ALEXANDER VARWIJK - 16TH OCTOBER 2025

THINKING ASYNC

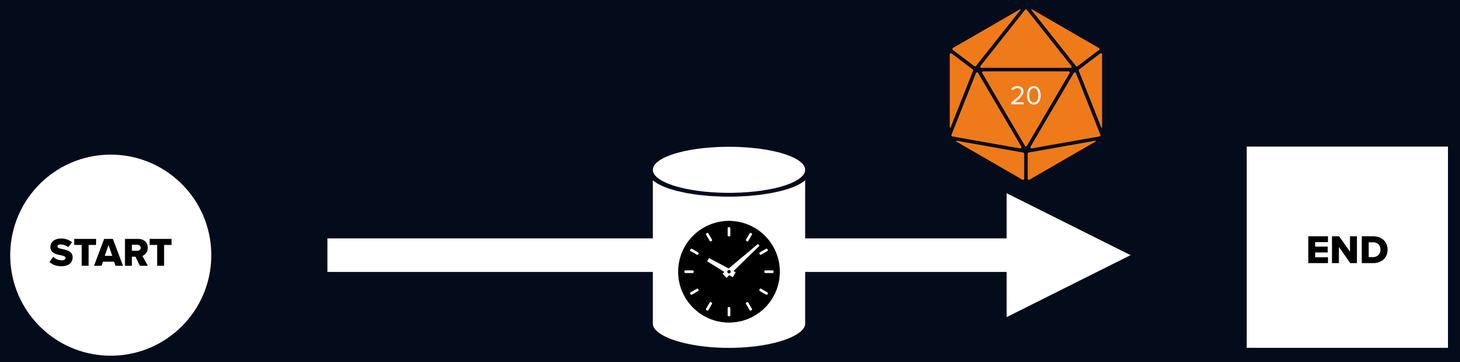
GET EXCITED ABOUT ASYNC DRUPAL
HELP ME MAKE THIS A REALITY

https://www.alexandervarwijk.com/talks/2025-10-16-thinking-async

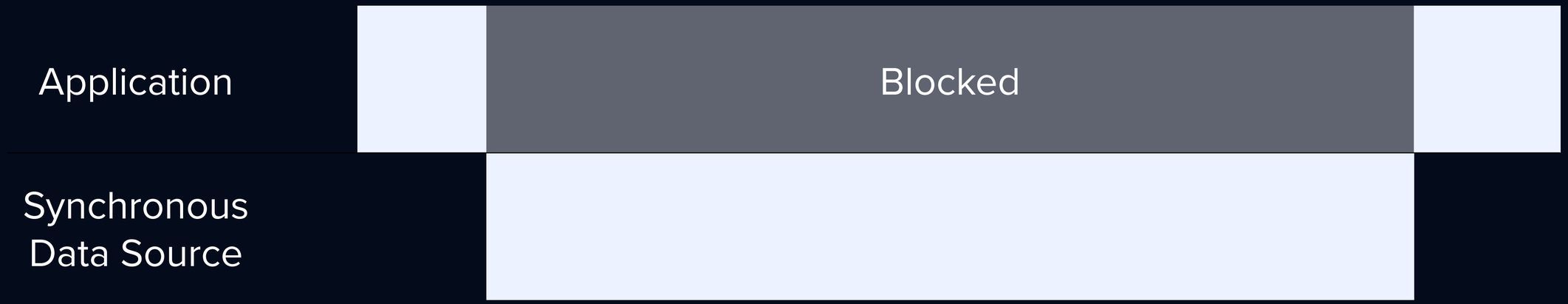
SYNCHRONOUS PROGRAMMING

SYNCHRONOUS DEMO

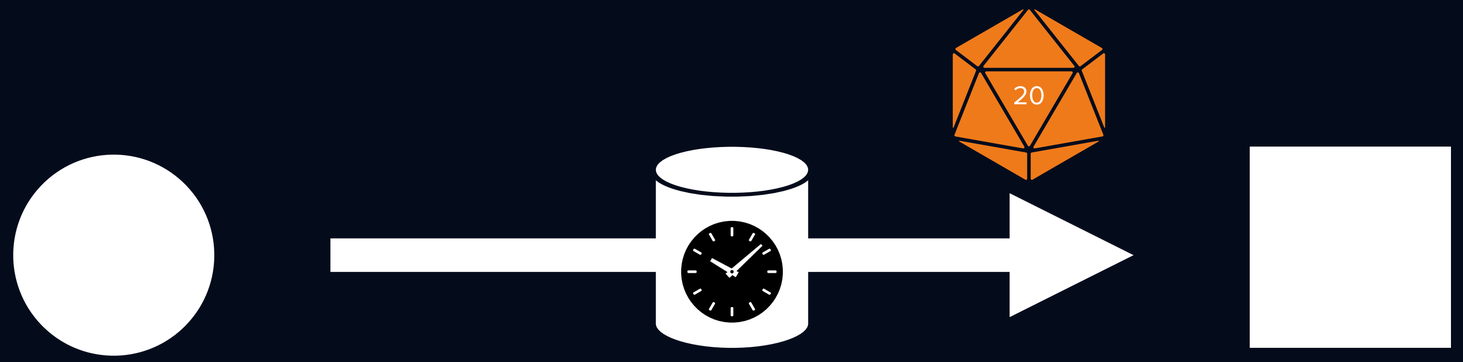
SYNCHRONOUS DEMO



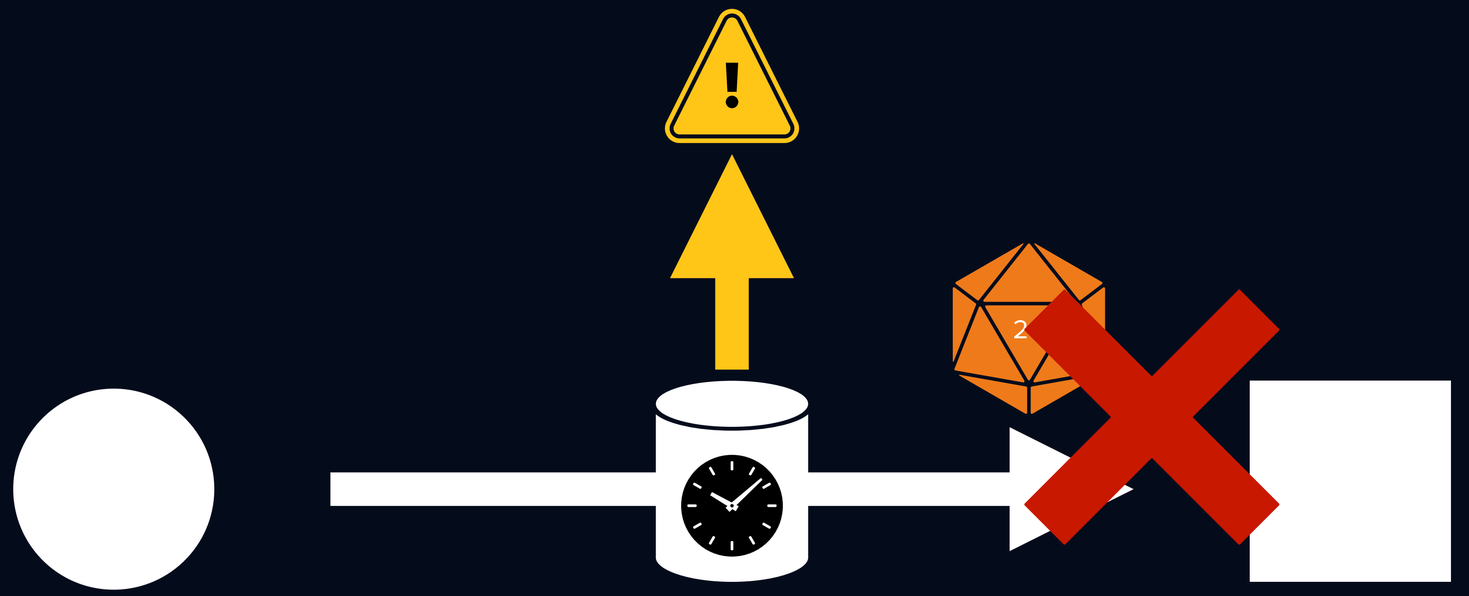
SYNCHRONOUS DEMO



SYNCHRONOUS DEMO



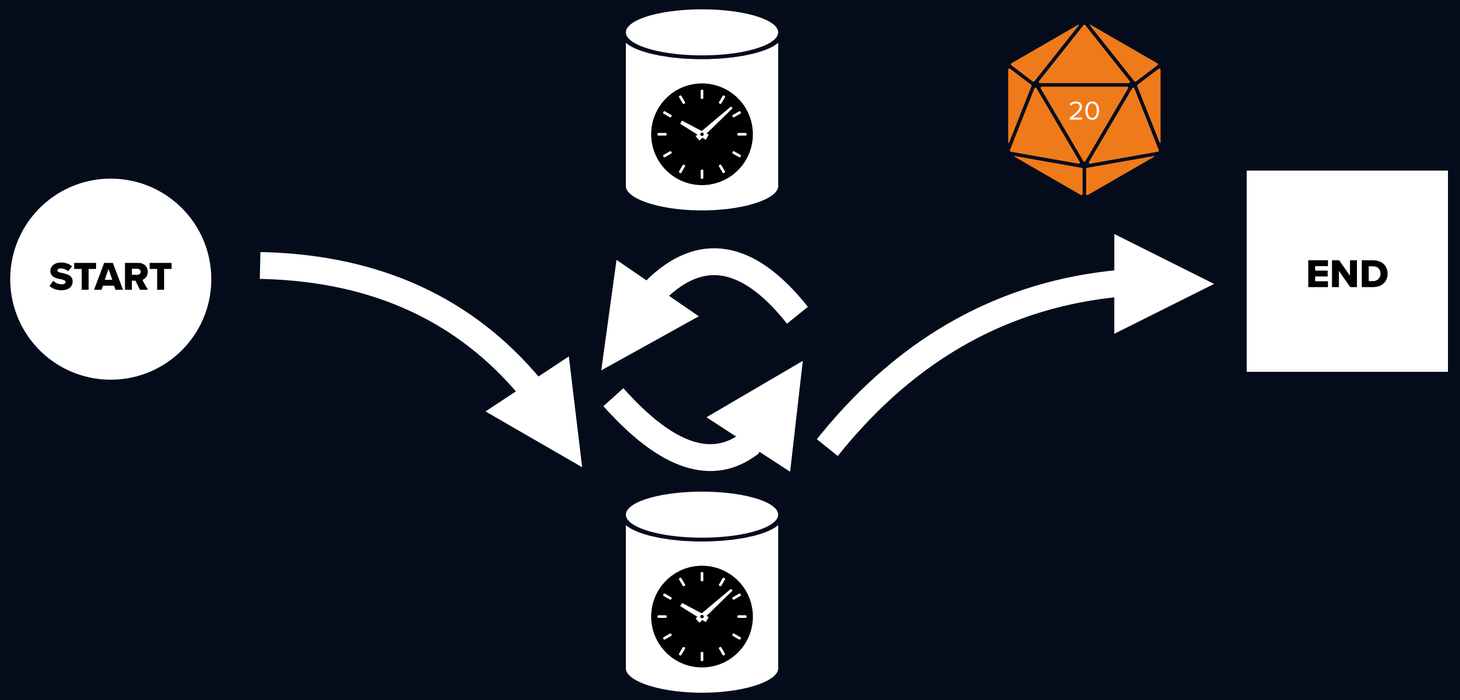
SYNCHRONOUS DEMO



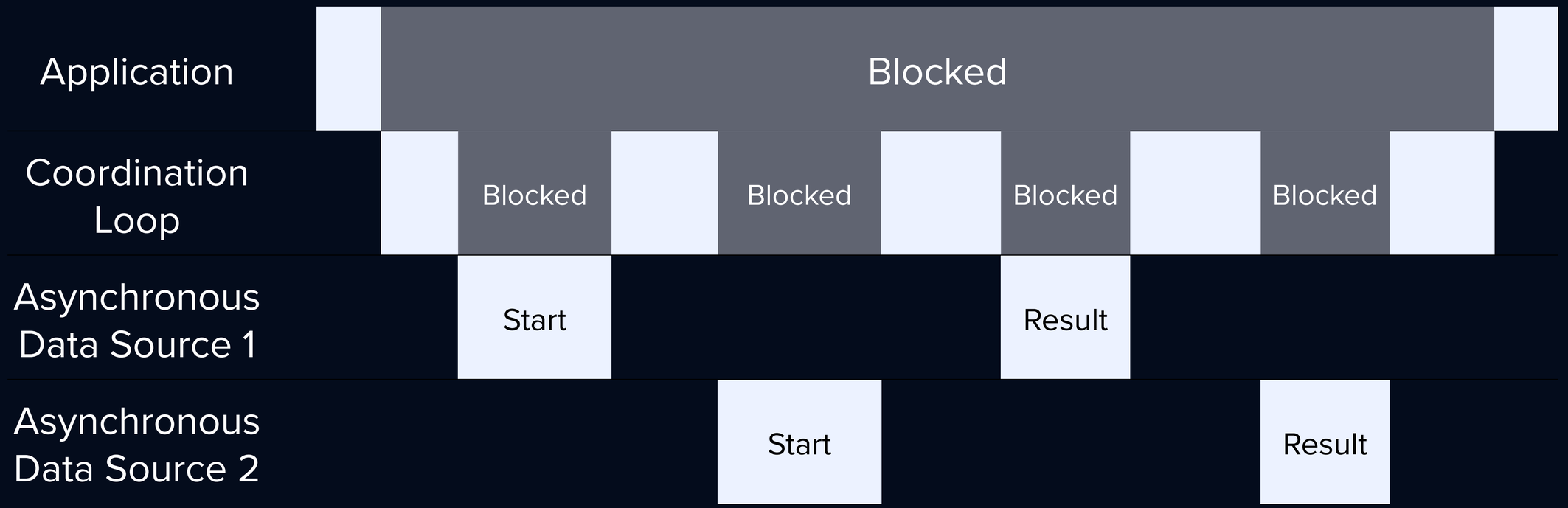
WHY ASYNC

ASYNCHRONOUS DEMO

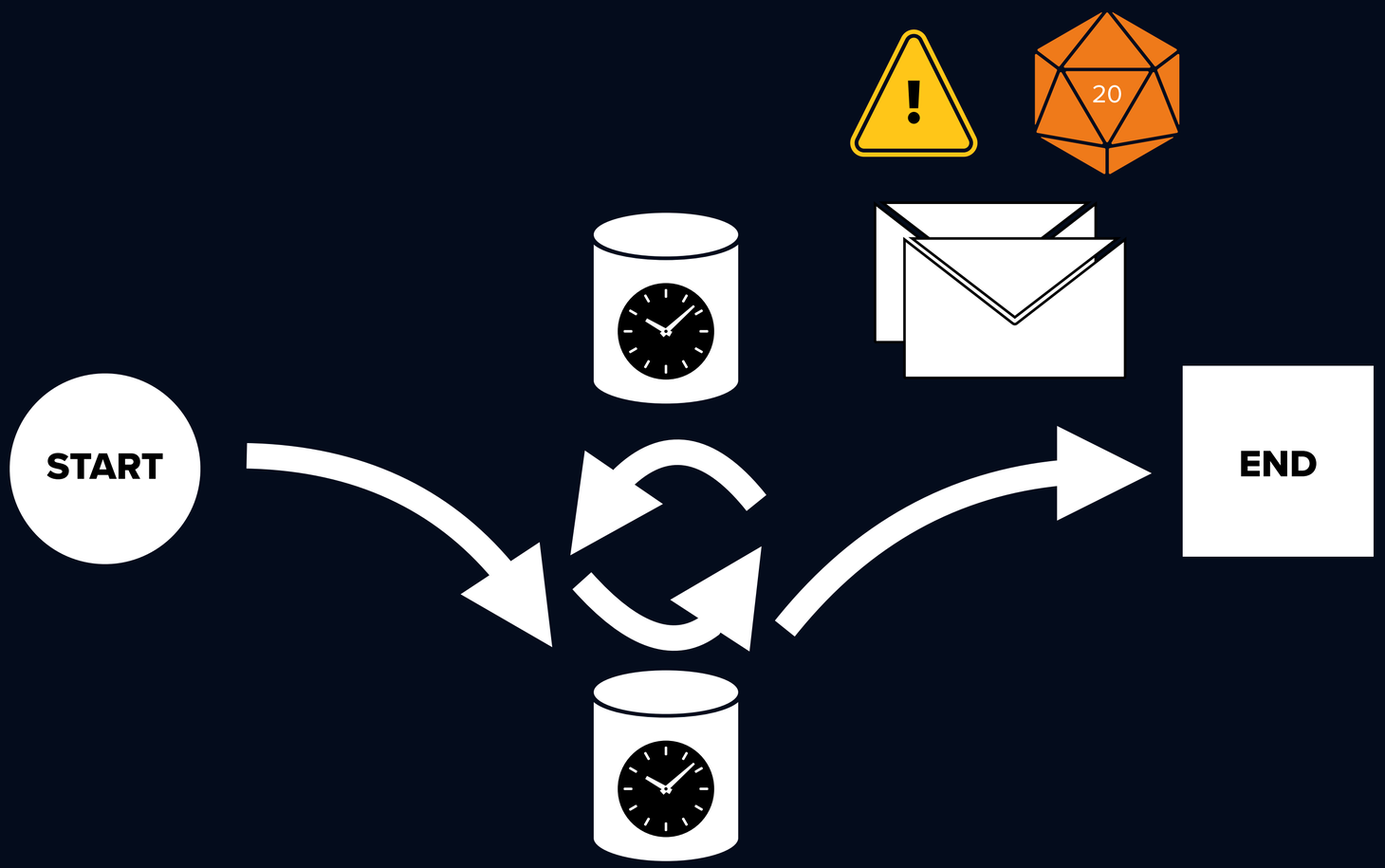
ASYNCHRONOUS DEMO



SYNCHRONOUS DEMO



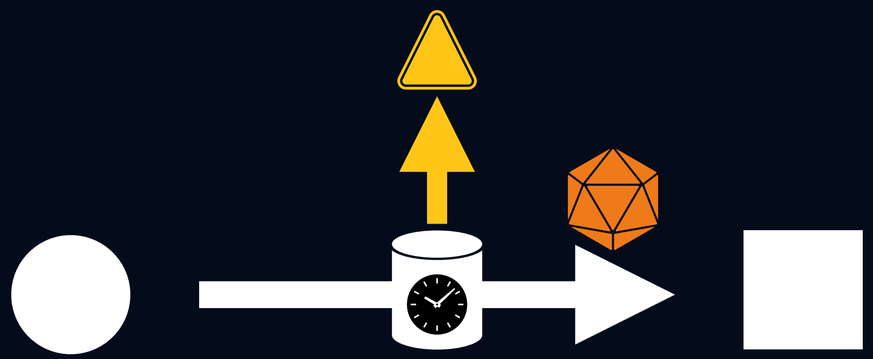
ASYNCHRONOUS DEMO



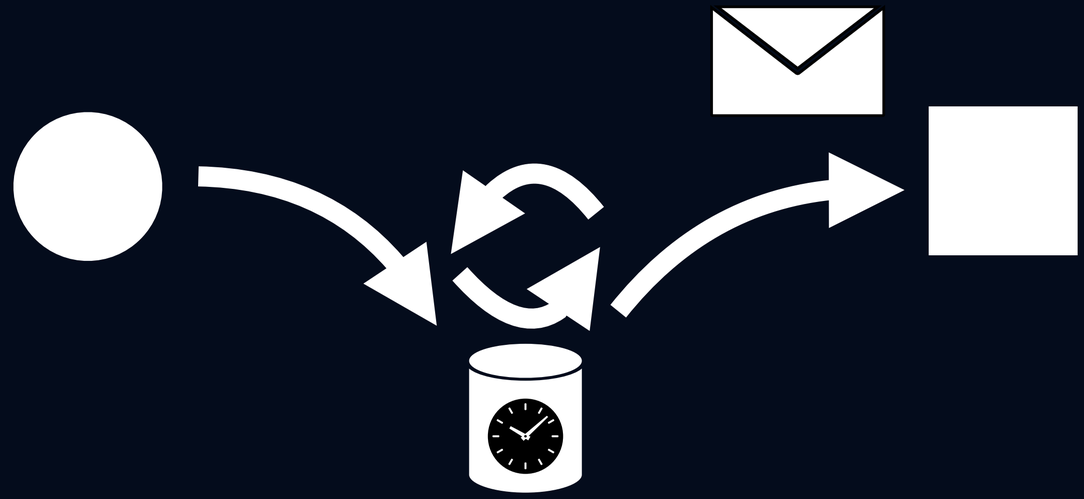
FUNCTION COLOR AND PHP FIBERS

FUNCTION COLOR AND PHP FIBERS

Synchronous



Asynchronous



FUNCTION COLOR AND PHP FIBERS

Synchronous

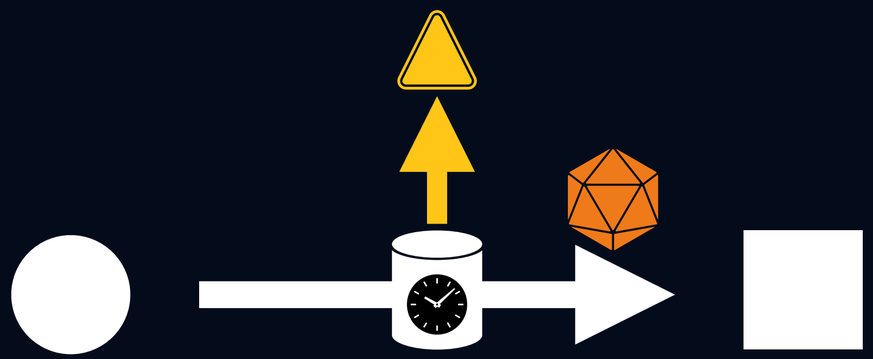
