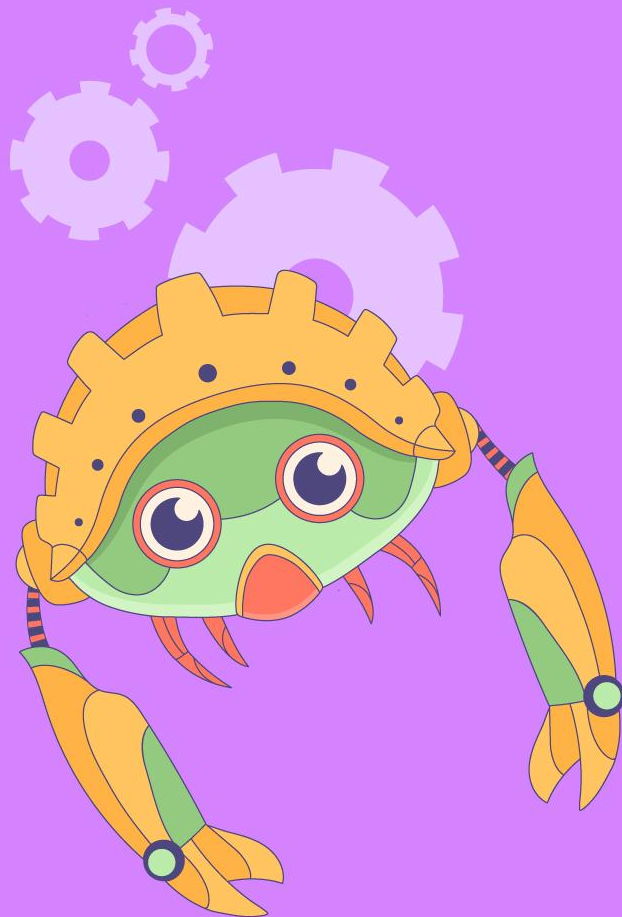


Hayden Stainsby

Engineering Manager @ HERE Technologies

# Observing Tokio

Shining a light into your async runtime.



# Hayden Stainsby

Engineering Manager @ HERE Technologies  
Tokio-rs maintainer

Let's be friends!

email: [hds@caffeineconcepts.com](mailto:hds@caffeineconcepts.com)

web-site: [hegdenu.net](http://hegdenu.net)

github: [hds](https://github.com/hds)

mastodon: [@hds@hachyderm.io](https://hds@hachyderm.io)

rust for lunch: [lunch.rs](http://lunch.rs)



# Overview

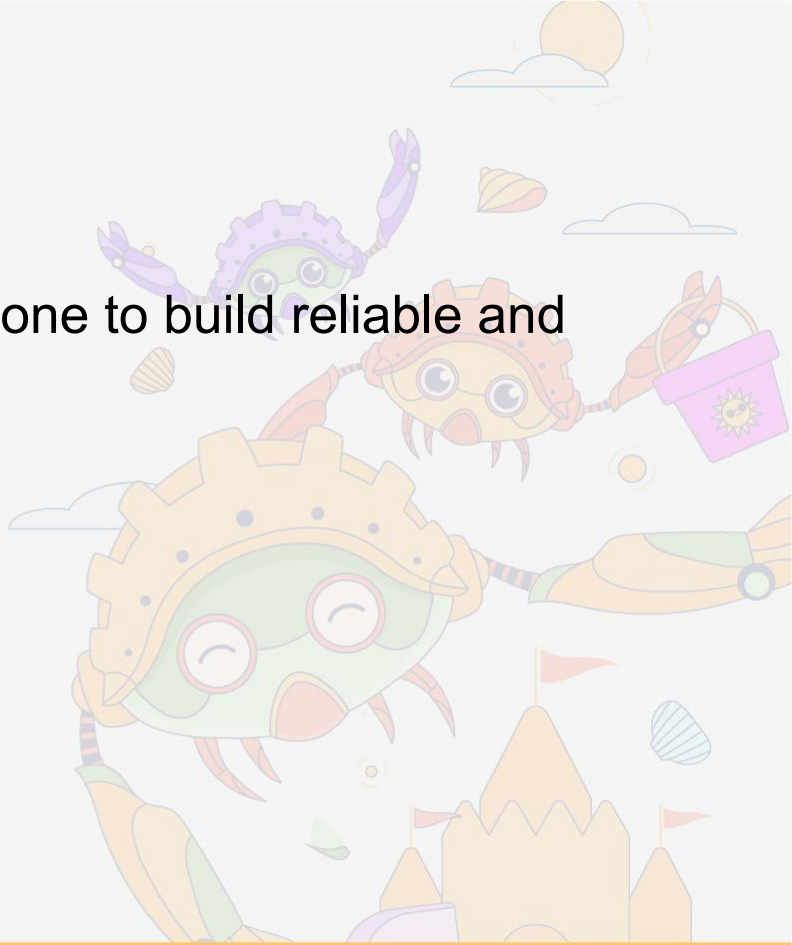
- Context
- Tokio
- Metrics
- Tokio Console
- Task Dump
- Present & Future
- Links

# Context

First, a little bit of context.

▶ Rust

- A language empowering everyone to build reliable and efficient software.
- ...we're at RustLab, right?

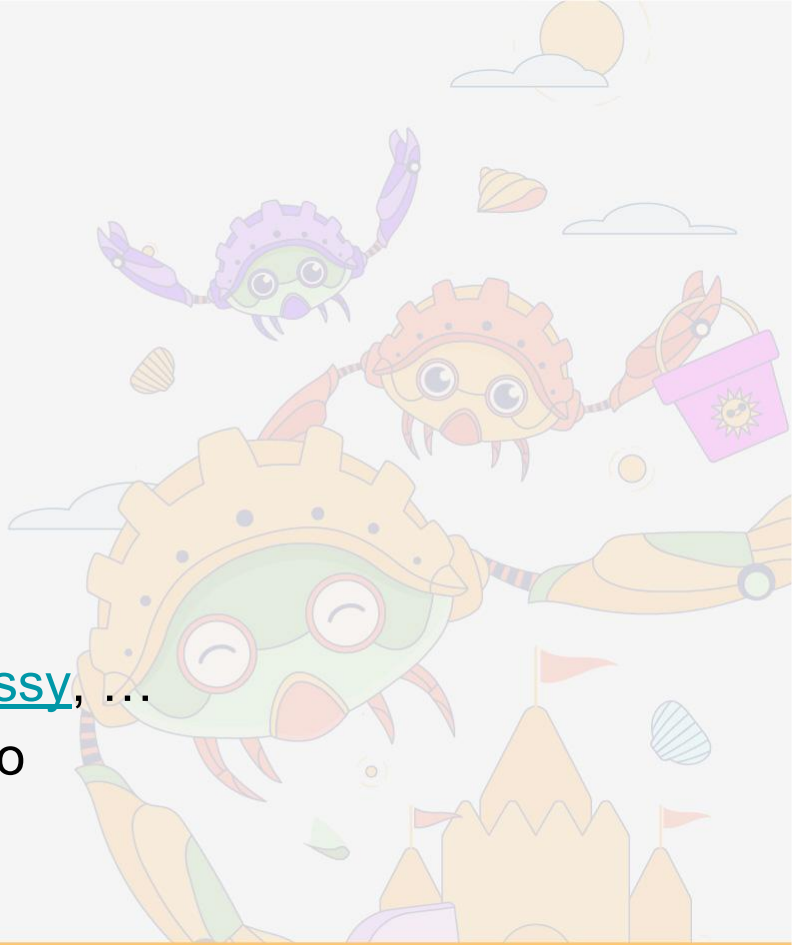


## ▶ Asynchronous (async) programming

- A concurrent programming model
- Difference between parallelism and concurrency?
  - Parallelism is when you **do** many things at the same time
  - Concurrency is when you **wait** for many things at the same time
- Basic unit of concurrency: task
- Run many async tasks on few OS threads
- (Often) scheduling is cooperative

