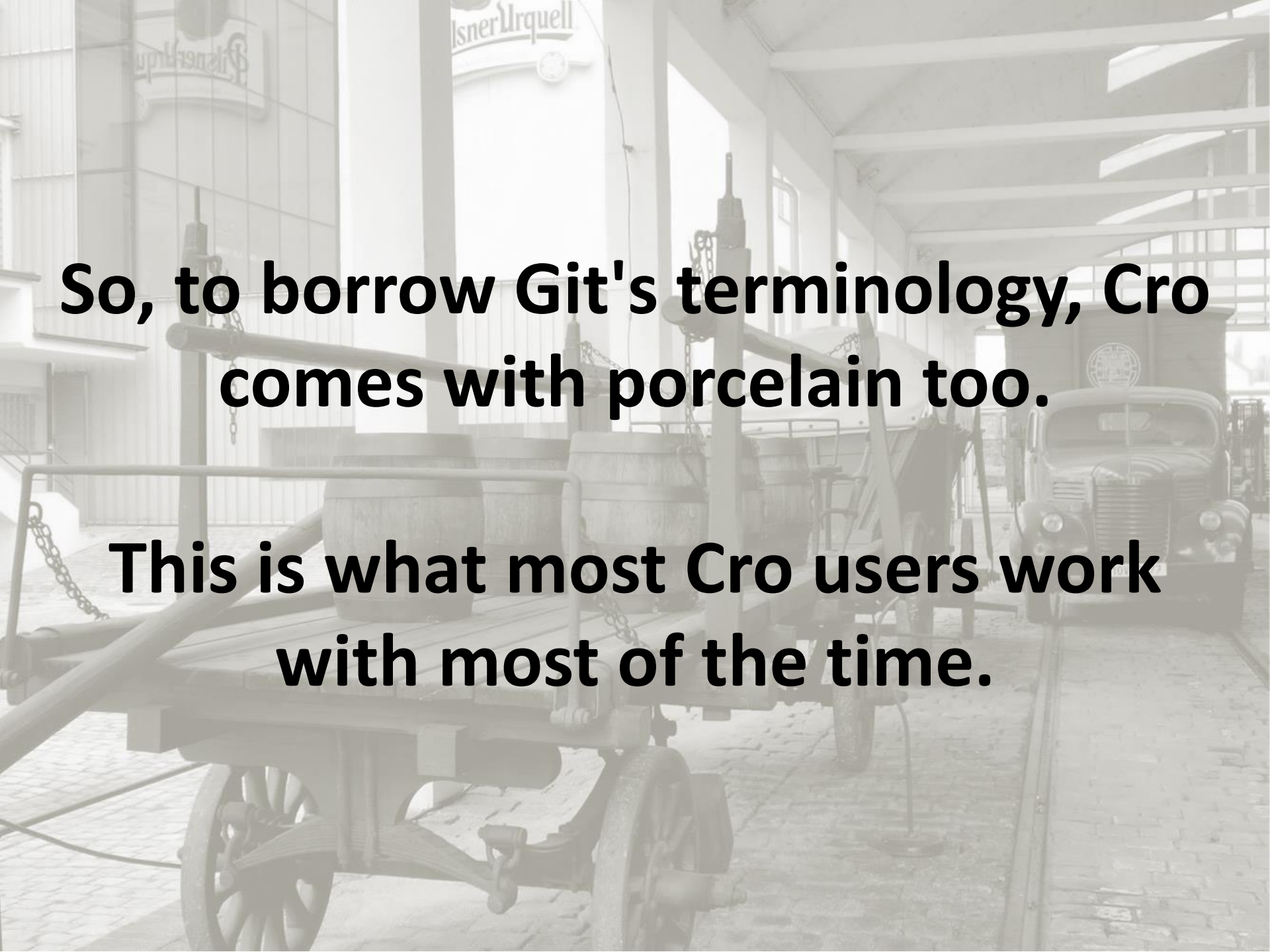
In Cro, everything - the network I/O, the request parser, the response serializer - is middleware



The pipeline level is all plumbing.