```
function expensive_calculation() : int {
  $something_went_wrong = FALSE;
  if ($something_went_wrong) {
    throw new \RuntimeException("Oh no!");
  }
  return 42;
}
```

Asynchronous

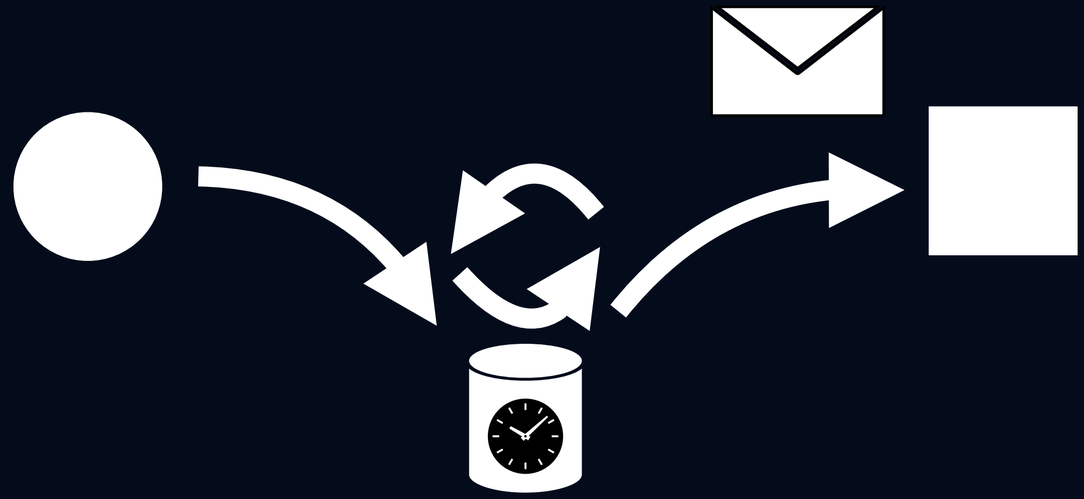
```
function expensive_calculation() : Promise {
  $something_went_wrong = FALSE;
  if ($something_went_wrong) {
    return Promise::reject("Oh no!")
  }
  return Promise::resolve(42);
}
```

FUNCTION COLOR AND PHP FIBERS

Synchronous

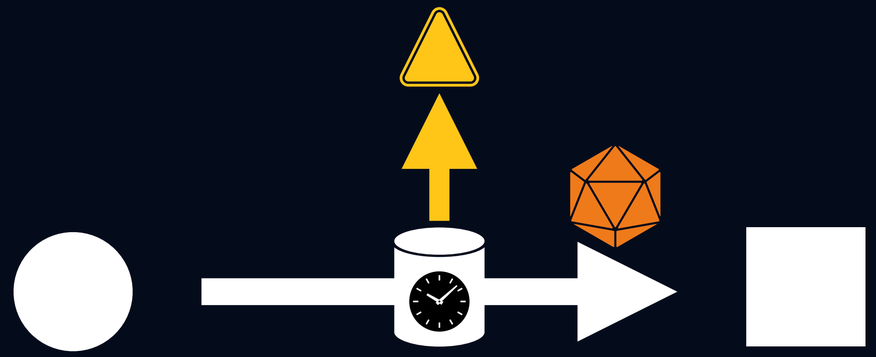


Asynchronous



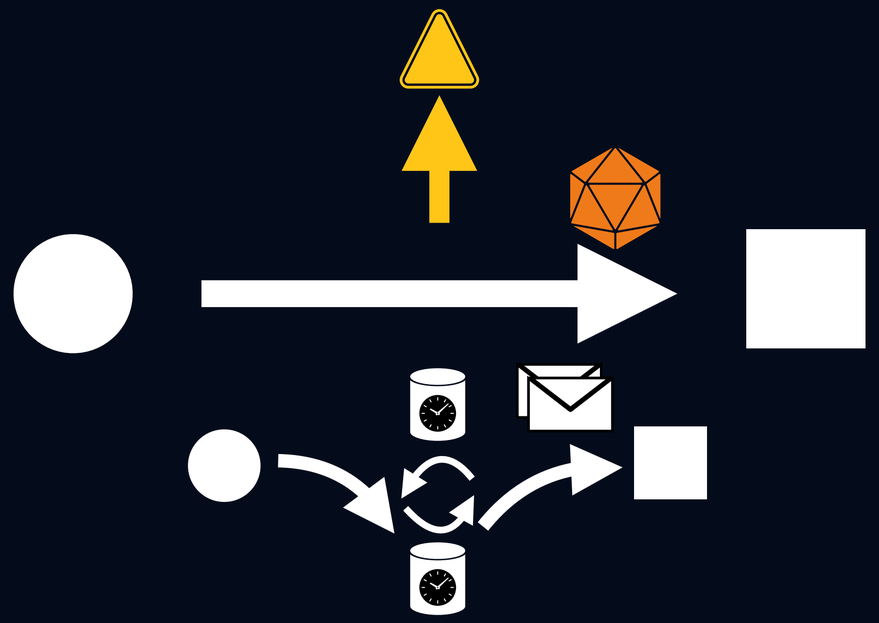
FUNCTION COLOR AND PHP FIBERS

Asynchronous with Fibers



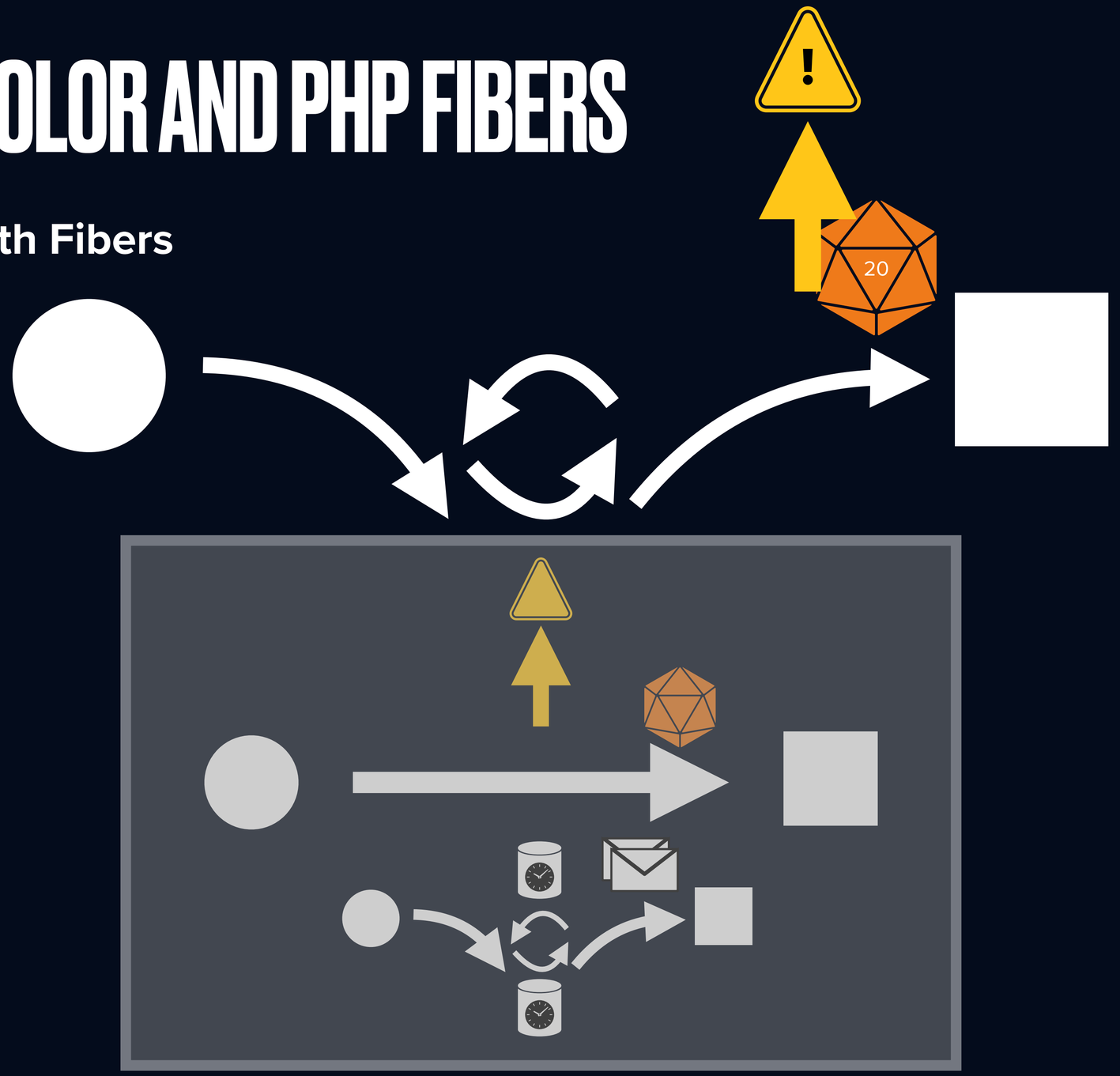
FUNCTION COLOR AND PHP FIBERS

Asynchronous with Fibers



FUNCTION COLOR AND PHP FIBERS

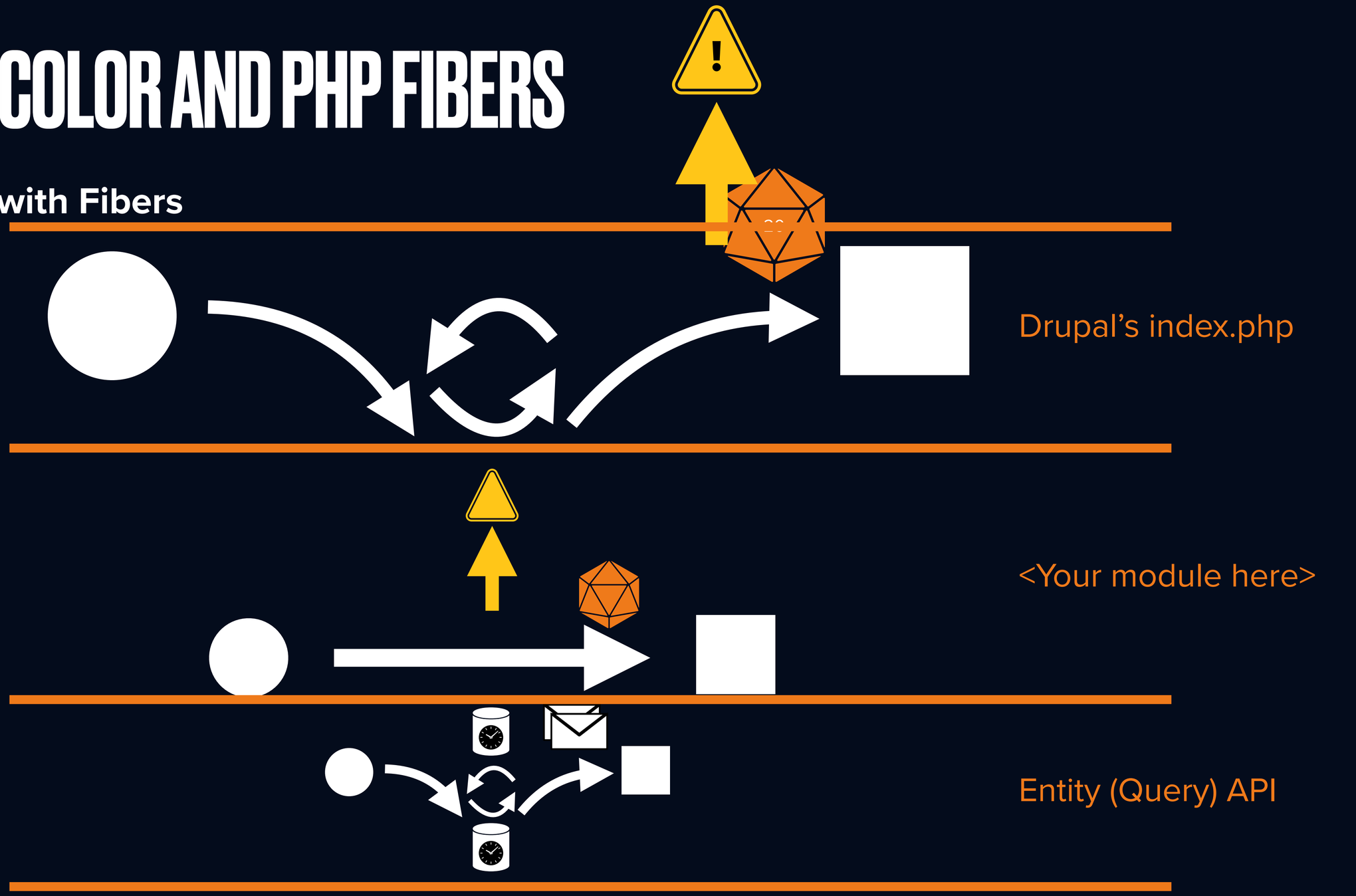
Asynchronous with Fibers



“Fibers allows building
an async sandwich
without changing code in the middle”