## ▶ Rust, but async

- Rust provides...
  - some async definitions
  - `async/.await` syntax
  - **no** async runtime!
- You get to pick your own
- [Tokio](#), [Async-std](#), [Smol](#), [Embassy](#), ...
- It's likely that you're using Tokio



▶ Async Rust can be a bit tricky

- Cooperative scheduling is fine until someone doesn't cooperate
- `async/.await` syntax gets converted into Futures
- Stack traces aren't so meaningful
- The debugger overly affects what happens next
- Sometimes your application just hangs...





# Async Rust is Hard



▶ Last minute slide

- Wait!
- Async Rust is hard?
- Didn't we already see this yesterday?
- Antonio Scandurra stole the premise of my talk! 😜
- [Property-Testing Async Code in Rust](#)
- The perfect companion for what I'm going to show
- Back to our regularly scheduled programming...



# Async Rust is Hard





# Async Rust is Hard



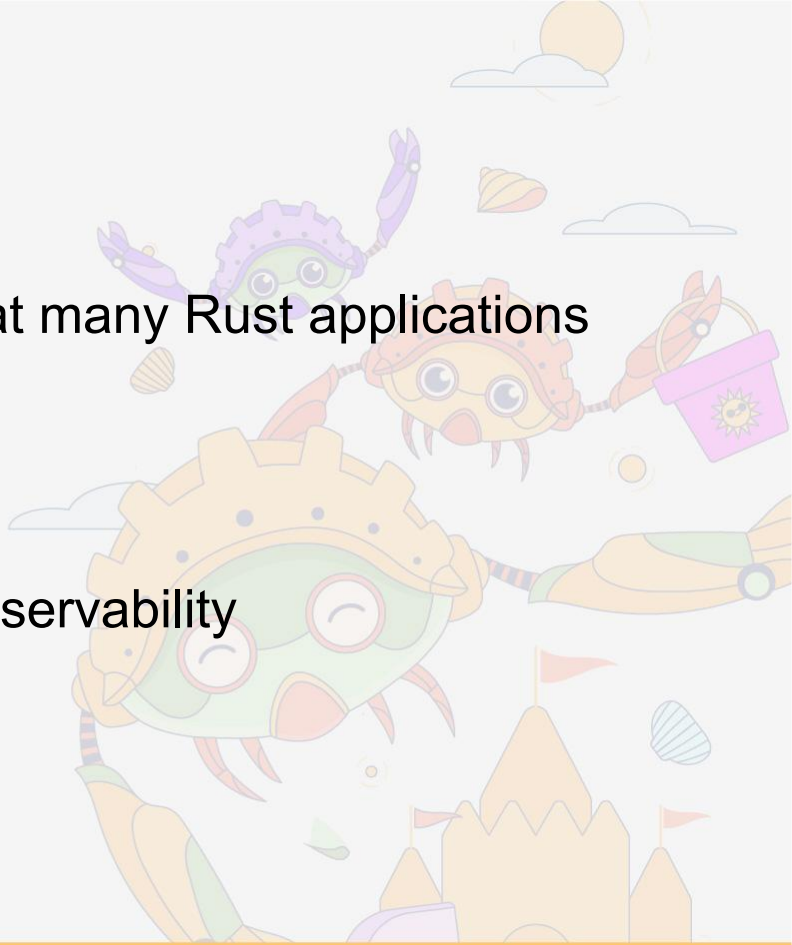


# now what?



## ► Superpowers

- Luckily we have something that many Rust applications don't have
- Superpowers!
- Actually: a runtime
- This is where we can insert observability



## ► Observability

*“[Observability is] a measure of how well you can understand and explain any state your system can get into, no matter how novel or bizarre.”*

– Observability Engineering by Charity Majors, Liz Fong-Jones, and George Miranda (O'Reilly).

- Do we have "full" observability of Tokio?
- Not yet. But more than you might imagine!

Observing Tokio

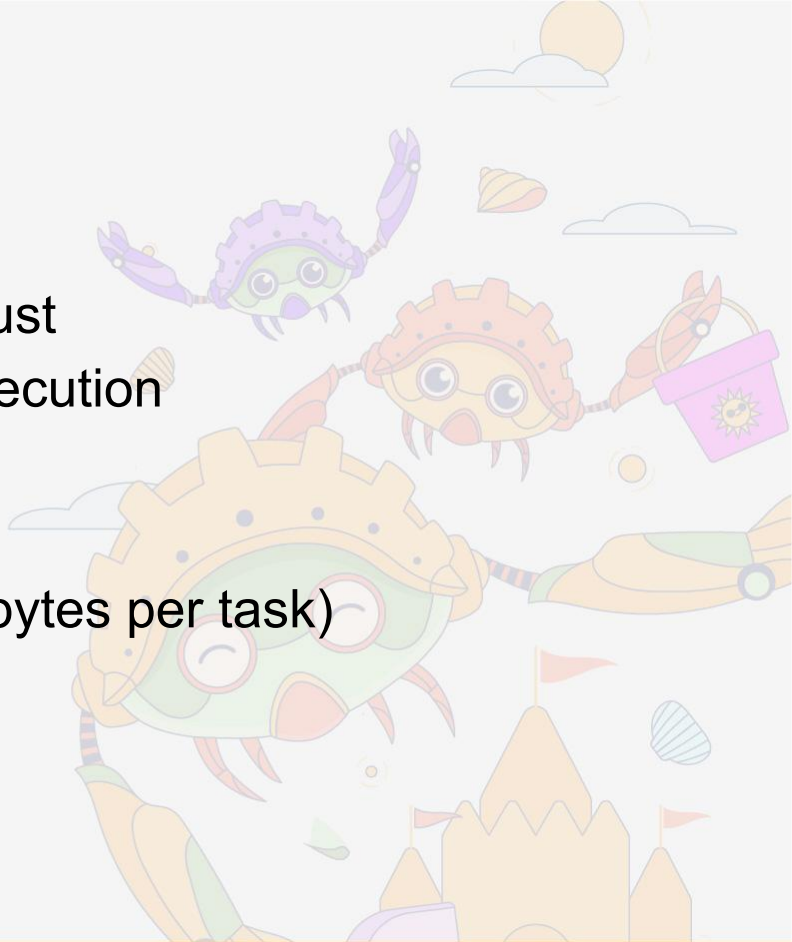
# Tokio

A runtime for writing reliable,  
asynchronous, and slim applications  
with the Rust programming language.