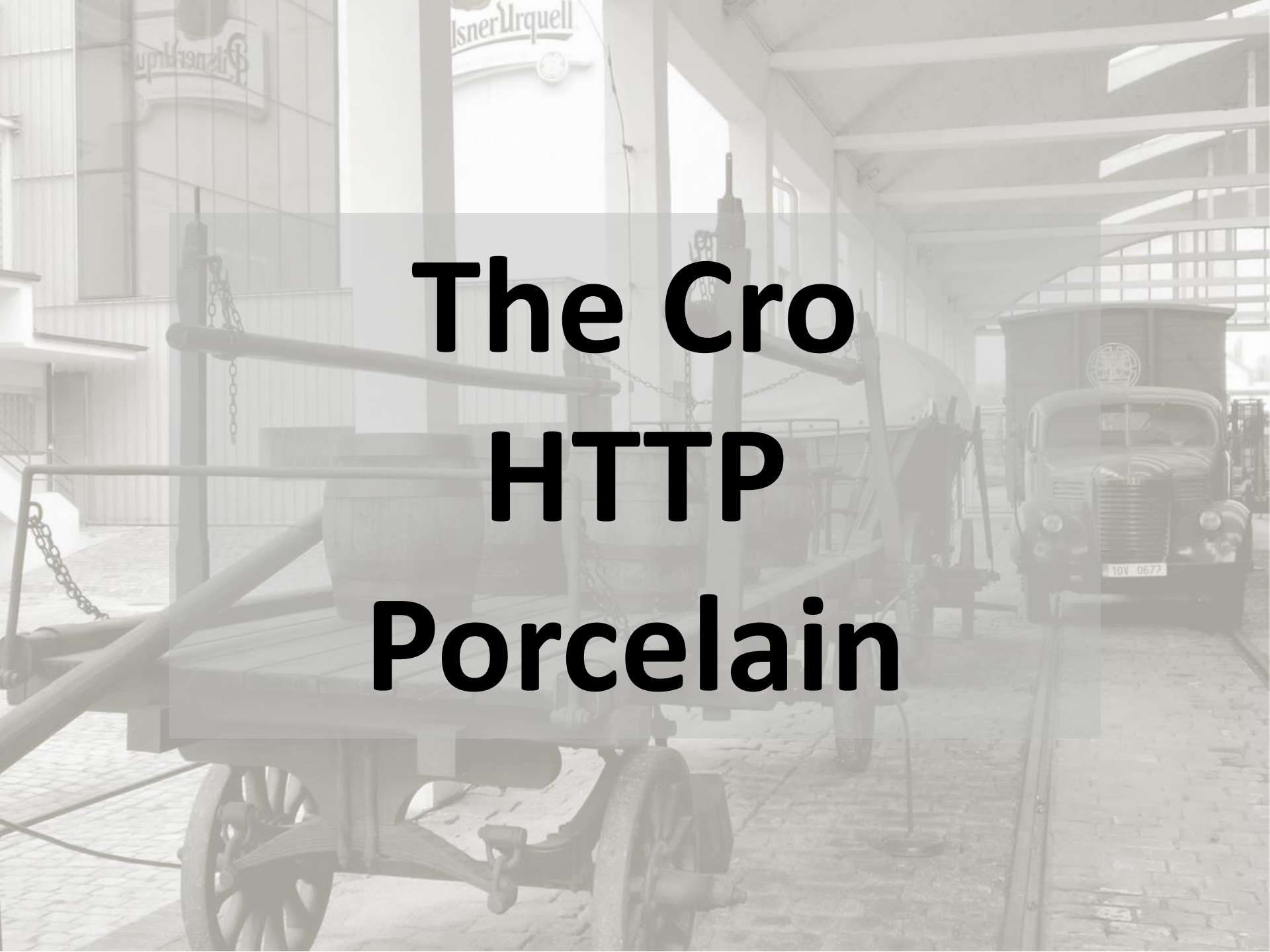
It's flexible. It's not very opinionated.

And it's not what you'd want to work with directly most of the time either.



So, to borrow Git's terminology, Cro comes with porcelain too.

This is what most Cro users work with most of the time.



The Cro HTTP Porcelain

Cro::HTTP::Server builds HTTP server pipelines

```
use Cro::HTTP::Server;
use Cro::HTTP::Log::File;

my Cro::Service $http-hello = Cro::HTTP::Server.new:
    :host<0.0.0.0>, :port<10000>,
    :ssl{
        private-key-file => 'server-key.pem',
        certificate-file => 'server-crt.pem';
    },
    :application(MyApp),
    :after[Cro::HTTP::Log::File.new];
$http-hello.start;
react whenever signal(SIGINT) { $http-hello.stop; done; }
```

Cro::HTTP::Router provides a nice way to write request handlers

```
use Cro::HTTP::Router;
my $application = route {
    get -> {
        content 'text/plain', "Hello from Cro\n";
    }
}
```

And the thing a route block returns is a Cro::Transform from requests to responses