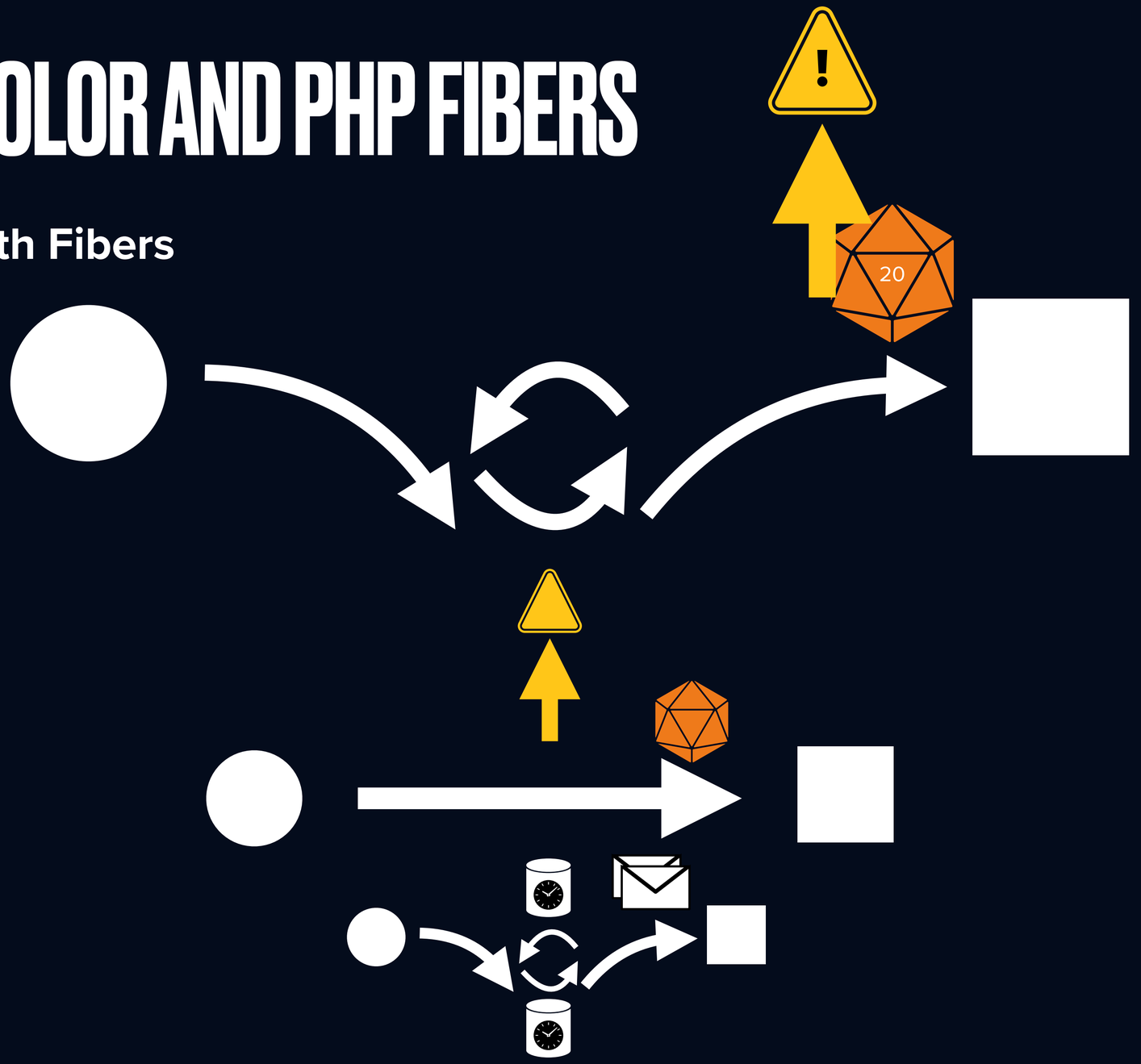
FUNCTION COLOR AND PHP FIBERS

Asynchronous with Fibers



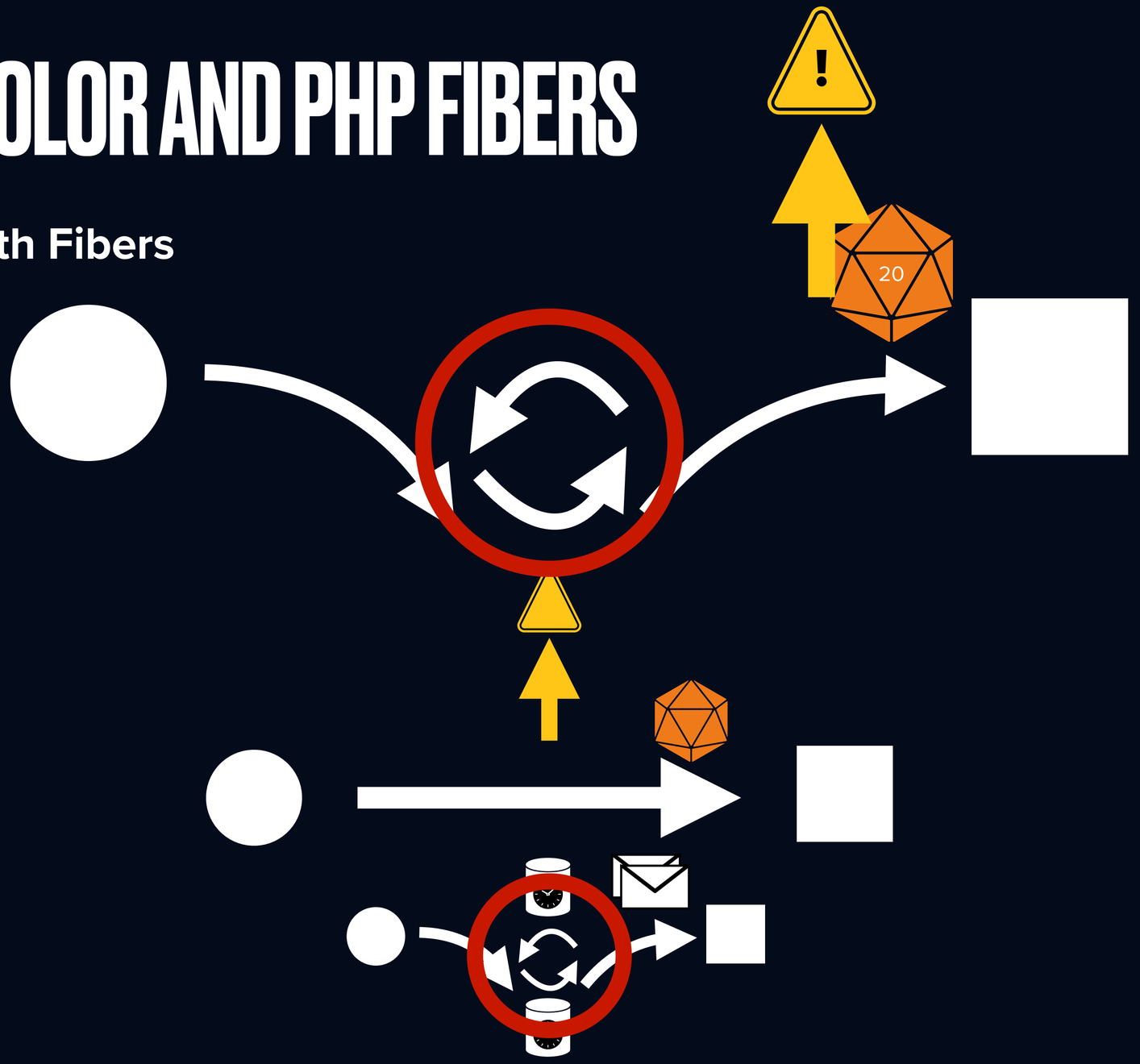
FUNCTION COLOR AND PHP FIBERS

Asynchronous with Fibers



FUNCTION COLOR AND PHP FIBERS

Asynchronous with Fibers



FUNCTION COLOR AND PHP FIBERS

```php
$fibers = [];
foreach ($elements['#attached']['placeholders'] as $placeholder => $placeholder_element) {
  // Get the render array for the given placeholder.
  $fibers[$placeholder] = new \Fiber(function () use ($placeholder_element) {
    return [$this->doRenderPlaceholder($placeholder_element), $placeholder_element];
  });
}

while (count($fibers) > 0) {
  foreach ($fibers as $placeholder => $fiber) {
    if (!$fiber->isStarted()) {
      $fiber->start();
    }
    elseif ($fiber->isSuspended()) {
      $fiber->resume();
    }
    // If the Fiber hasn't terminated by this point, move onto the next
    // placeholder, we'll resume this fiber again when we get back here.
    if (!$fiber->isTerminated()) {
      continue;
    }
    [$markup, $placeholder_element] = $fiber->getReturn();

    $elements = $this->doReplacePlaceholder($placeholder, $markup, $elements, $placeholder_element);
    unset($fibers[$placeholder]);
  }
}
```

FUNCTION COLOR AND PHP FIBERS

```php
$fibers = [];
foreach ($elements['#attached']['placeholders'] as $placeholder => $placeholder_element) {
  // Get the render array for the given placeholder.
  $fibers[$placeholder] = new \Fiber(function () use ($placeholder_element) {
    return [$this->doRenderPlaceholder($placeholder_element), $placeholder_element];
  });
}

while (count($fibers) > 0) {
  foreach ($fibers as $placeholder => $fiber) {
    if (!$fiber->isStarted()) {
      $fiber->start();
    }
    elseif ($fiber->isSuspended()) {
      $fiber->resume();
    }
    // If the Fiber hasn't terminated by this point, move onto the next
    // placeholder, we'll resume this fiber again when we get back here.
    if (!$fiber->isTerminated()) {
      continue;
    }
    [$markup, $placeholder_element] = $fiber->getReturn();

    $elements = $this->doReplacePlaceholder($placeholder, $markup, $elements, $placeholder element);
    unset($fibers[$placeholder]);
  }
}
```

Plan tasks

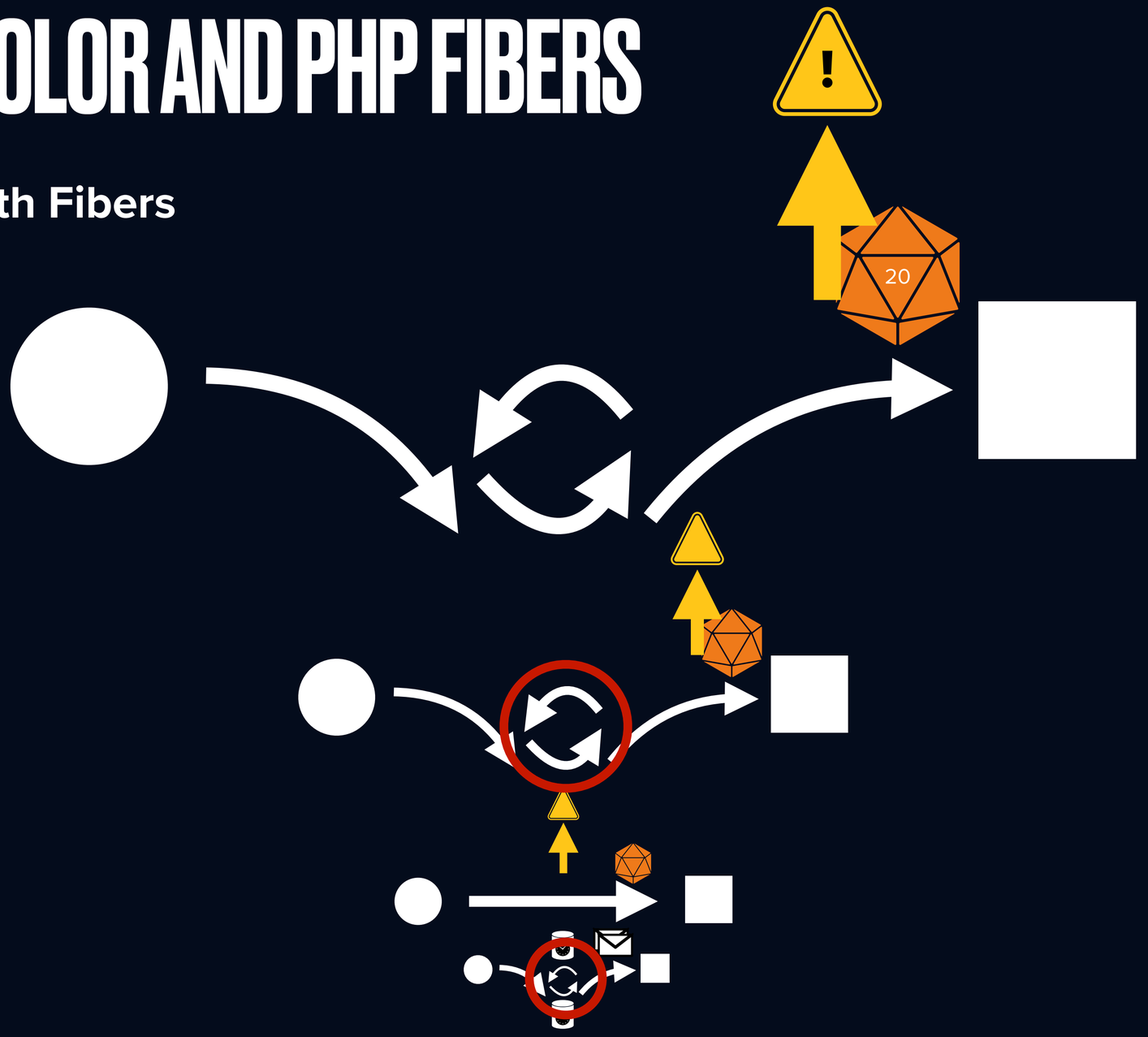
Fiber management

Use result

Fiber management

FUNCTION COLOR AND PHP FIBERS

Asynchronous with Fibers



FUNCTION COLOR AND PHP FIBERS

```php
$fibers = [];
foreach ($elements['#attached']['placeholders'] as $placeholder => $placeholder_element) {
  // Get the render array for the given placeholder.
  $fibers[$placeholder] = new \Fiber(function () use ($placeholder_element) {
    return [$this->doRenderPlaceholder($placeholder_element), $placeholder_element];
  });
}
$iterations = 0;
while (count($fibers) > 0) {
  foreach ($fibers as $placeholder => $fiber) {
    if (!$fiber->isStarted()) {
      $fiber->start();
    }
    elseif ($fiber->isSuspended()) {
      $fiber->resume();
    }
    // If the Fiber hasn't terminated by this point, move onto the next
    // placeholder, we'll resume this fiber again when we get back here.
    if (!$fiber->isTerminated()) {
      // If we've gone through the placeholders once already, and they're
      // still not finished, then start to allow code higher up the stack to
      // get on with something else.
      if ($iterations) {
        $fiber = \Fiber::getCurrent();
        if ($fiber !== NULL) {
          $fiber->suspend();
        }
      }
      continue;
    }
    [$markup, $placeholder_element] = $fiber->getReturn();

    $elements = $this->doReplacePlaceholder($placeholder, $markup, $elements, $placeholder_element);
    unset($fibers[$placeholder]);
  }
  $iterations++;
}
```

Plan tasks

Fiber management

Use result

Fiber management

FUNCTION COLOR AND PHP FIBERS

```php
$fibers = [];
foreach ($elements['#attached']['placeholders'] as $placeholder => $placeholder_element) {
  // Get the render array for the given placeholder.
  $fibers[$placeholder] = new \Fiber(function () use ($placeholder_element) {
    return [$this->doRenderPlaceholder($placeholder_element), $placeholder_element];
  });
}
$iterations = 0;
while (count($fibers) > 0) {
  foreach ($fibers as $placeholder => $fiber) {
    if (!$fiber->isStarted()) {
      $fiber->start();
    }
    elseif ($fiber->isSuspended()) {
      $fiber->resume();
    }
    // If the Fiber hasn't terminated by this point, move onto the next
    // placeholder, we'll resume this fiber again when we get back here.
    if (!$fiber->isTerminated()) {
      // If we've gone through the placeholders once already, and they're
      // still not finished, then start to allow code higher up the stack to
      // get on with something else.
      if ($iterations) {
        $fiber = \Fiber::getCurrent();
        if ($fiber !== NULL) {
          $fiber->suspend();
        }
      }
      continue;
    }
    [$markup, $placeholder_element] = $fiber->getReturn();

    $elements = $this->doReplacePlaceholder($placeholder, $markup, $elements, $placeholder_element);
    unset($fibers[$placeholder]);
  }
  $iterations++;
}
```