## ► What is Tokio?

- A popular async runtime for Rust
- Drives task scheduling and execution
- Cooperative scheduling
- Work stealing
- Tasks are low-overhead (~88 bytes per task)



▶ What else?

- Timers
- Synchronization primitives
- Network and file I/O
- Message passing channels
- ~~Kitchen sink~~
- "Batteries included"



## ► Instrumentation

- Tokio has instrumentation built in
  - Metrics
  - Traces
  - Task Dumps
- This provides the observability we need



## ► Unstable

- Most tools in this presentation require unstable APIs
- Unstable doesn't mean untested
- (although it does mean less used)
- API may break outside major version bumps

## ► Unstable

```
# Set the RUSTFLAGS variable
RUSTFLAGS="--cfg tokio_unstable" cargo build

# Contents of .cargo/config
[build]
rustflags = ["--cfg", "tokio_unstable"]
```

# Metrics

Monitor key metrics of tokio tasks and runtimes.

## ► Metrics in Tokio

- Metrics are counts or measures aggregated over time
- Tokio has [runtime metrics](#)
- The [tokio-metrics](#) util crate adds functionality



## ▶ Runtime metrics

- Available from runtime `Handle`
- 25 metrics, including
  - Active task count
  - Poll time histograms
  - Worker metrics
  - I/O driver metrics
  - and more...





## ▶ Using runtime metrics

```
#[tokio::main]
async fn main() {
    let metrics: tokio::runtime::RuntimeMetrics = tokio::runtime::Handle::current().metrics();

    for _ in 0..10 {
        tokio::spawn(tokio::time::sleep(Duration::from_millis(10)));
    }

    let n = metrics.active_tasks_count();
    println!("Runtime has {} active tasks", n);
}
```

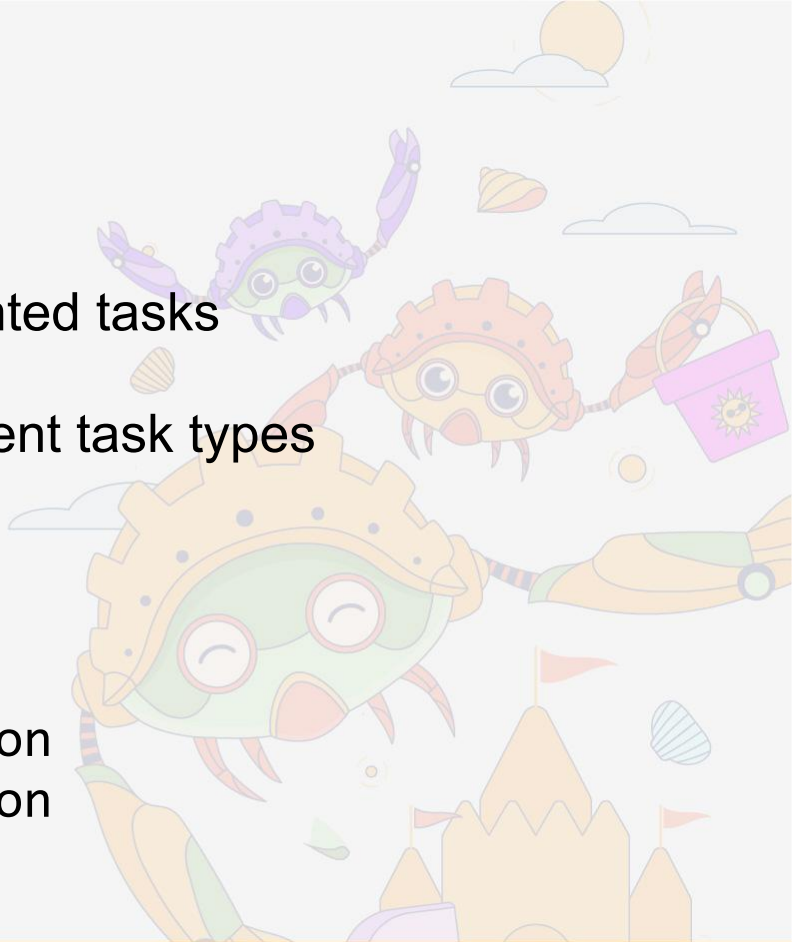
Runtime has 10 active tasks

▶ `tokio-metrics crate`

- Provides patterns for monitoring tasks or whole runtimes
- Iterator yields metric intervals
- Metrics are provided for the interval since the last call
- Great for preparing metrics for a collector
  - e.g. Prometheus

▶ Task Monitor (no `tokio_unstable` required)

- Track key metrics of instrumented tasks
- Explicitly instrument tasks
- Use distinct monitors for different task types
- Recorded metrics
  - `total_poll_count`
  - `total_idle_duration`
- Derived metrics
  - `mean_scheduled_duration`
  - `mean_fast_poll_duration`



▶ Slow polls and long delays

- Separate poll times into slow and fast (50 $\mu$ s)
- Separate schedule delays into long and short (50 $\mu$ s)
- Answer questions like, are my tasks...
  - taking longer to poll?
  - spending more time waiting to be polled?
  - spending more time waiting on external events to complete?
- Incredible [documentation!](#)

## ▶ Using Task Monitor

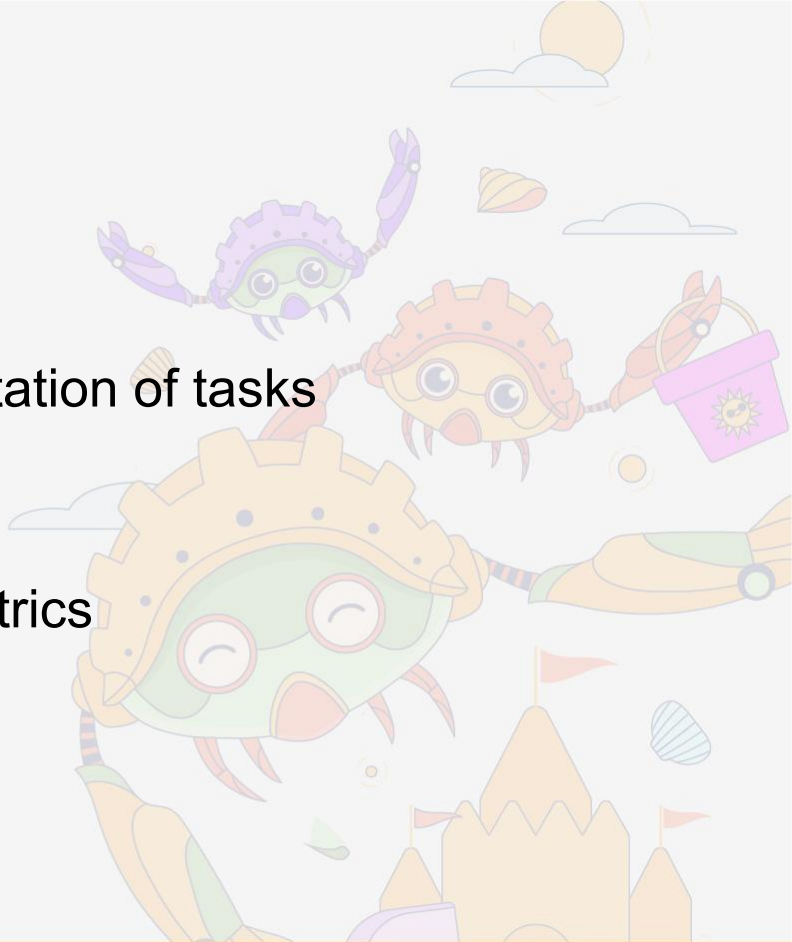
```
let task_monitor = tokio_metrics::TaskMonitor::new();

for idx in 0..100 {
    tokio::spawn(task_monitor.clone().instrument(good_times(idx)));
}

let mut intervals = task_monitor.intervals();
for _ in 0..10 {
    tokio::time::sleep(Duration::from_millis(100)).await;
    let interval = intervals.next().unwrap();
    println!("{:?}", interval);
}
```