The router uses Perl 6 signatures to specify the routes, as well as other requirements on the request

Also has a bunch of functions to help with constructing responses

Literals and positional parameters match URI path segments

```
get -> 'greet', $name {
    content 'text/plain', "Hello, $name\n";
}
```

Named parameters are sourced from the query string, by default

```
get -> 'greet', $name, :$greeting = 'hello' {
    content 'text/plain', "$greeting.tclc(), $name\n";
}
```

Slurpy parameters match an arbitrary number of path segments - very handy when serving assets

```
get -> 'css', *@path {
    static 'static-content/css', @path;
}

get -> 'js', *@path {
    static 'static-content/js', @path;
}
```

(Includes protection against ../ hackery too!)

Use type constraints, including subset types, to restrict the allowable values of route segments, query string values, etc.

```
my subset UUIDv4 of Str where /^
    <[0..9a..f]> ** 12
    4 <[0..9a..f]> ** 3
    <[89ab]> <[0..9a..f]> ** 15
    $/;

get -> 'user-log', UUIDv4 $id {
    ...
}
```

The request object has an `auth` property, settable by authorization or session middleware

Provided the type in there does the `Cro::HTTP::Auth` role, it can be taken - and maybe constrained - ahead of the route segments

```
# Declare subset types for authorization needs
my subset Admin of My::App::Session where .is-admin;
my subset LoggedIn of My::App::Session where .is-logged-in;

my $application = route {
    get -> LoggedIn $user, 'my', 'profile' {
        # Use $user in some way
    }

    get -> Admin, 'system', 'log' {
        # Just use the type and don't name a variable, if
        # the session/user object is not needed
    }
}
```

Appropriate HTTP error codes

Route segments don't match → 404

Method doesn't match → 405

Auth doesn't match → 401

Query string doesn't match → 400

Support for HTTP/2 push promises

Silently ignored for HTTP/1 requests

```
get -> {
    push-promise '/css/global.css';
    push-promise '/css/main.css';
    content 'text/html', $some-content;
}

get 'css', *@path {
    cache-control :public, :maxage(300);
    static 'assets/css', @path;
}
```

Cro::HTTP::Router::WebSocket

```
my $chat = Supplier.new;
get -> 'chat' {
    web-socket -> $incoming, $close {
        supply {
            whenever $incoming -> $message {
                $chat.emit(await $message.body-text);
            }
            whenever $chat -> $text {
                emit $text;
            }
            whenever $close {
                $chat.emit("A user left the chat");
            }
        }
    }
}
```

Use `include` to compose `routes`

```
module FooApp::Search;
sub search-routes is export {
    route {
        get -> :$query {
            ...
        }
    }
}
```

```
use FooApp::Search;
my $app = route {
    # Prefix with /search
    include search => search-routes();
}
```

The `include` function only works with other `route` blocks

By contrast, `delegate` works with any `Cro::Transform` that turns a request into a response - so anything can be mounted there

Body parsing/serialization

```
post -> 'product' {
    # When it's application/json, and destructures as
    # required, process it. Otherwise, 400 Bad Request.
    request-body 'application/json' => -> (
            Str :$name!,
            Str :$description!,
            Real :$price! where * > 0) {
        # Store stuff.
        my $id = $store.add-product($name, $description,
            $price);

        # Send JSON response.
        content 'application/json', { :$id };
    }
}
```

Built-in support for URL-encoded and multi-part form data, and JSON

Can plug in your own body parsers and serializers, at the server level or just within a given `route` block

Per route block middleware

```
# Before any matching route in this block
before My::Request::Middleware;

# After processing any matching route in this block
after My::Response::Middleware;

# For simple things, block form of before/after middleware
before {
    unless .connection.peer-host eq '127.0.0.1' | '::1' {
        forbidden;
    }
}
after {
    header 'Strict-transport-security',
        'max-age=31536000; includeSubDomains'
}
```

Requests are processed in the Perl 6 thread pool, so applications handle parallel requests automatically


Did I mention there's also a
Cro::HTTP::Client?

Just one example - to show how to receive HTTP/2 push promises ☺

```
use Cro::HTTP::Client;

my $client = Cro::HTTP::Client.new(:http<2>, :push-promises);
my $resp = await $client.get($uri);

react whenever $resp.push-promises -> $pp {
    whenever $pp.response -> $resp {
        whenever $resp.body-blob -> $blob {
            say "Push of $pp.target() " ~
                "(status: $resp.status(), bytes: $blob.bytes())";
        }
        QUIT {
            default {
                # Ignore cancelled push promises
            }
        }
    }
}
```

The client handles...