Plan tasks

Fiber management

Delegation

Use result

Fiber management

FUNCTION COLOR AND PHP FIBERS

More async fun

- Races - Use the fastest result
  e.g. cache lookup vs fresh calculation
- Timeouts - Ensure tasks don’t run forever
  e.g. an external web request may take at most 5 seconds
- Repeats - Execute a task every N seconds
  e.g. a keep-alive ping on a websocket connection
- Streams - Do nothing until data comes in on a stream
  e.g. perform background tasks until there’s an HTTP request

FUNCTION COLOR AND PHP FIBERS

Racing tasks

- Use the fastest result
  e.g. cache lookup vs fresh calculation

FUNCTION COLOR AND PHP FIBERS

Racing tasks

- Use the fastest result
  e.g. cache lookup vs fresh calculation

```php
function await_first(iterable $fibers): mixed {
  foreach ($fibers as $fiber) {
    if (!$fiber->isStarted()) {
      $fiber->start();
    }
  }

  while (true) {
    foreach ($fibers as $fiber) {
      if ($fiber->isSuspended()) {
        $fiber->resume();
      }
      if ($fiber->isTerminated()) {
        return $fiber->getReturn();
      }
    }

    // Allow other fibers or processes to progress.
    if (Fiber::getCurrent() !== null) {
      // Inside a Fiber: yield control to the scheduler.
      Fiber::suspend();
    } else {
      // Not in a Fiber (main thread): avoid a hot loop.
      usleep(100_000); // 100ms
    }
  }
}

$tasks = [
  new Fiber(async_task(name: "Task A", delay: 5)),
  new Fiber(async_task(name: "Task B", delay: 2)),
  new Fiber(async_task(name: "Task C", delay: 3)),
];

$result = await_first($tasks);

echo "First result: $result\n";
```

FUNCTION COLOR AND PHP FIBERS

Racing tasks

- Use the fastest result
  e.g. cache lookup vs fresh calculation

Simplified example

- No clean-up/cancellation of pending tasks
  e.g. abort network request to reduce load
- No error handling fallback
  If the initial task fails (e.g. network exception) then entire execution fails
- No timeout
  If all tasks take long then there's no simple way to cancel.

```php
function await_first(iterable $fibers): mixed {
  foreach ($fibers as $fiber) {
    if (!$fiber->isStarted()) {
      $fiber->start();
    }
  }

  while (true) {
    foreach ($fibers as $fiber) {
      if ($fiber->isSuspended()) {
        $fiber->resume();
      }
      if ($fiber->isTerminated()) {
        return $fiber->getReturn();
      }
    }

    // Allow other fibers or processes to progress.
    if (Fiber::getCurrent() !== null) {
      // Inside a Fiber: yield control to the scheduler.
      Fiber::suspend();
    } else {
      // Not in a Fiber (main thread): avoid a hot loop.
      usleep(100_000); // 100ms
    }
  }
}

$tasks = [
  new Fiber(async_task(name: "Task A", delay: 5)),
  new Fiber(async_task(name: "Task B", delay: 2)),
  new Fiber(async_task(name: "Task C", delay: 3)),
];

$result = await_first($tasks);

echo "First result: $result\n";
```

FUNCTION COLOR AND PHP FIBERS

Intermezzo

- Promises help manage results of async tasks non-disruptively
  “I promise to give you a result in the future”
- Promises change the return types of your functions which cascades throughout your application
- Fibers allows us to scope async code to limit the impact of changes needed to adopt async
- Scoping async changes allows gradually benefiting from async performance improvements within an existing codebase like Drupal
- Two parts need to be async:
  - The bottom level task needs to indicate when other work can happen
  - The top level task needs to kick-off and wait for multiple tasks at the same time

FUNCTION COLOR AND PHP FIBERS

Problems

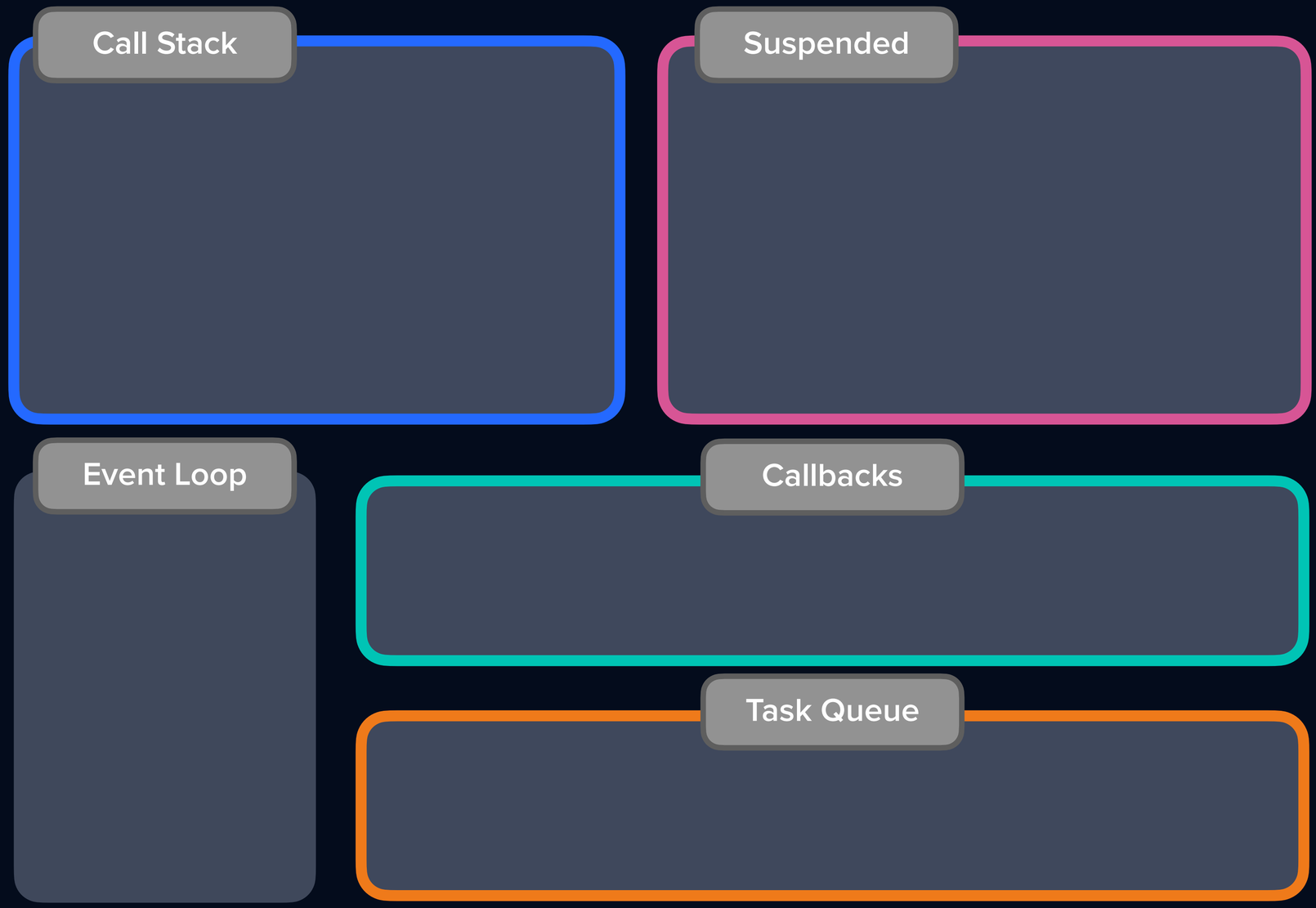
- Using Fibers correctly to avoid accidentally writing synchronous/blocking code is HARD
- To make things play nicely we need to implement these co-operative loops everywhere
- Orchestrating Fibers for different asynchronous patterns is HARDER

- What if rather than many loops, we built a single loop?

TASK SCHEDULING WITH THE EVENT LOOP

TASK SCHEDULING WITH THE EVENT LOOP

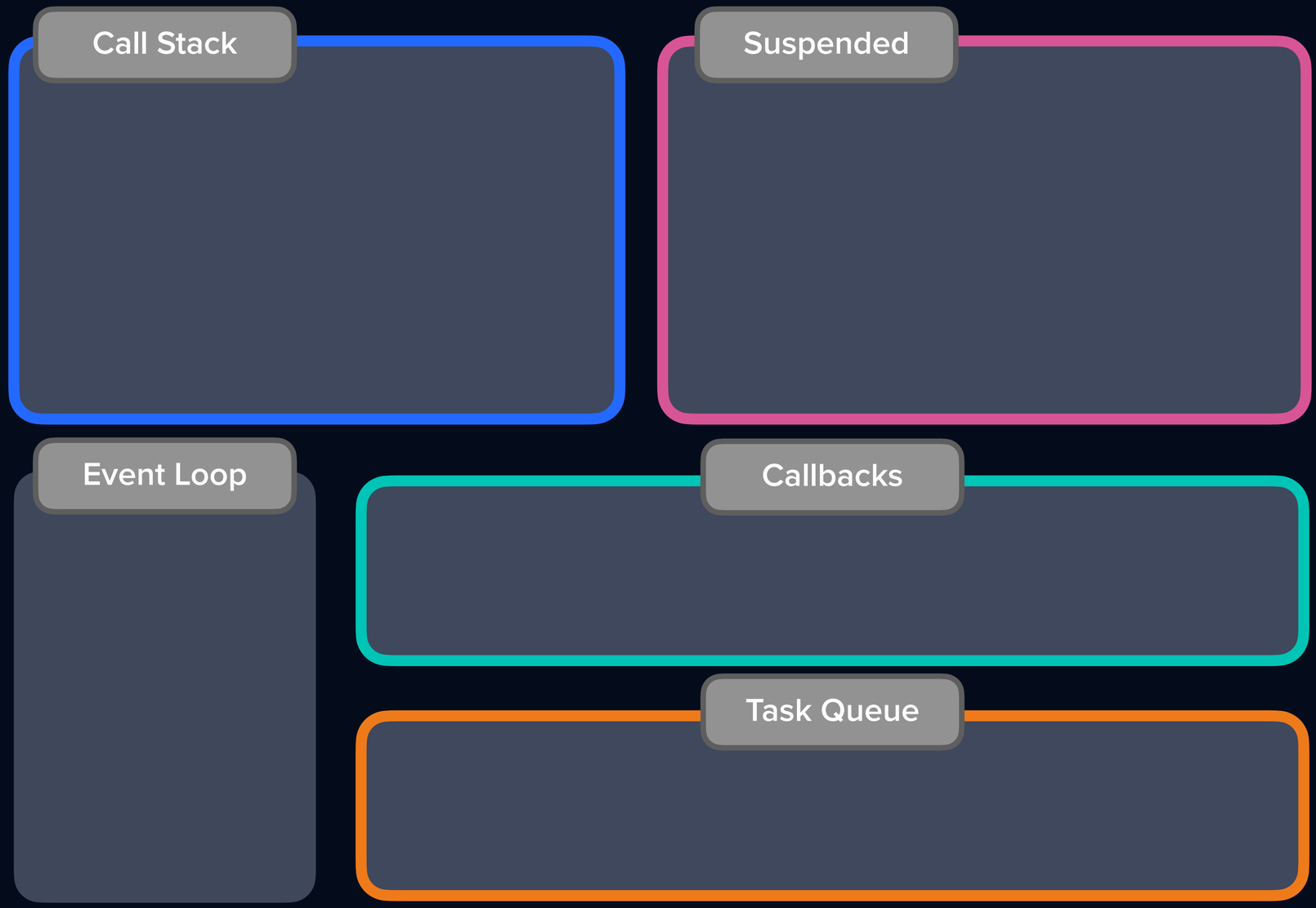
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

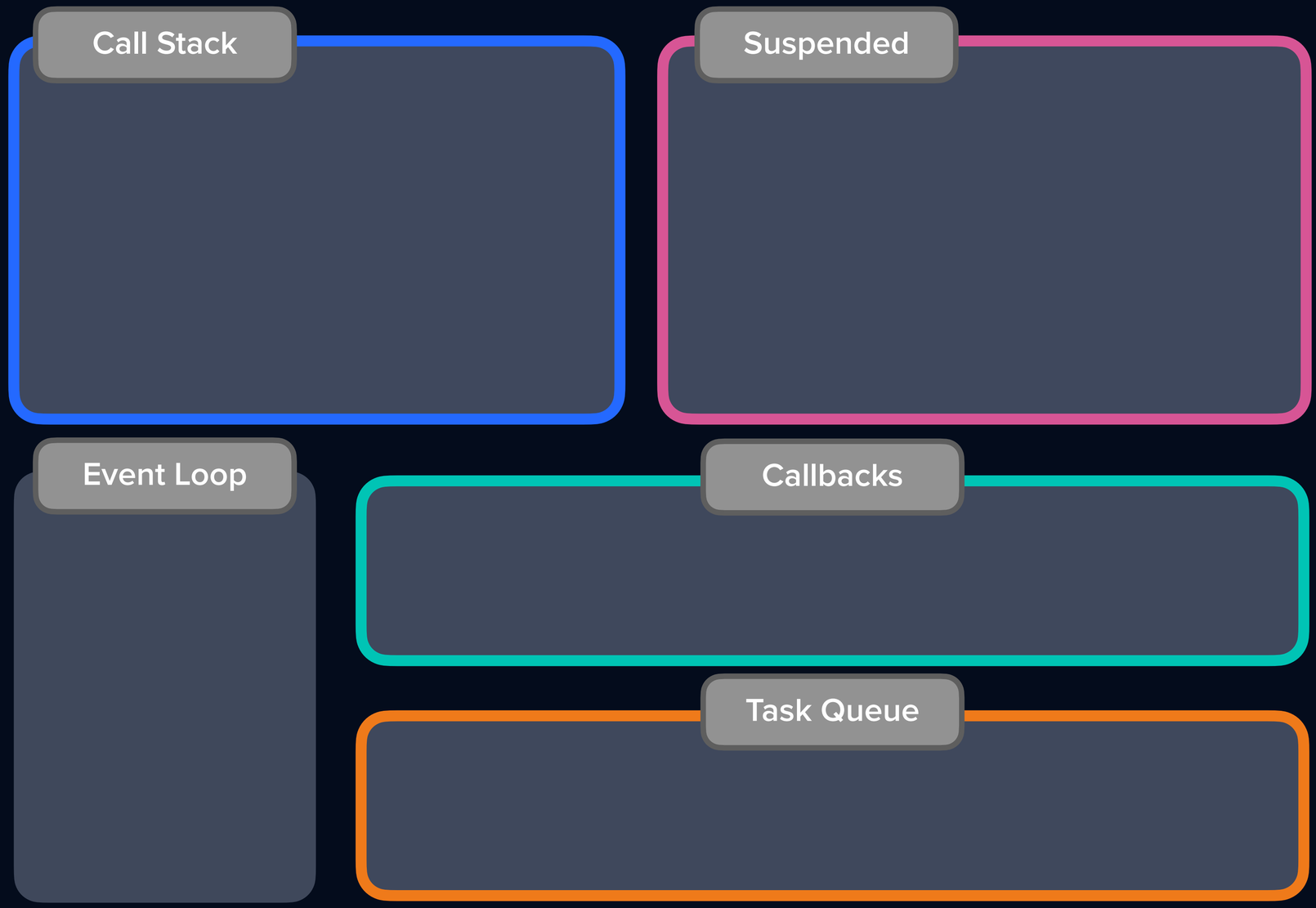
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