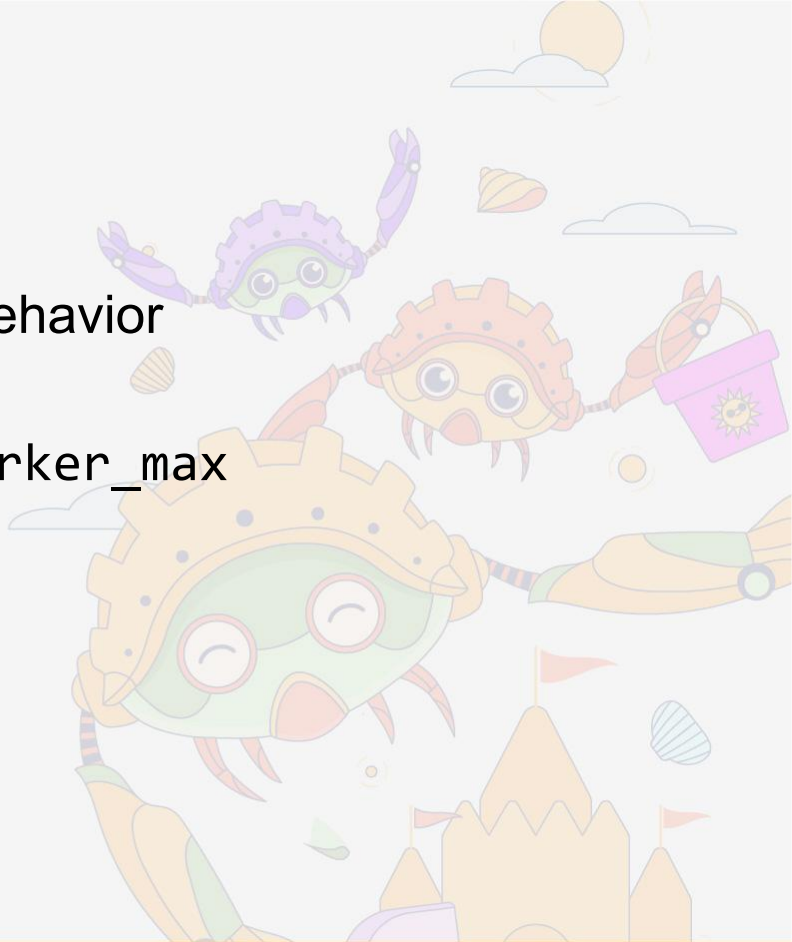
## ▶ Runtime Monitor

- Metrics for entire runtime
- No need for explicit instrumentation of tasks
- Wraps Tokio's RuntimeMetrics
  - Intervals for metrics
  - Snapshot of all worker metrics
  - Derived metrics



▶ Worker metrics

- Useful summaries of worker behavior
  - `max_busy_duration`
  - `mean_poll_duration_worker_max`
  - `min_steal_operations`
  - `max_overflow_count`



## ▶ Using runtime monitor

```
let handle = tokio::runtime::Handle::current();
let runtime_monitor = tokio_metrics::RuntimeMonitor::new(&handle);

for _ in 0..10 {
    tokio::spawn(tokio::time::sleep(Duration::from_millis(10)));
}

let mut intervals = runtime_monitor.intervals();
for _ in 0..10 {
    let interval = intervals.next().unwrap();
    println!("{:?}", interval);
    tokio::time::sleep(Duration::from_millis(10)).await;
}
```



## ▶ Integration with prometheus

- Monitors are easy to integrate with collectors
- The prometheus crate for example
- But there is some boilerplate involved...

```
let tokio_metrics = TokioMetrics {
    worker_count: register(
        registry,
        IntGauge::with_opts(Opts::new(
            "tokio_worker_count_total",
            "The number of worker threads used by the runtime.",
        )),
    ),
    // Many more here...
}

// During `/metrics` request ...
let runtime_metrics = intervals.next().unwrap();
tokio_metrics.worker_count.set(runtime_metrics.workers_count as i64);
// And more ...
```

## Metrics

### ▶ Did someone say pretty graphs?



## ▶ Summary: Metrics

- Available from runtime Handle
- Can be exposed to tools such as Prometheus
- Task metrics for targeted instrumentation
  - No `tokio_unstable` required
- Lightweight
- Suitable for running in production

# Tokio Console

A diagnostics and debugging tool for asynchronous Rust programs.

## ▶ Introducing Tokio Console

- Built on tracing instrumentation in Tokio
- **Subscriber**
  - Collects and analyzes traces from within target application
- **Console**
  - Terminal application to visualize and explore runtime state
- **API**
  - gRPC API that defines the communication between subscriber and console

## ▶ Enabling console subscriber

- Add the crate to your Cargo.toml
  
- Not using tracing yet?  
Easy!

```
[dependencies]
console-subscriber = "0.2"

#[tokio::main]
async fn main() {
    console_subscriber::init();
    // All your code ...
}
```

## ▶ Enabling console subscriber

- Already using another tracing subscriber?
- Also easy!

```
fn tracing_setup() {  
    use tracing_subscriber::prelude::*;  
  
    let another_layer = // ...  
    let console_layer =  
        console_subscriber::ConsoleLayer::builder()  
            .spawn();  
  
    tracing_subscriber::registry()  
        .with(console_layer)  
        .with(another_layer)  
        .init();  
}
```

## ▶ Start the console

- Install from cargo
- And run from the command line
- For best experience, ensure UTF-8

```
cargo install tokio-console
```

```
tokio-console
```

```
tokio-console --lang=en_us.UTF-8
```

```
# or
```

```
LANG="en_us.UTF-8" tokio-console
```