HTTPS, HTTP/2.0
Persistent connections
Pluggable body parsers/serializers
Streaming response bodies
Automatic redirect following
Optional cookie jar (pluggable too)

Closing Remarks

This is just the beginning for what we'd like to achieve with Cro

However, judging by feedback from early adopters, it's already a useful and interesting beginning

Learn more:

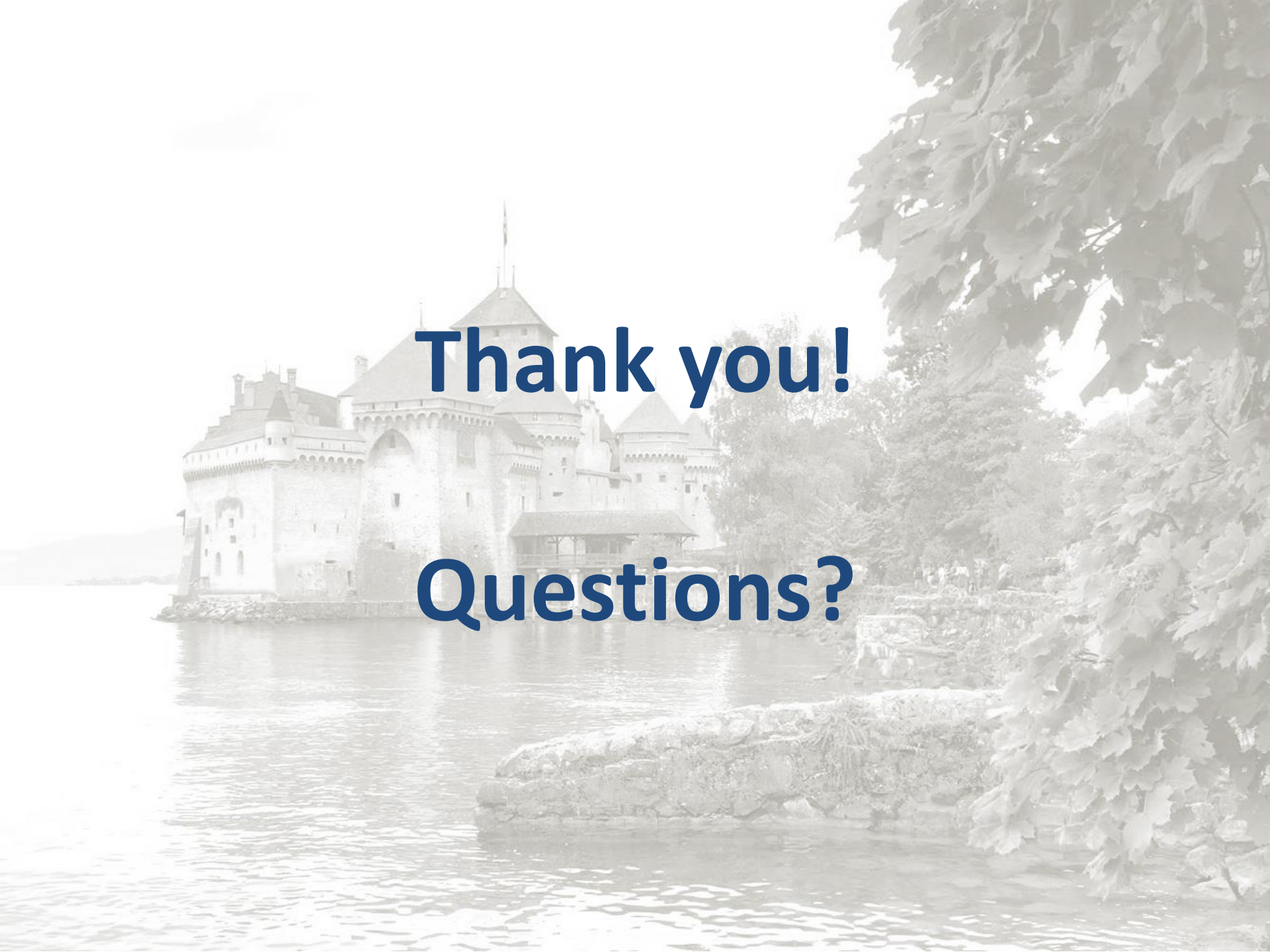
http://cro.services/

IRC:

#cro on freenode.org

Twitter:

@croservices



Thank you!

Questions?