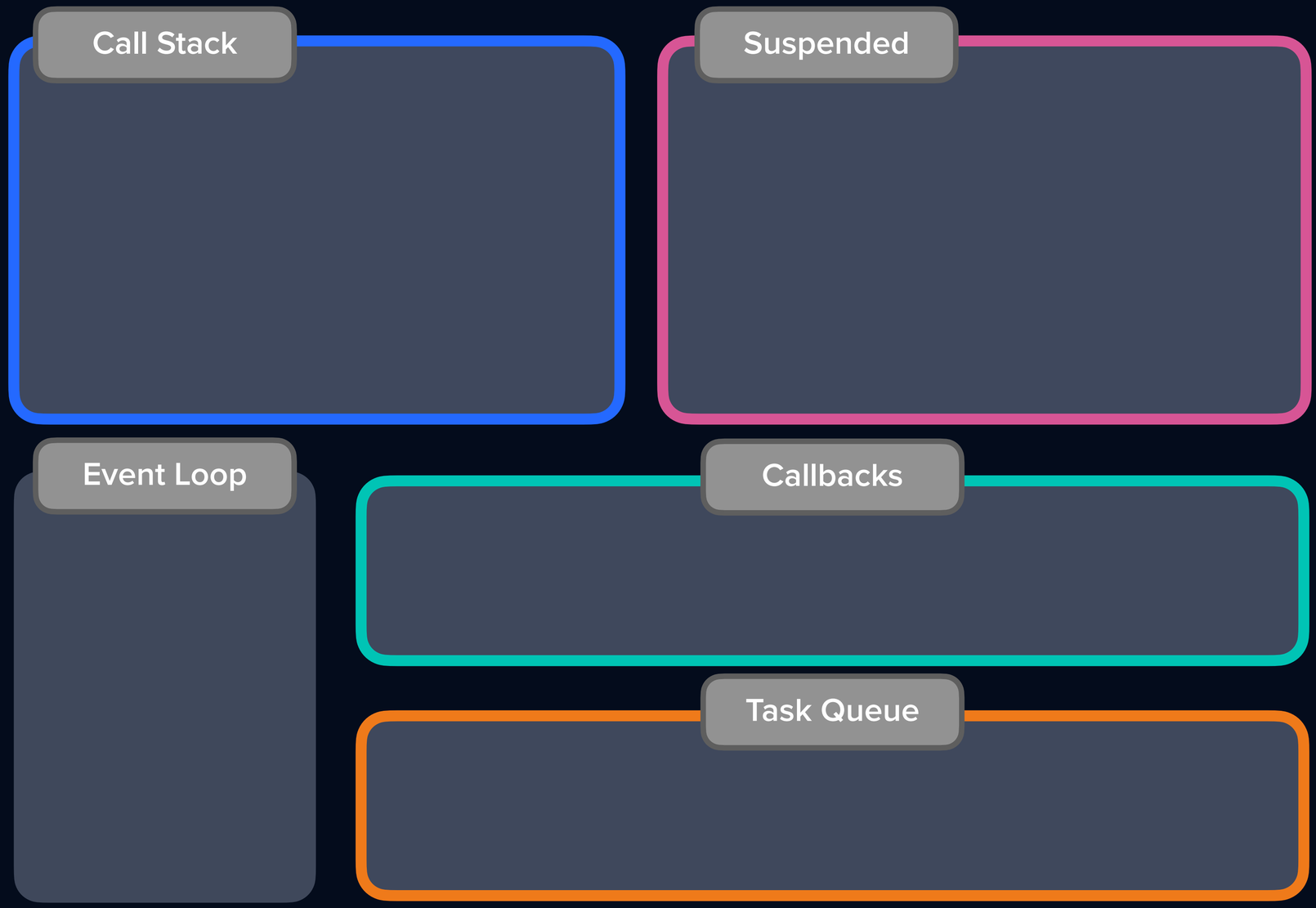
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

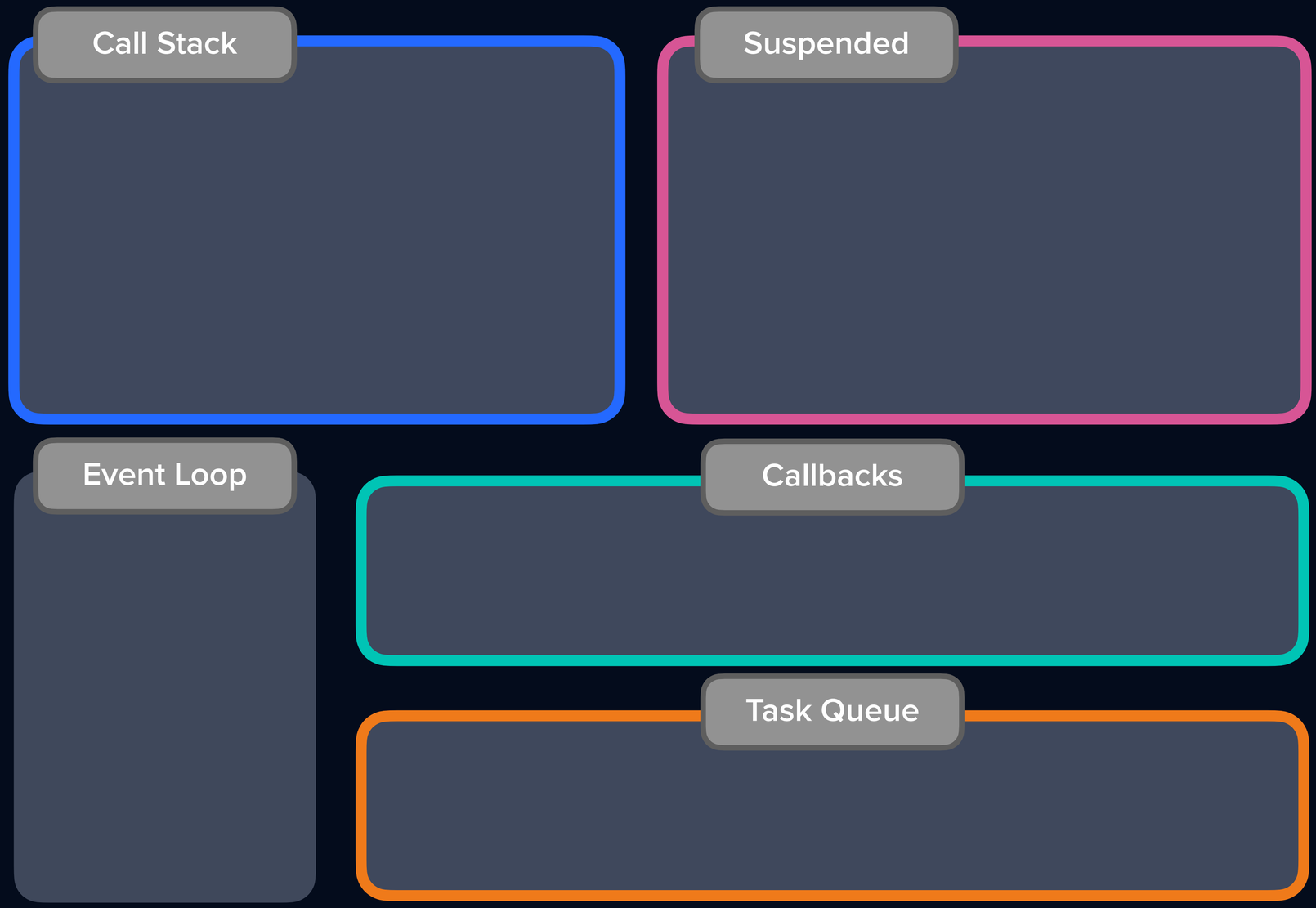
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

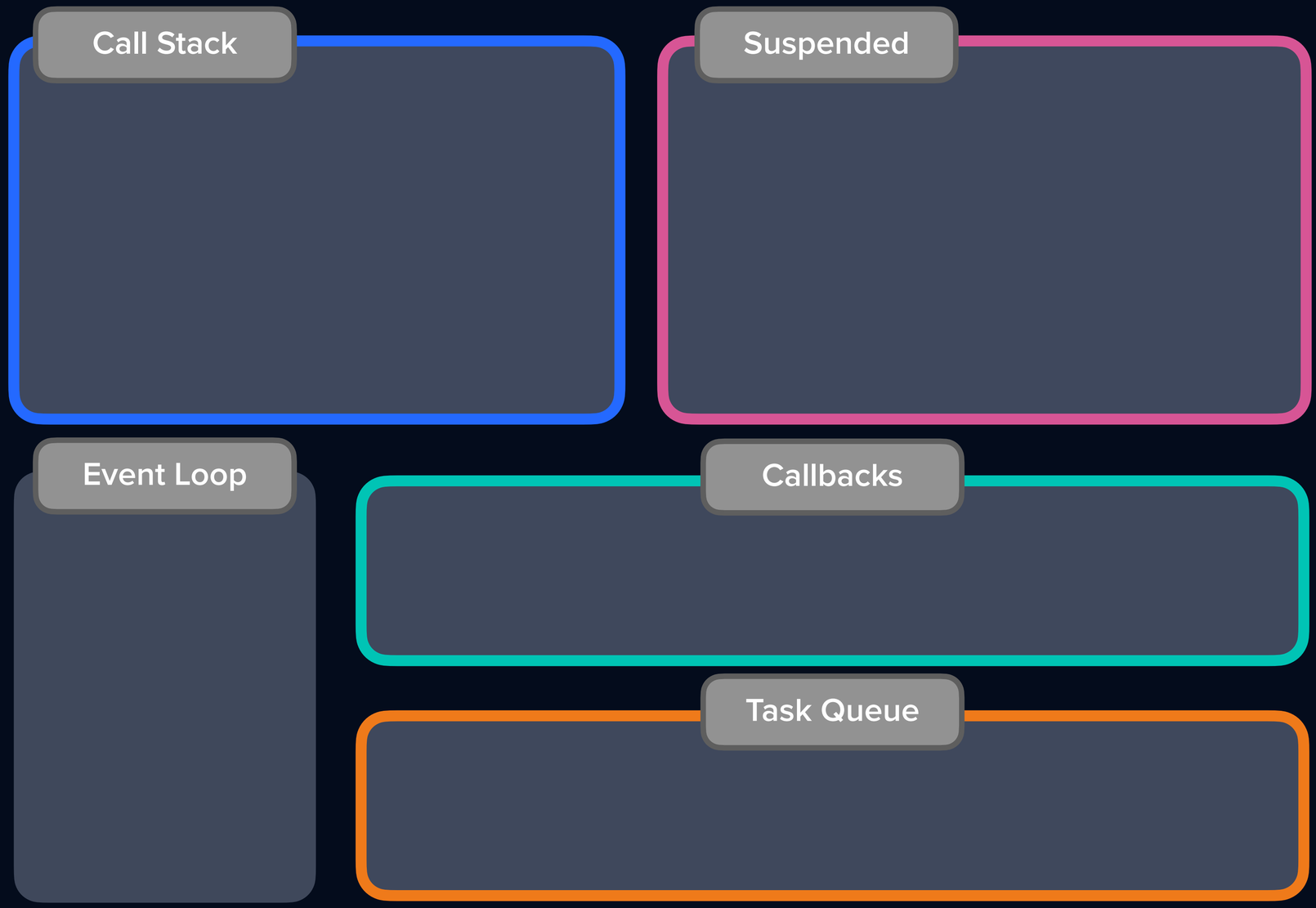
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

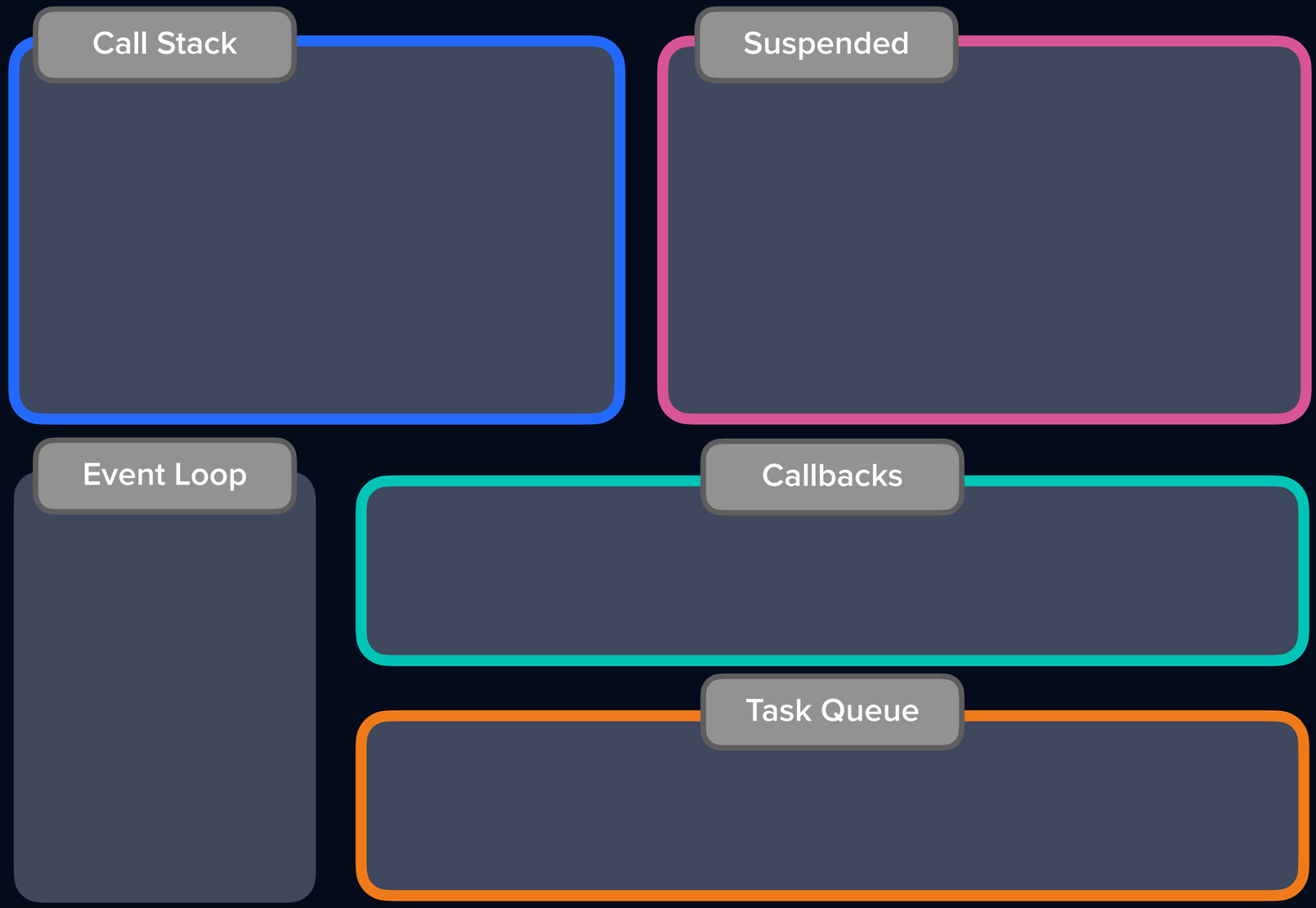
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

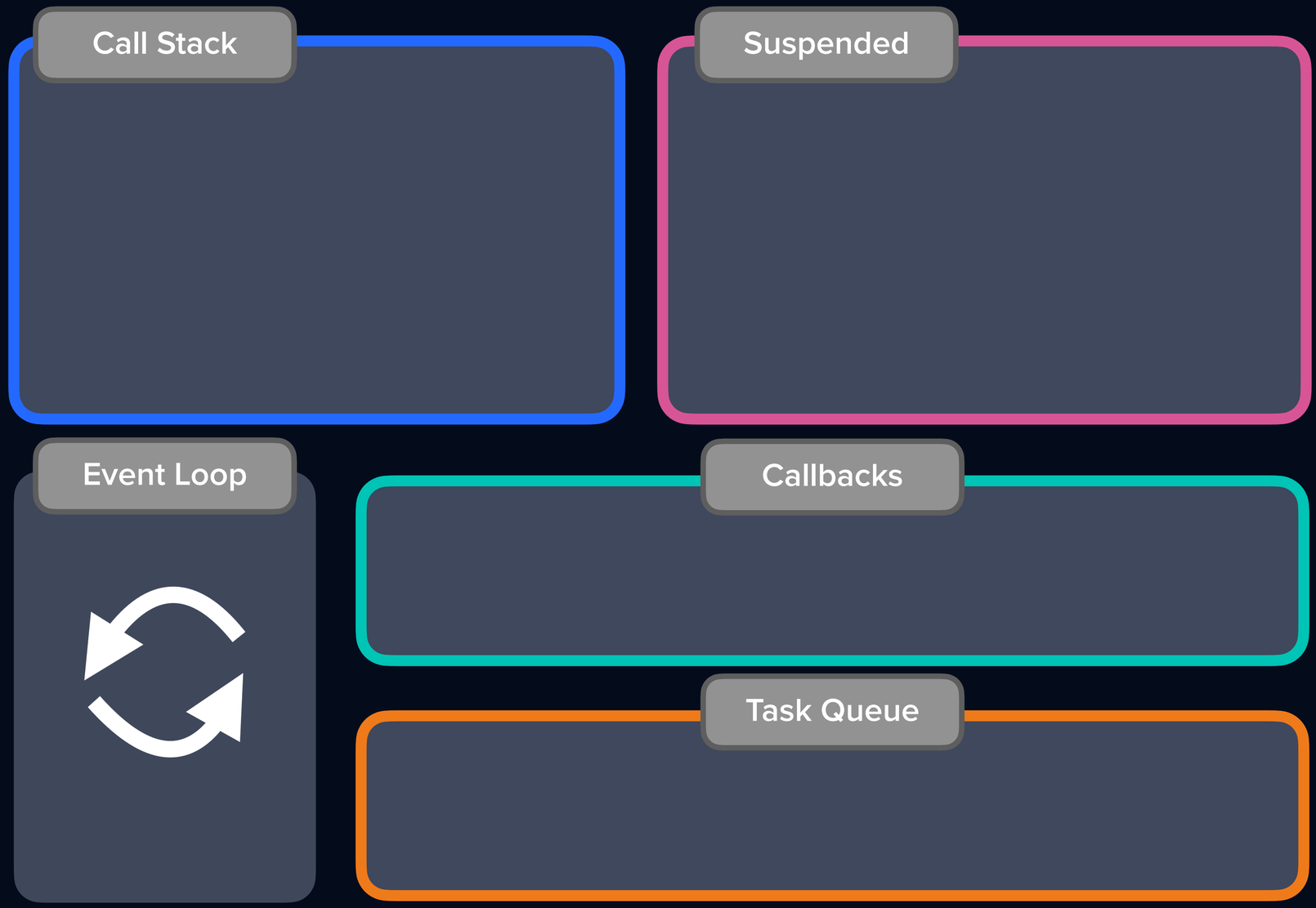
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