## ▶ Task list

```

connection: http://localhost:6669/ (CONNECTED)
views: t = tasks, r = resources
controls: select column (sort) = ← or h, l, scroll = ↑ or k, j, view details = ↵, invert sort (highest/lowest) = i, scroll to top = gg, scroll to bottom = G,
toggle pause = space, quit = q

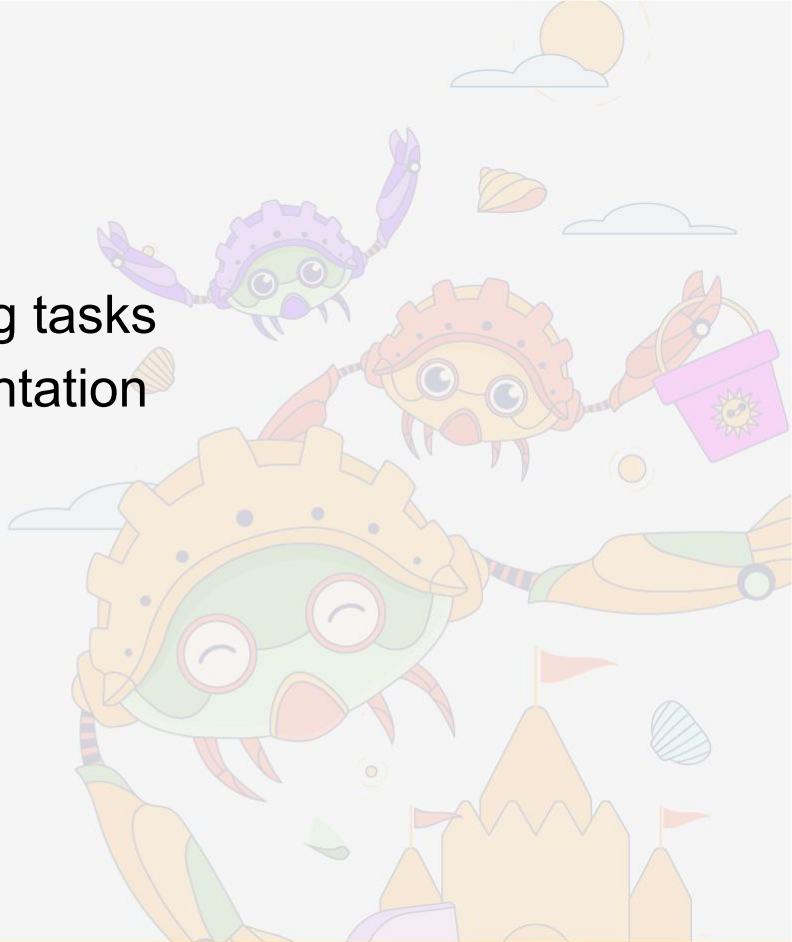
Warnings
▲ 1 tasks have lost their waker

Tasks (25) ▶ Running (13) ◀ Idle (7)
Warn ID State Name Total Busy Sched Idle Poll Kind Location Fields
▲ 1 20 ▢ 20s 44µs 0ns 20s 1 task simple-console/src/bin/quick_start.rs:7:5 target-tokio::task
29 ▢ 18s 1s 9ms 17s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
30 ▢ 18s 1s 9ms 17s 95 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
31 ▢ 18s 1s 11ms 17s 90 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
28 ▢ 17s 1s 9ms 16s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
38 ▢ 17s 1s 9ms 16s 97 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
39 ▶ 17s 1s 7ms 15s 91 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
37 ▢ 16s 1s 11ms 15s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
27 ▢ 16s 1s 9ms 15s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
40 ▢ 16s 987ms 7ms 15s 85 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
41 ▢ 16s 923ms 9ms 15s 80 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
26 ▢ 15s 1s 8ms 14s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
36 ▢ 15s 1s 7ms 14s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
48 ▢ 15s 1s 7ms 14s 99 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
25 ▢ 14s 1s 9ms 13s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
35 ▢ 14s 1s 8ms 13s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
45 ▢ 14s 1s 9ms 13s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
24 ▢ 13s 1s 9ms 12s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
34 ▢ 13s 1s 10ms 12s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
44 ▢ 13s 1s 10ms 12s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
43 ▢ 12s 1s 8ms 11s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
33 ▢ 12s 1s 7ms 11s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
32 ▢ 11s 1s 8ms 10s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task
42 ▢ 11s 1s 5ms 10s 102 task simple-console/src/bin/quick_start.rs:11:13 target-tokio::task

```

## ▶ Task Names

- Task builder API allows naming tasks
- Only used by tracing instrumentation
- Very useful in Tokio Console



## ▶ Sample code with task names

```
#[tokio::main]
async fn main() {
    console_subscriber::init();

    tokio::task::Builder::new()
        .name("great-times")
        .spawn(great_times())
        .unwrap();
}
```

## ▶ Task list with names

```

connection: http://localhost:6669/ (CONNECTED)
views: t = tasks, r = resources
controls: select column (sort) = ← or h, l, scroll = ↑↓ or k, j, view details = ↵, invert sort (highest/lowest) = i, scroll to top = gg, scroll to bottom = G,
toggle pause = space, quit = q

Warnings
▲ 1 tasks have lost their waker

Tasks (23) ▶ Running (1) ◄ Idle (16)
Warn ID State Name Total Busy Sched Idle Polls Kind Location Fields
▲ 1 20 ◄ great-times 15s 38µs 0ms 15s 1 task simple-console/src/bin/task_names.rs:9:10 target-tokio::task
25 ▣ good-times-3 14s 1s 6ms 13s 102 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
26 ◄ good-times-4 14s 1s 6ms 13s 92 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
27 ▶ good-times-5 14s 996ms 8ms 13s 86 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
28 ◄ good-times-6 14s 926ms 7ms 13s 80 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
31 ◄ good-times-7 14s 852ms 6ms 13s 74 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
24 ▣ good-times-2 13s 1s 10ms 12s 102 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
32 ◄ good-times-8 13s 797ms 7ms 13s 69 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
33 ◄ good-times-9 13s 746ms 6ms 12s 65 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
36 ◄ good-times-2 13s 1s 7ms 11s 94 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
23 ▣ good-times-1 12s 1s 7ms 11s 102 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
37 ◄ good-times-3 12s 1s 6ms 11s 86 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
35 ▣ good-times-1 12s 1s 8ms 11s 102 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
38 ◄ good-times-4 12s 923ms 6ms 11s 79 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
39 ◄ good-times-5 12s 849ms 5ms 11s 73 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
40 ◄ good-times-6 12s 783ms 7ms 11s 68 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
41 ◄ good-times-7 12s 732ms 5ms 11s 63 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
22 ▣ good-times-0 11s 1s 9ms 10s 102 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
34 ▣ good-times-0 11s 1s 10ms 10s 102 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
42 ◄ good-times-8 11s 686ms 4ms 11s 59 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
43 ◄ good-times-9 11s 631ms 5ms 10s 55 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
44 ◄ good-times-0 11s 1s 5ms 10s 97 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task
45 ◄ good-times-1 11s 1s 9ms 10s 87 task simple-console/src/bin/task_names.rs:16:18 target-tokio::task

```

## ▶ Task details

The screenshot shows the Tokio Console interface with a task details view. The window title is 'tokio-console' and the system tray shows '11:11'. The connection status is 'http://localhost:6669/ (CONNECTED)'. The views are 't = tasks, r = resources' and the controls are 'return to task list = esc, toggle pause = space, quit = q'.

**-Task-**

- ID: 24
- Name: good-times-2
- Target: tokio::task
- Location: simple-console/src/bin/task\_names.rs:16:18
- Total Time: 11.60s
- Busy: 1.01s (8.72%)
- Scheduled: 7.30ms (0.06%)
- Idle: 10.58s (91.22%)

**-Waker-**

- Current wakers: 1 (clones: 173, drops: 172)
- Woken: 86 times, last woken: 25.198774ms ago

**-Poll Times Percentiles-**

- p10: 10.49ms
- p25: 10.94ms
- p50: 11.60ms
- p75: 12.32ms
- p90: 12.78ms
- p95: 12.85ms
- p99: 13.30ms

**-Poll Times Histogram-**

The histogram shows the distribution of poll times, with a peak around 11.60ms. The x-axis ranges from 10.22ms to 13.30ms, and the y-axis shows a count of 5.

**-Scheduled Times Percentiles-**

- p10: 19.71μs
- p25: 36.86μs
- p50: 66.56μs
- p75: 108.54μs
- p90: 180.22μs
- p95: 214.02μs
- p99: 370.69μs

**-Scheduled Times Histogram-**

The histogram shows the distribution of scheduled times, with a peak around 66.56μs. The x-axis ranges from 15.94μs to 370.69μs, and the y-axis shows a count of 6.

**-Fields-**

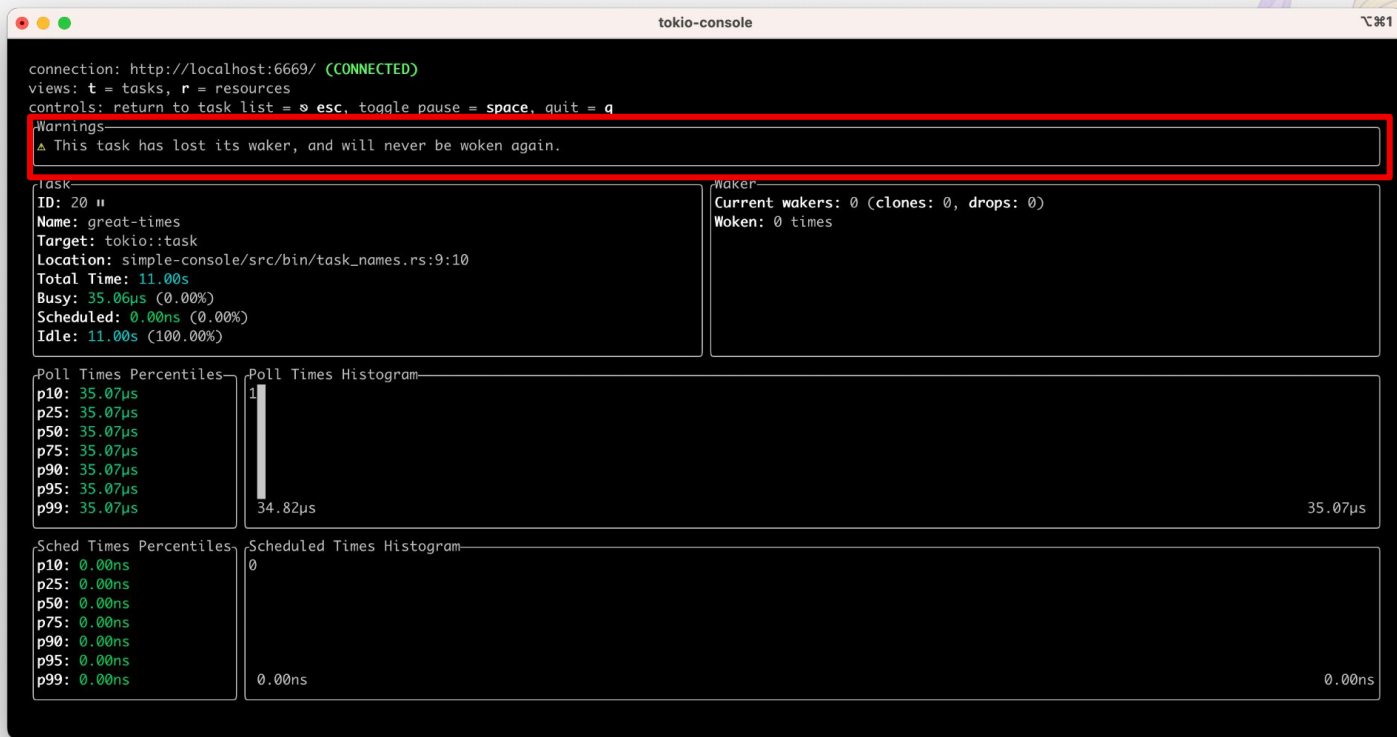
- target: tokio::task



# A wild warning has appeared!



## ▶ Task details

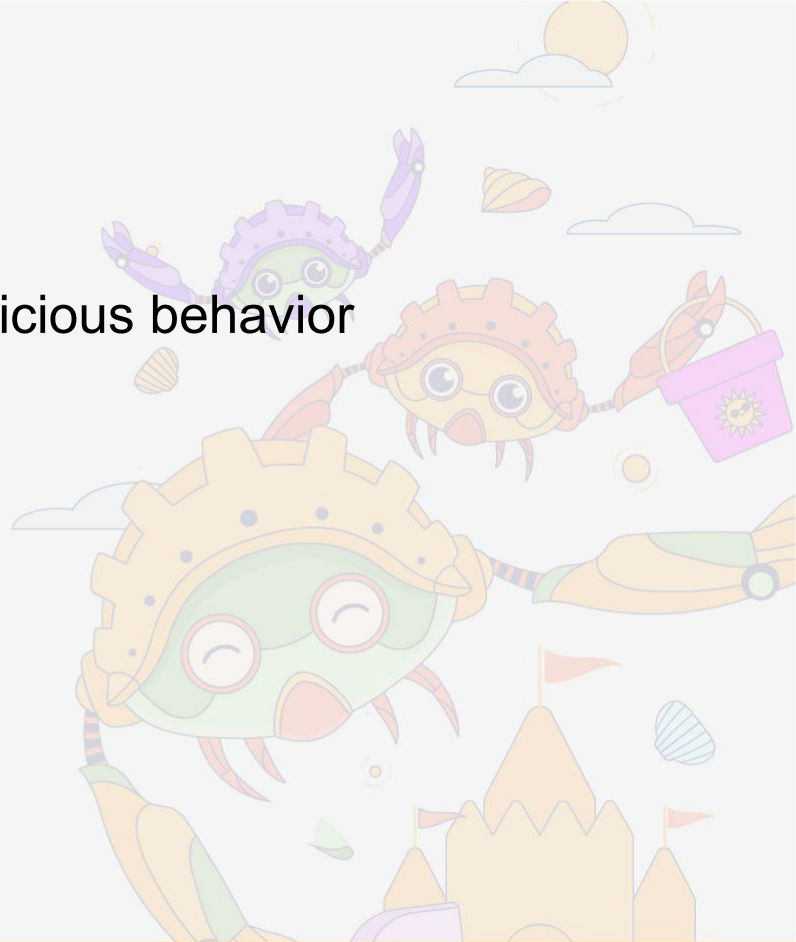


The screenshot shows the Tokio Console interface with a task details view. A red box highlights a warning message: "⚠ This task has lost its waker, and will never be woken again." The task information includes ID, name, target, location, and various timing statistics. The poll times histogram shows a single bar at 35.07µs, and the scheduled times histogram shows a single bar at 0.00ns.

```
tokio-console ⌂ 31  
connection: http://localhost:6669/ (CONNECTED)  
views: t = tasks, r = resources  
controls: return to task list = esc, toggle pause = space, quit = q  
Warnings  
⚠ This task has lost its waker, and will never be woken again.  
Task  
ID: 20 tt  
Name: great-times  
Target: tokio::task  
Location: simple-console/src/bin/task_names.rs:9:10  
Total Time: 11.00s  
Busy: 35.06µs (0.00%)  
Scheduled: 0.00ns (0.00%)  
Idle: 11.00s (100.00%)  
Waker  
Current wakers: 0 (clones: 0, drops: 0)  
Woken: 0 times  
Poll Times Percentiles: Poll Times Histogram  
p10: 35.07µs  
p25: 35.07µs  
p50: 35.07µs  
p75: 35.07µs  
p90: 35.07µs  
p95: 35.07µs  
p99: 35.07µs  
34.82µs 35.07µs  
Scheduled Times Percentiles: Scheduled Times Histogram  
p10: 0.00ns  
p25: 0.00ns  
p50: 0.00ns  
p75: 0.00ns  
p90: 0.00ns  
p95: 0.00ns  
p99: 0.00ns  
0.00ns 0.00ns
```