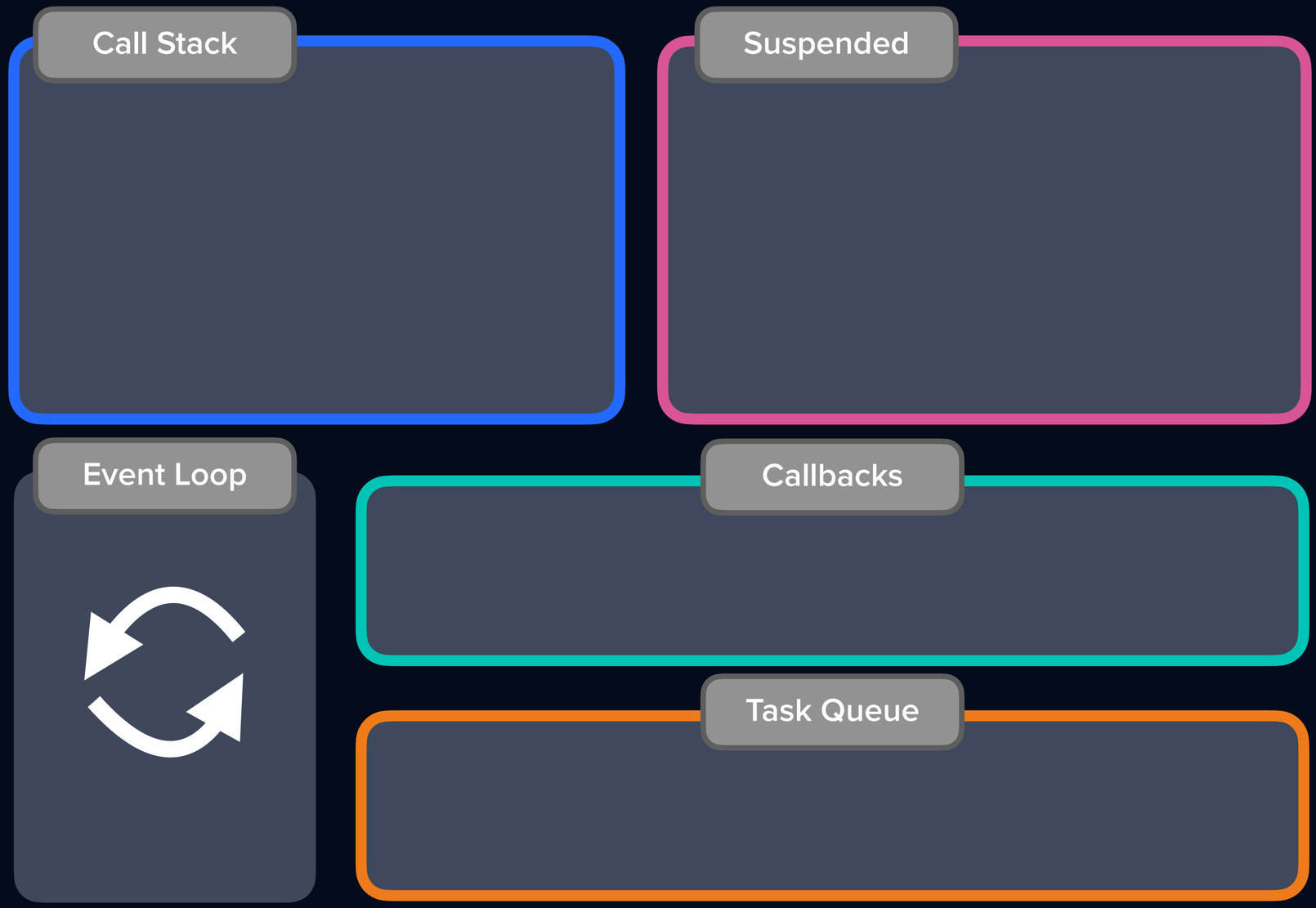
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

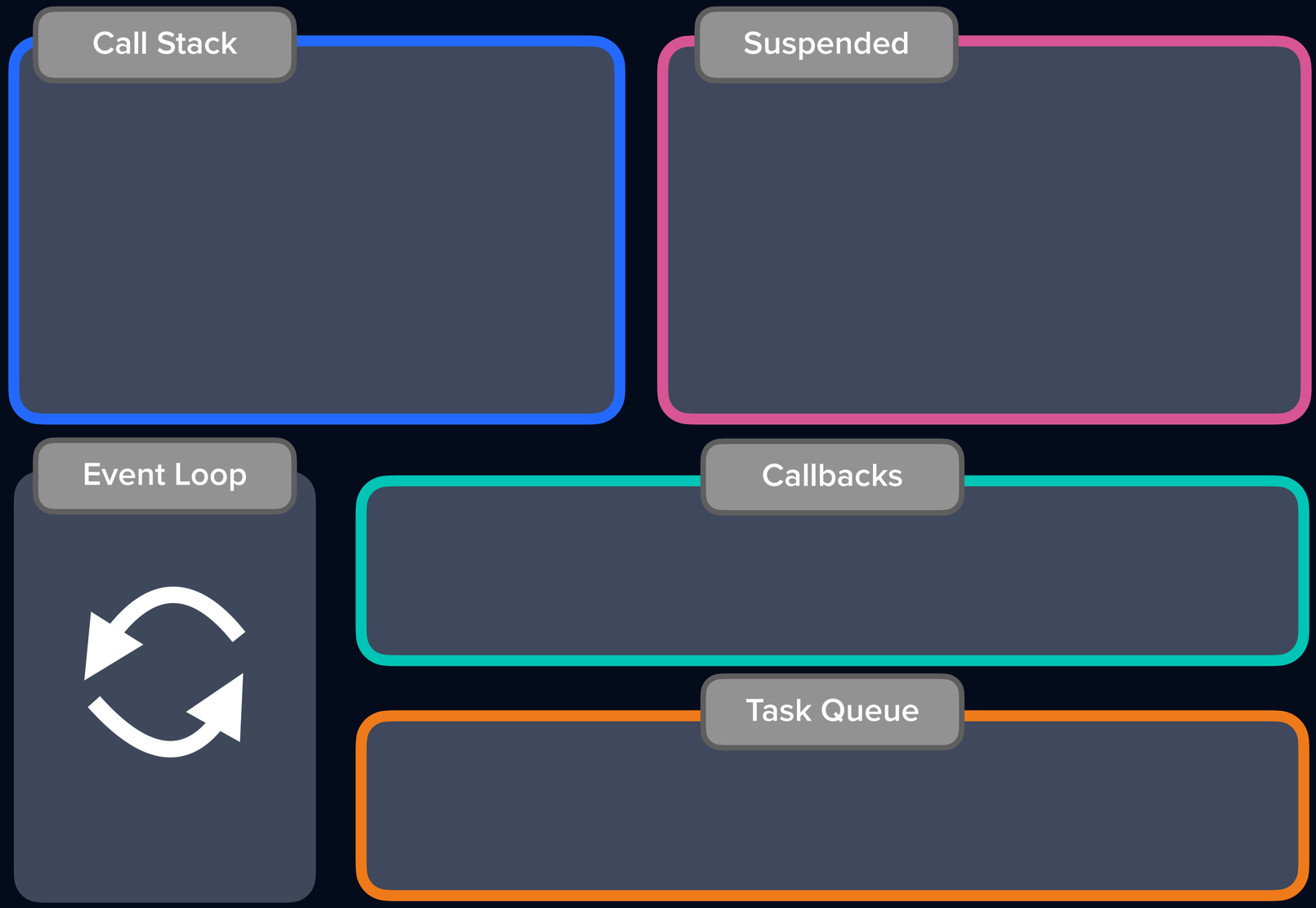
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

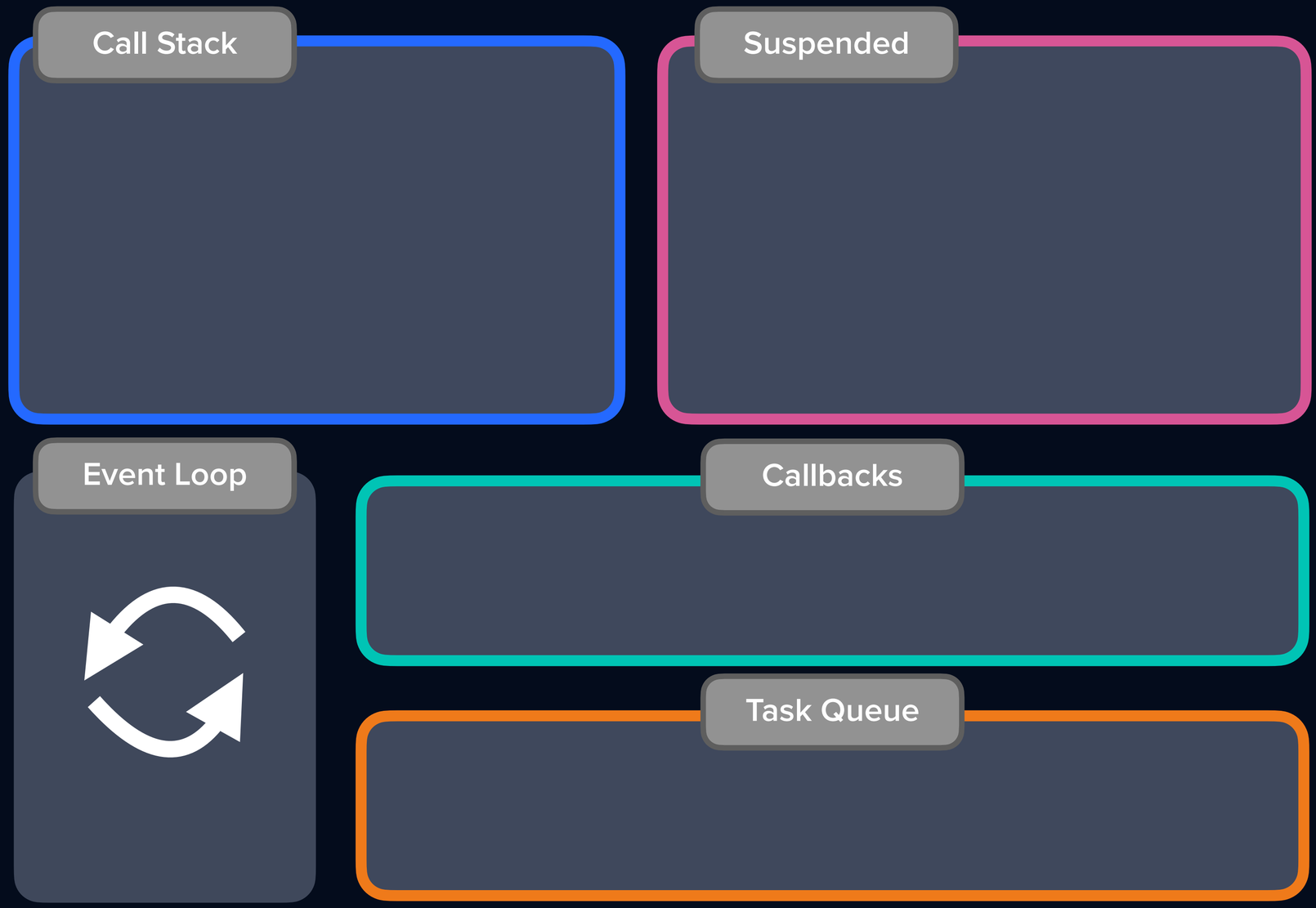
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

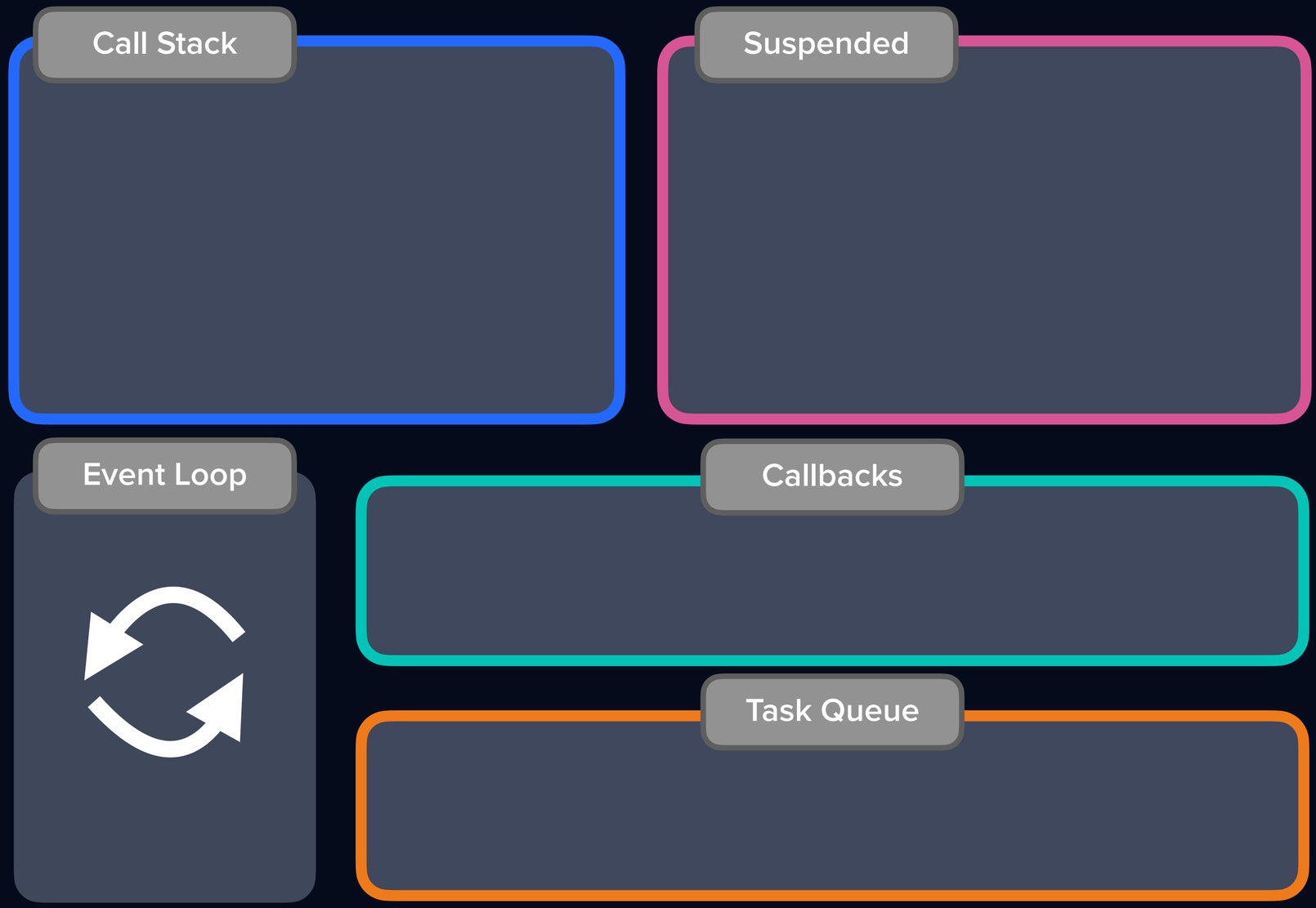
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