## ► Lints

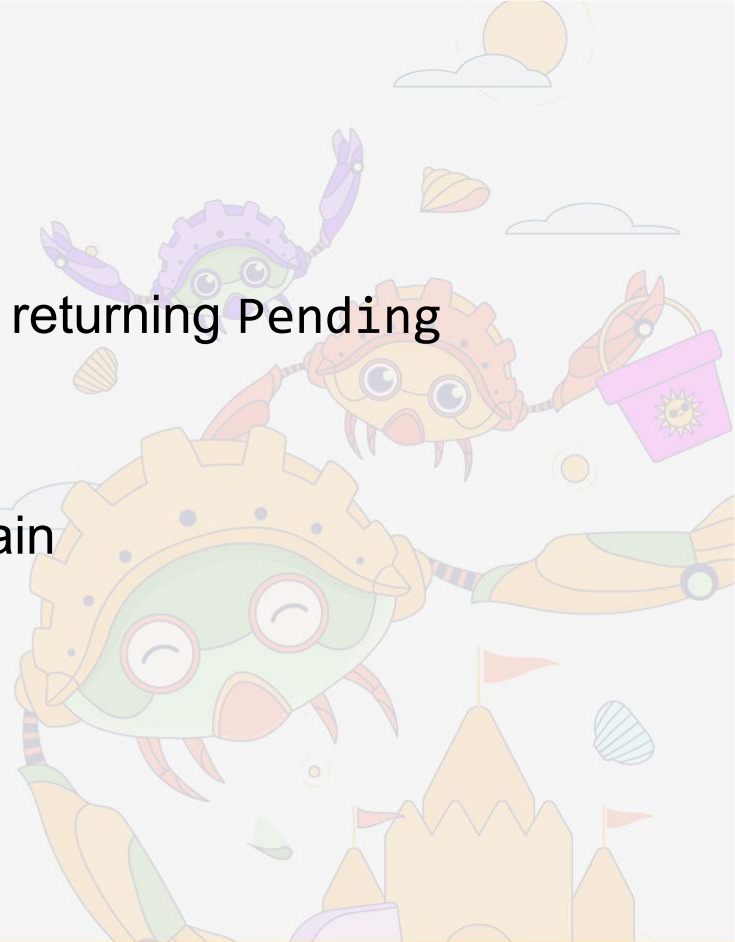
- Tokio console can detect suspicious behavior
- Detection provided by lints
  - Lost waker
  - Self wakes
  - Never yielded
  - More to come...





## ▶ Lost waker

- An idle task must be woken after returning Pending
- Done via the “waker”
- If the waker isn’t held on to...
- The task will never be woken again



## ▶ Lost waker – what could it be?

```
fn poll(
    mut self: std::pin::Pin<&mut Self>,
    cx: &mut std::task::Context<'_>,
) -> std::task::Poll<Self::Output> {
    if !self.slept {
        let mut sleep = Box::pin(tokio::time::sleep(Duration::from_millis(100)));
        if let Poll::Ready(_) = sleep.poll_unpin(cx) {
            self.slept = true
        }
        return Poll::Pending;
    }
    Poll::Ready(())
}
```

We're not keeping `sleep` anywhere!

## ▶ Lost waker - diagnostics

The screenshot shows the Tokio Console interface with a warning and diagnostic data for a task. The warning states: "This task has lost its waker, and will never be woken again." The diagnostic data is organized into four panels:

- Task:**
  - ID: 21
  - Name: sleep once
  - Target: tokio::task
  - Location: simple-console/src/bin/lints.rs:25:10
  - Total Time: 10.00s
  - Busy: 461.60µs (0.00%)
  - Scheduled: 0.00ns (0.00%)
  - Idle: 10.00s (100.00%)
- Waker:**
  - Current wakers: 0 (clones: 1, drops: 1)
  - Woken: 0 times
- Poll Times Percentiles:**
  - p10: 462.85µs
  - p25: 462.85µs
  - p50: 462.85µs
  - p75: 462.85µs
  - p90: 462.85µs
  - p95: 462.85µs
  - p99: 462.85µs
- Poll Times Histogram:** A histogram showing a single bar at 462.85µs.
- Sched Times Percentiles:**
  - p10: 0.00ns
  - p25: 0.00ns
  - p50: 0.00ns
  - p75: 0.00ns
  - p90: 0.00ns
  - p95: 0.00ns
  - p99: 0.00ns
- Scheduled Times Histogram:** A histogram showing a single bar at 0.00ns.

## ▶ Self wakes

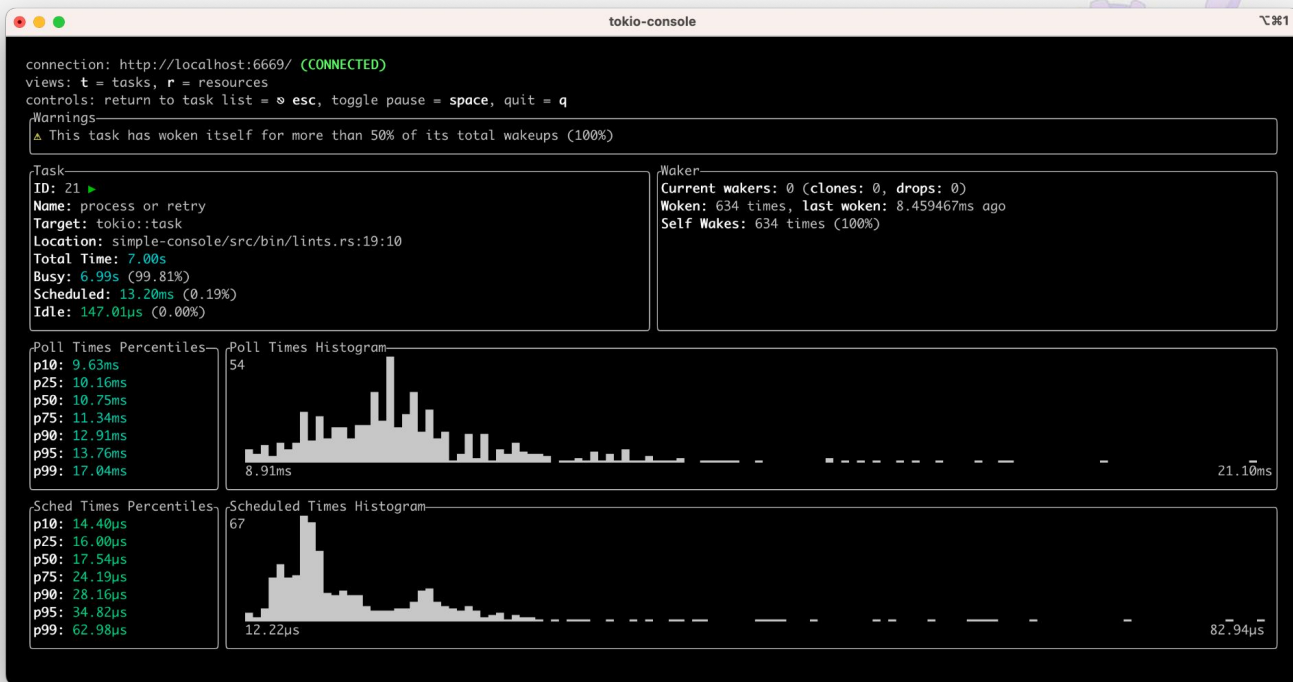
- A task may self wake (e.g. `yield_now()`)
- Self waking too often may indicate a problem
  - Tokio's coop budget is forcing the task to yield
  - The task would otherwise block
  - Consider moving it outside the runtime