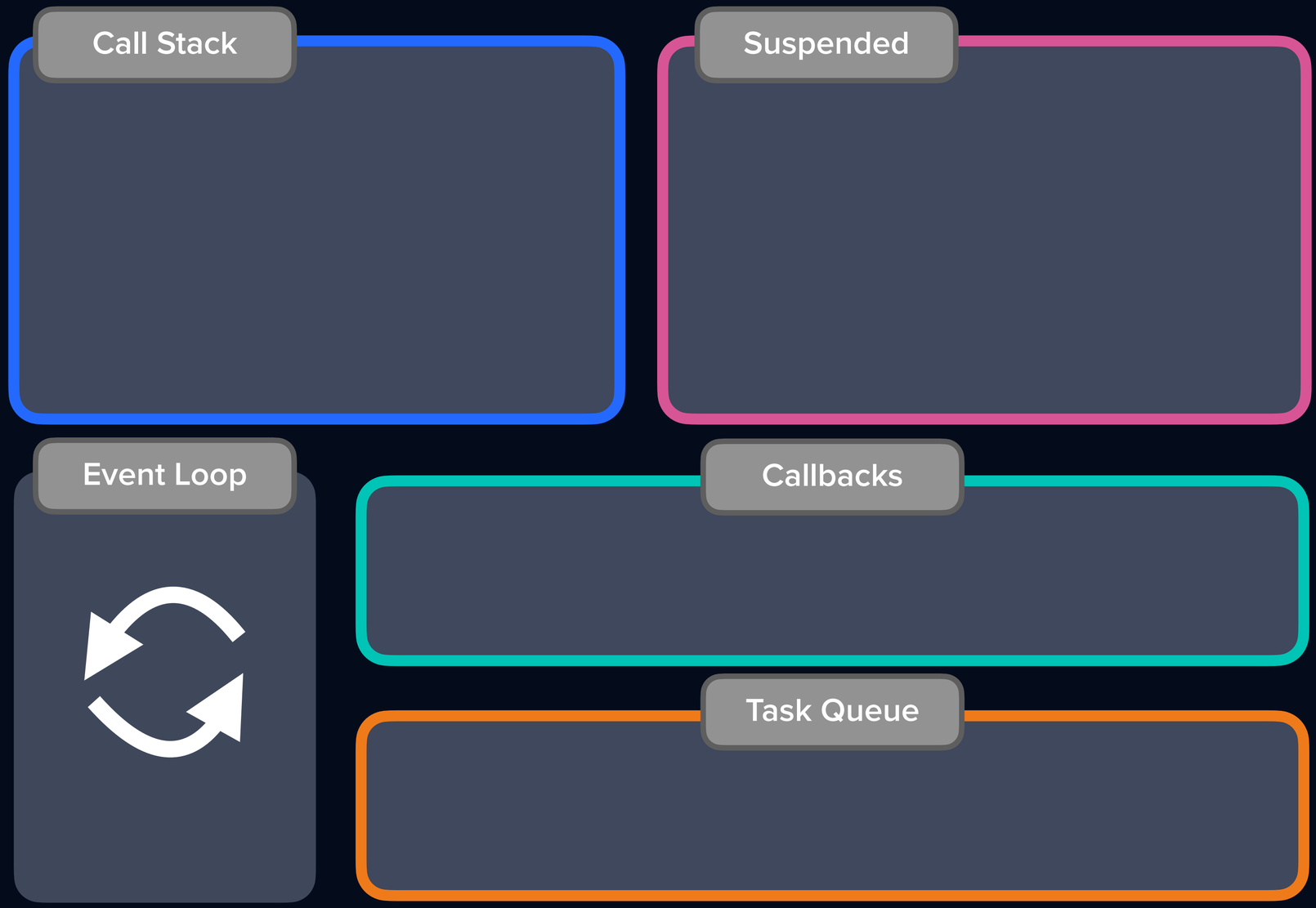
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

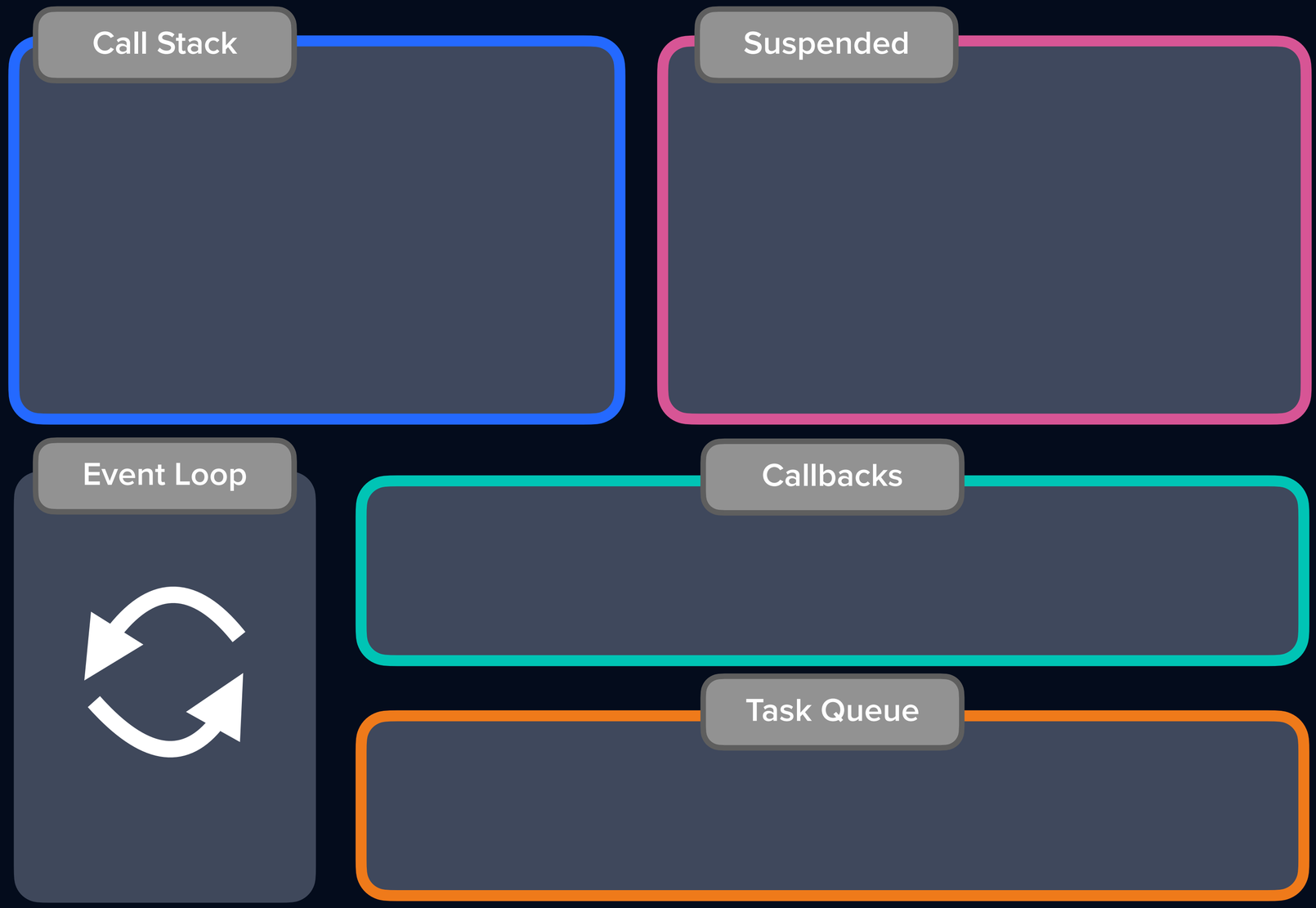
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

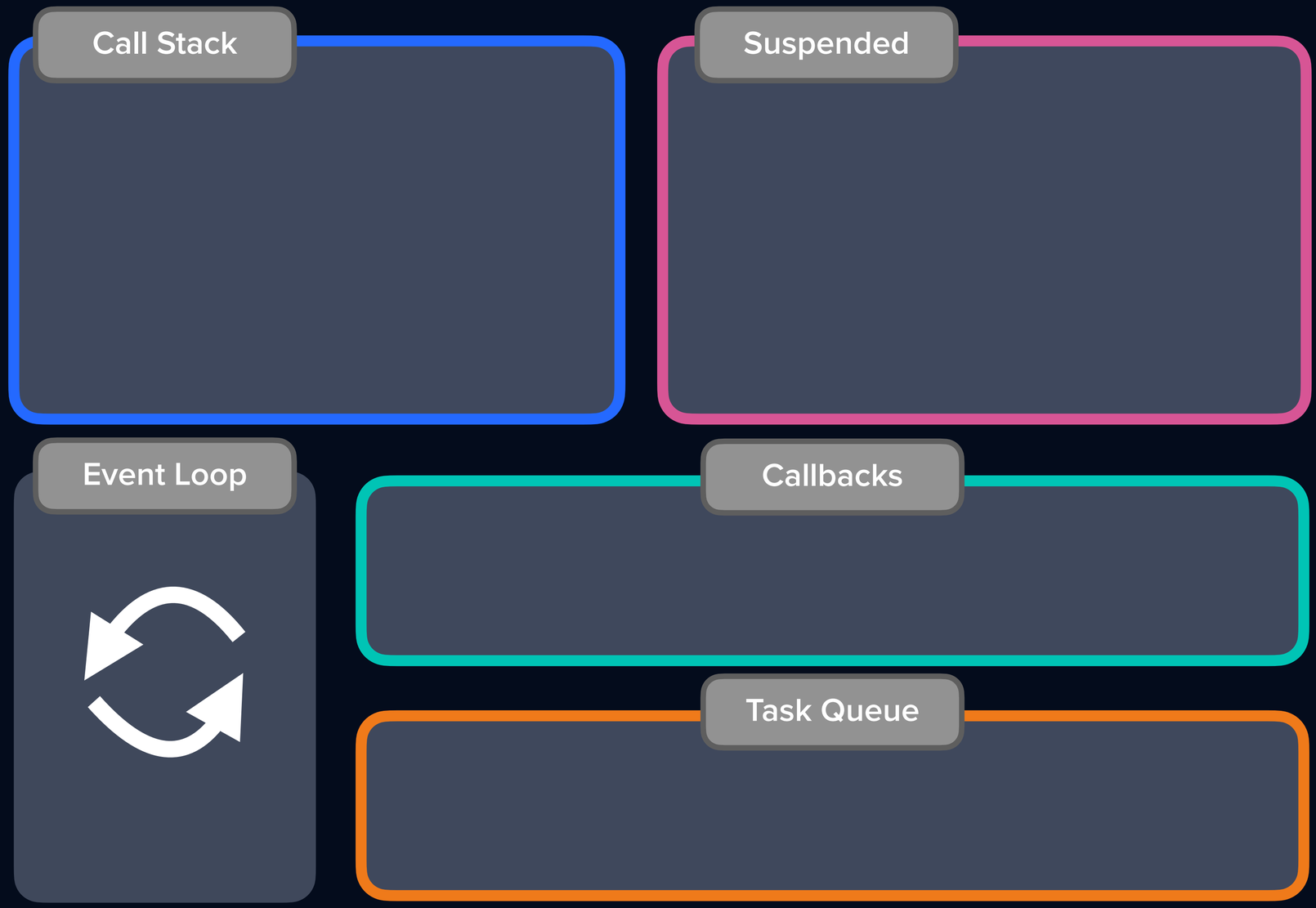
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

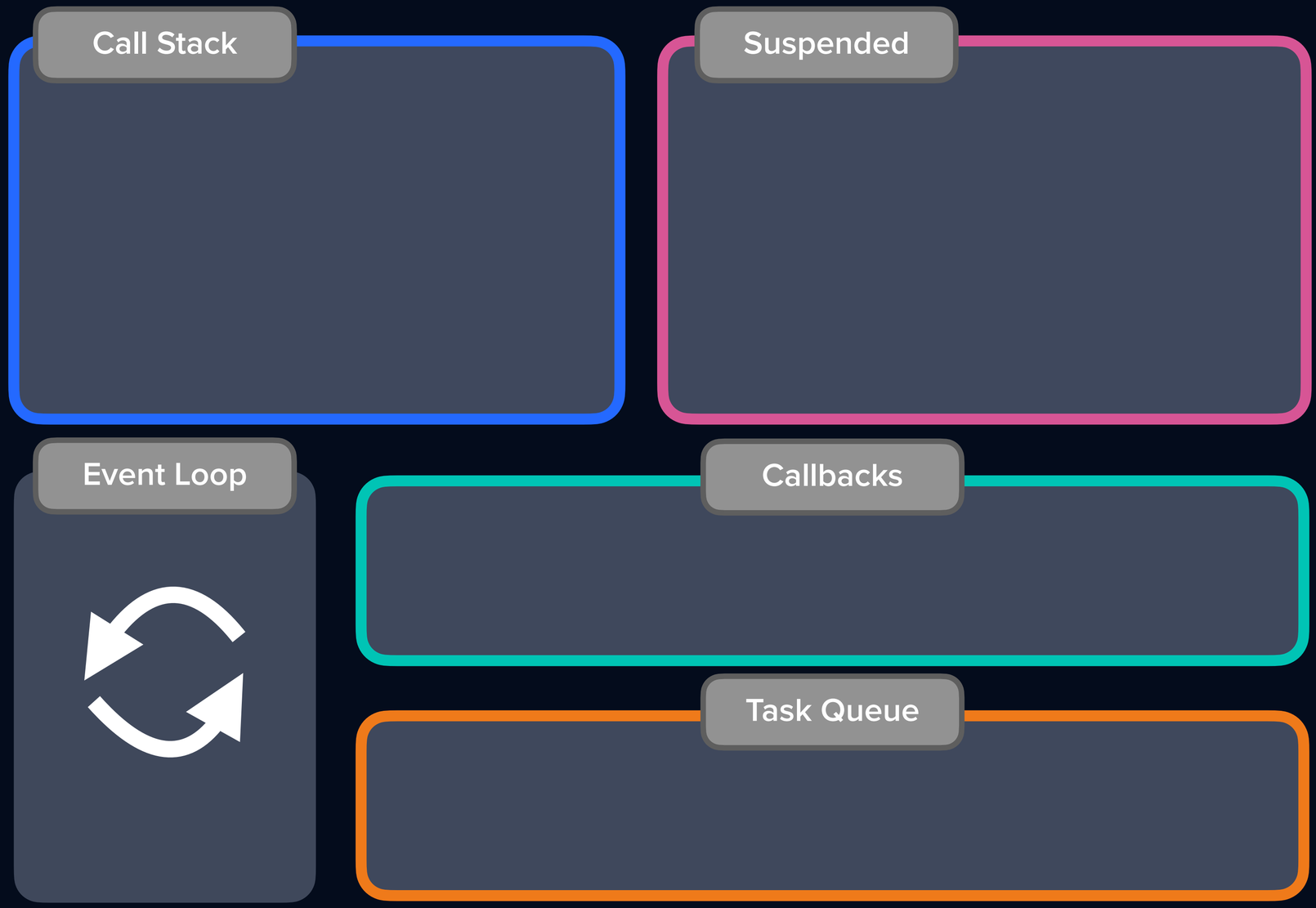
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

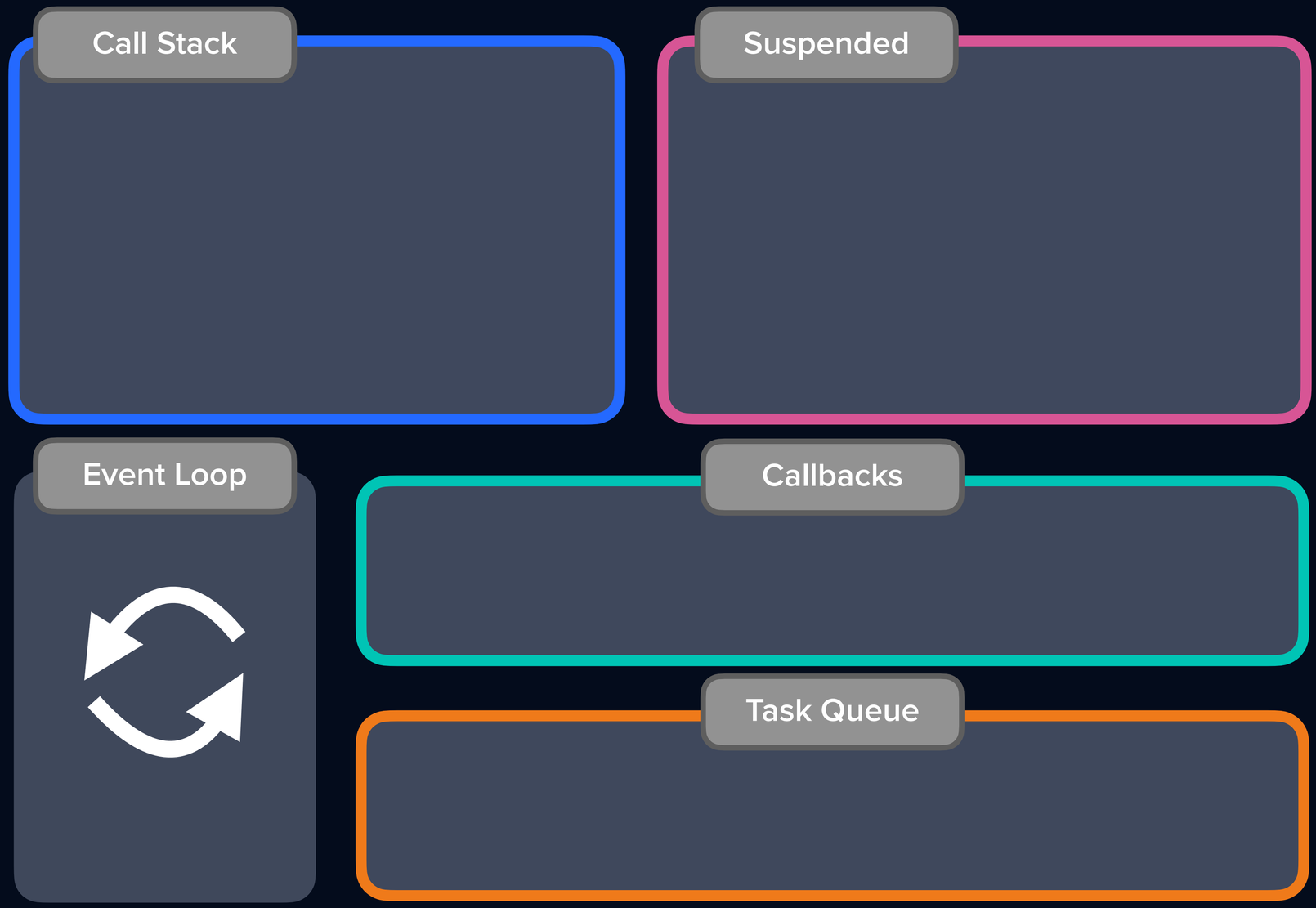
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

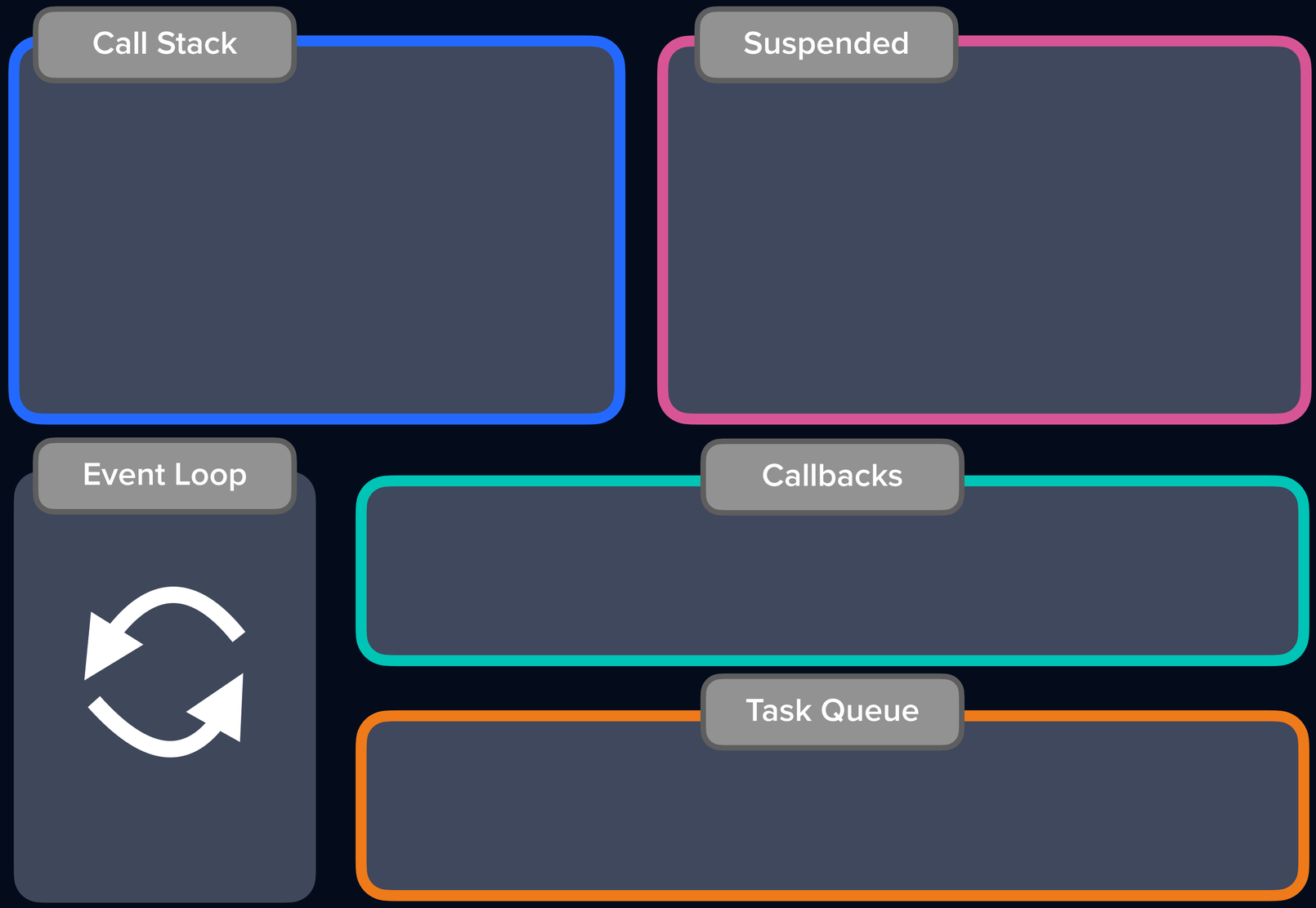
A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

A single loop



Visualisation inspired by “JavaScript Visualised - Event Loop, Web APIs, (Micro)task Queue” by Lydia Hallie
https://www.youtube.com/watch?v=eiC58R16hb8

TASK SCHEDULING WITH THE EVENT LOOP

A single loop

- Has the global overview of priorities and can schedule accordingly
- Allows downstream code to focus on describing async workloads
- Reduces the risk of forgetting to relinquish control to a higher level loop
- The loop is started after set-up is done and runs until program end

ONE LOOP TO RULE THEM ALL

REVOLT

- One event loop to rule them all
- Ensures that libraries all know how to schedule their task
- Adopted by Drupal in https://www.drupal.org/i/3425114

- Ensures we stay “off the island”
- Works with libraries such as AMPHP and ReactPHP

REVOLT EXAMPLE

```php
use Revolt\EventLoop;

$suspension = EventLoop::getSuspension();

$repeatId = EventLoop::repeat(1, function (): void {
  print '++ Executing callback created by EventLoop::repeat()' . PHP_EOL;
});

EventLoop::delay(5, function () use ($suspension, $repeatId): void {
  print '++ Executing callback created by EventLoop::delay()' . PHP_EOL;

  EventLoop::cancel($repeatId);
  $suspension->resume(null);

  print '++ Suspension::resume() is async!' . PHP_EOL;
});

print '++ Suspending to event loop...' . PHP_EOL;

$suspension->suspend();

print '++ Script end' . PHP_EOL;
```

```
++ Suspending to event loop...
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::delay()
++ Suspension::resume() is async!
++ Script end
```

REVOLT EXAMPLE

```php
use Revolt\EventLoop;

$suspension = EventLoop::getSuspension();

$repeatId = EventLoop::repeat(1, function (): void {
  print '++ Executing callback created by EventLoop::repeat()' . PHP_EOL;
});

EventLoop::delay(5, function () use ($suspension, $repeatId): void {
  print '++ Executing callback created by EventLoop::delay()' . PHP_EOL;

  EventLoop::cancel($repeatId);
  $suspension->resume(null);

  print '++ Suspension::resume() is async!' . PHP_EOL;
});

print '++ Suspending to event loop...' . PHP_EOL;

$suspension->suspend();

print '++ Script end' . PHP_EOL;
```

```
++ Suspending to event loop...
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::delay()
++ Suspension::resume() is async!
++ Script end
```

REVOLT EXAMPLE

```php
use Revolt\EventLoop;

$suspension = EventLoop::getSuspension();

$repeatId = EventLoop::repeat(1, function (): void {
  print '++ Executing callback created by EventLoop::repeat()' . PHP_EOL;
});

EventLoop::delay(5, function () use ($suspension, $repeatId): void {
  print '++ Executing callback created by EventLoop::delay()' . PHP_EOL;

  EventLoop::cancel($repeatId);
  $suspension->resume(null);

  print '++ Suspension::resume() is async!' . PHP_EOL;
});

print '++ Suspending to event loop...' . PHP_EOL;

$suspension->suspend();

print '++ Script end' . PHP_EOL;
```

```
++ Suspending to event loop...
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::delay()
++ Suspension::resume() is async!
++ Script end
```

REVOLT EXAMPLE

```php
use Revolt\EventLoop;

$suspension = EventLoop::getSuspension();

$repeatId = EventLoop::repeat(1, function (): void {
  print '++ Executing callback created by EventLoop::repeat()' . PHP_EOL;
});

EventLoop::delay(5, function () use ($suspension, $repeatId): void {
  print '++ Executing callback created by EventLoop::delay()' . PHP_EOL;

  EventLoop::cancel($repeatId);
  $suspension->resume(null);

  print '++ Suspension::resume() is async!' . PHP_EOL;
});

print '++ Suspending to event loop...' . PHP_EOL;

$suspension->suspend();

print '++ Script end' . PHP_EOL;
```

```
++ Suspending to event loop...
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::repeat()
++ Executing callback created by EventLoop::delay()
++ Suspension::resume() is async!
++ Script end
```

REVOLT PRIMITIVES

Defer - The callback is executed in the next iteration of the event loop. If there are defers scheduled, the event loop won't wait between iterations.

```php
use Revolt\EventLoop;

echo "line 1\n";

EventLoop::defer(function (): void {
  echo "line 3\n";
});

echo "line 2\n";

EventLoop::run();
```

https://revolt.run/timers

REVOLT PRIMITIVES

Defer - The callback is executed in the next iteration of the event loop. If there are defers scheduled, the event loop won't wait between iterations.

```php
use Revolt\EventLoop;

echo "line 1\n";

EventLoop::defer(function (): void {
  echo "line 3\n";
});

echo "line 2\n";

EventLoop::run();
```

line 1
line 2
line 3

https://revolt.run/timers

REVOLT PRIMITIVES

Delay - The callback is executed after the specified number of seconds. Fractions of a second may be expressed as floating point numbers.

```php
use Revolt\EventLoop;

date_default_timezone_set('Europe/Vienna');

EventLoop::delay(3, function (): void {
  echo "Delayed: " . date('Y-m-d h:i:s') . PHP_EOL;
});

echo "Pre-run: " . date('Y-m-d h:i:s') . PHP_EOL;

EventLoop::run();
```

https://revolt.run/timers

REVOLT PRIMITIVES

Delay - The callback is executed after the specified number of seconds. Fractions of a second may be expressed as floating point numbers.

```php
use Revolt\EventLoop;

date_default_timezone_set('Europe/Vienna');

EventLoop::delay(3, function (): void {
  echo "Delayed: " . date('Y-m-d h:i:s') . PHP_EOL;
});

echo "Pre-run: " . date('Y-m-d h:i:s') . PHP_EOL;

EventLoop::run();
```

```
Pre-run: 2025-10-16 10:49:34
Delayed: 2025-10-16 10:49:37
```

https://revolt.run/timers

REVOLT PRIMITIVES

Repeat - The callback is executed after the specified number of seconds, repeatedly. Fractions of a second may be expressed as floating point numbers.

```php
use Revolt\EventLoop;

EventLoop::repeat(0.1, function ($callbackId): void {
  static $i = 0;

  if ($i++ < 3) {
    echo "tick\n";
  } else {
    EventLoop::cancel($callbackId);
  }
});

EventLoop::run();
```

https://revolt.run/timers

REVOLT PRIMITIVES

Repeat - The callback is executed after the specified number of seconds, repeatedly. Fractions of a second may be expressed as floating point numbers.

```php
use Revolt\EventLoop;

EventLoop::repeat(0.1, function ($callbackId): void {
  static $i = 0;

  if ($i++ < 3) {
    echo "tick\n";
  } else {
    EventLoop::cancel($callbackId);
  }
});

EventLoop::run();
```

```
tick
tick
tick
```

https://revolt.run/timers

REVOLT PRIMITIVES

Stream readable/writable - The callback is executed when there's data on the stream to be read, or the connection closed. The callback is executed when there's enough space in the write buffer to accept new data to be written.

```php
use Revolt\EventLoop;

EventLoop::onReadable($socket, function ($callbackId, $socket) {
  $socketId = (int) $socket;
  $newData = @fread($socket, IO_GRANULARITY);
  if ($newData != "") {
    // There was actually data and not an EOF notification. Let's consume it!
    parseIncrementalData($socketId, $newData);
  } elseif (isStreamDead($socket)) {
    EventLoop::cancel($callbackId);
  }
});

EventLoop::run();
```

https://revolt.run/streams

REVOLT PRIMITIVES

Signal - The callback is executed when the process received a specific signal from the Operating System.

```php
use Revolt\EventLoop;

// Let's tick off output once per second, so we can see activity.
EventLoop::repeat(1, function (): void {
  echo "tick: ", date('c'), "\n";
});

// What to do when a SIGINT signal is received
EventLoop::onSignal(SIGINT, function (): void {
  echo "Caught SIGINT! exiting ...\n";
  exit;
});

EventLoop::run();
```

https://revolt.run/signals

REVOLT PRIMITIVES

Signal - The callback is executed when the process received a specific signal from the Operating System.

```php
use Revolt\EventLoop;

// Let's tick off output once per second, so we can see activity.
EventLoop::repeat(1, function (): void {
  echo "tick: ", date('c'), "\n";
});

// What to do when a SIGINT signal is received
EventLoop::onSignal(SIGINT, function (): void {
  echo "Caught SIGINT! exiting ...\n";
  exit;
});

EventLoop::run();
```

```
tick: 2025-10-16T08:58:11+00:00
tick: 2025-10-16T08:58:12+00:00
tick: 2025-10-16T08:58:13+00:00
^CCaught SIGINT! exiting ...
```

https://revolt.run/signals

REVOLT IN DRUPAL

RENDERER REWRITTEN

```php
$fibers = [];
foreach ($elements['#attached']['placeholders'] as $placeholder => $placeholder_element) {
  // Get the render array for the given placeholder.
  $fibers[$placeholder] = new \Fiber(function () use ($placeholder_element) {
    return [$this->doRenderPlaceholder($placeholder_element), $placeholder_element];
  });
}
$iterations = 0;
while (count($fibers) > 0) {
  foreach ($fibers as $placeholder => $fiber) {
    if (!$fiber->isStarted()) {
      $fiber->start();
    }
    elseif ($fiber->isSuspended()) {
      $fiber->resume();
    }
    // If the Fiber hasn't terminated by this point, move onto the next
    // placeholder, we'll resume this fiber again when we get back here.
    if (!$fiber->isTerminated()) {
      // If we've gone through the placeholders once already, and they're
      // still not finished, then start to allow code higher up the stack to
      // get on with something else.
      if ($iterations) {
        $fiber = \Fiber::getCurrent();
        if ($fiber !== NULL) {
          $fiber->suspend();
        }
      }
      continue;
    }
    [$markup, $placeholder_element] = $fiber->getReturn();

    $elements = $this->doReplacePlaceholder($placeholder, $markup, $elements, $placeholder_element);
    unset($fibers[$placeholder]);
  }
  $iterations++;
}
```

RENDERER REWRITTEN

```php
// Prepare the tasks for our placeholders.
$placeholder_operations = [];
foreach ($elements['#attached']['placeholders'] as $placeholder => $placeholder_element) {
  // Get the render array for the given placeholder.
  $placeholder_operations[$placeholder] = fn () => [
    $this->doRenderPlaceholder($placeholder_element),
    $placeholder_element,
  ];
}

// Execute the tasks and process the rendered placeholders as they become available.
foreach (stream($placeholder_operations) as $placeholder => $result) {
  if ($result->isError()) {
    throw $result->getValue();
  }
  [$markup, $placeholder_element] = $result->getValue();

  $elements = $this->doReplacePlaceholder($placeholder, $markup, $elements, $placeholder_element);
}
```

RENDERER REWRITTEN

```php
use Drupal\Core\Result;
use Revolt\EventLoop;

function stream(array $operations) : \Generator {
  $suspension = EventLoop::getSuspension();

  // Kick of all the events that happen.
  foreach ($operations as $key => $operation) {
    EventLoop::defer(function () use ($suspension, $key, $operation) {
      try {
        $result = $operation();
        $suspension->resume([$key, Result::ok($result)]);
      }
      catch (\Throwable $e) {
        $suspension->resume([$key, Result::error($e)]);
      }
    });
  }

  // We must suspend the same number of times that we queued operations to make
  // sure we complete them all.
  for ($i = 0, $j = count($operations); $i < $j; $i++) {
    // We don't need to try/catch here since we ensure errors thrown by the
    // operation itself are caught earlier.
    [$key, $result] = $suspension->suspend();
    yield $key => $result;
  }
}
```

AMPHP

AMPHP

- https://amphp.org/
- A collection of asynchronous libraries compatible with Revolt

Highlighted packages

- amphp/amp - Fundamental primitives for task coordination
- amphp/http-client - Non-blocking HTTP Client
- amphp/mysql - Async connection pool for MySQL
- amphp/parallel - Spawn and utilise native threads in a non-blocking manner
- amphp/websocket-client - Websocket client connections that are non-blocking

AMPHP/AMP

- Fundamental primitives for task coordination

- await($iterable, $cancellation) - Await all tasks, abort on first error
- awaitAnyN($count, $iterable, $cancellation) - Allow errors, wait for N successes
- awaitAll($iterable, $cancellation) - Await all tasks, but return both successes and errors
- awaitFirst($iterable, $cancellation) - Get result of first completed success/error
  Replaces our custom await_first implementation but supports cancellation.
- awaitAny($iterable, $cancellation) - Get the first successful completion

- Cancellation primitives: TimeoutCancellation, SignalCancellation, DeferredCancellation, NullCancellation, and CompositeCancellation.

SUMMARISING

IN SUMMARY

- Async code lets us intertwine different tasks rather than doing one thing at a time
- Async code can be incredibly powerful
- Async code can be incredibly difficult

- Event Loops exist to help coordinate different type of taks cooperatively
- Revolt is the chosen event loop for Drupal as it was co-developed by multiple async library maintainers

- Use Revolt primitives to schedule your own tasks rather than using Fibers directly
- Pull in a library such as Amphp/amp for more complex tasks and flow control such as races or cancellation

GET EXCITED ABOUT ASYNC DRUPAL
HELP ME MAKE THIS A REALITY

https://www.alexandervarwijk.com/talks/2025-10-16-thinking-async