## ► Self wakes - what could it be?

```
async fn process_or_retry(tx: mpsc::Sender<Msg>, mut rx: mpsc::Receiver<Msg>) {  
    while let Some(msg) = rx.recv().await {  
        if !msg.ready() {  
            msg.step(); // 10µs  
            tx.send(msg).await.unwrap();  
        } else {  
            msg.process() // 40µs  
        }  
    }  
}
```

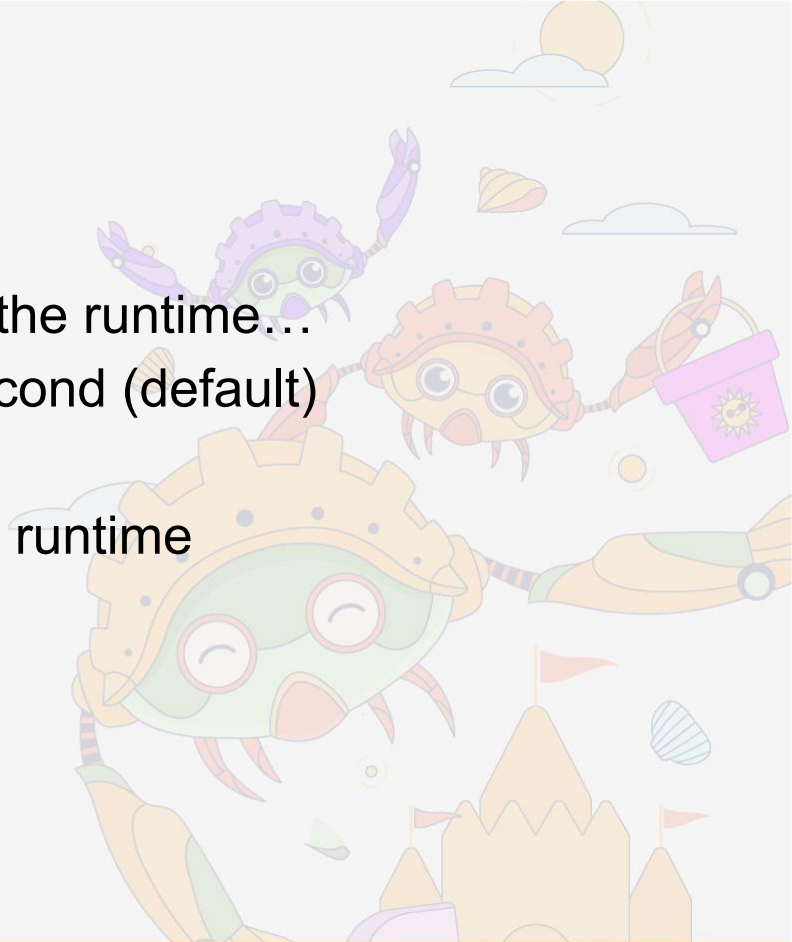
We might be keeping our queue “always ready”!

## ▶ Self wakes - diagnostics



## ▶ Never yielded

- The task has never yielded to the runtime...
- and has been running for 1 second (default)
- The task is blocking
- Consider moving it outside the runtime



## ▶ Never yielded - what could it be?

```
async fn do_work() {  
    let initial_items = pre_work().await;  
  
    for item in initial_items {  
        process_item(item).await;  
        task::yield_now().await;  
    }  
}
```

How long does this take?





## ▶ Never yielded - diagnostics

The screenshot shows the Tokio Console interface with the following content:

- connection: `http://localhost:6669/ (CONNECTED)`
- views: `t = tasks, r = resources`
- controls: `return to task list = esc, toggle pause = space, quit = q`
- Warnings: `⚠ This task has never yielded (14.0094159s)`
- Task details:
  - ID: 22
  - Name: `pre-work`
  - Target: `tokio::task`
  - Location: `simple-console/src/bin/lints.rs:31:10`
  - Total Time: `14.00s`
  - Busy: `14.00s (100.00%)`
  - Scheduled: `0.00ns (0.00%)`
  - Idle: `213.97µs (0.00%)`
- Waker details:
  - Current wakers: `0 (clones: 0, drops: 0)`
  - Woken: `0 times`
- Poll Times Percentiles:
  - p10: `0.00ns`
  - p25: `0.00ns`
  - p50: `0.00ns`
  - p75: `0.00ns`
  - p90: `0.00ns`
  - p95: `0.00ns`
  - p99: `0.00ns`
- Poll Times Histogram: `0`
- Sched Times Percentiles:
  - p10: `0.00ns`
  - p25: `0.00ns`
  - p50: `0.00ns`
  - p75: `0.00ns`
  - p90: `0.00ns`
  - p95: `0.00ns`
  - p99: `0.00ns`
- Scheduled Times Histogram: `0`

## ▶ Summary: Tokio Console

- Built on tracing instrumentation in Tokio
- Console subscriber collects and aggregates
- Tokio console visualizes application state
- Warns about possibly incorrect behavior



# Task Dump

Capture a snapshot of the runtime's state.

## ▶ Dumping tasks

- Records a snapshot of all tasks on a runtime
- Each task includes a backtrace from all "leaf futures"
  - Leaf futures are top most future on the callstack
  - Requires leaf futures to be instrumented
- Like thread dumps (Java) but for Rust!
- Some limitations apply

## ▶ Extra config

```
# Set the RUSTFLAGS variable
RUSTFLAGS="--cfg tokio_unstable --cfg tokio_taskdump" cargo build

# Contents of .cargo/config

[build]
rustflags = ["--cfg", "tokio_unstable", "--cfg", "tokio_taskdump"]
```

### ▶ Creating a task dump

```
let handle = tokio::runtime::Handle::current();

if let Ok(dump) = tokio::time::timeout(Duration::from_secs(2), handle.dump()).await {
    for (i, task) in dump.tasks().iter().enumerate() {
        let trace = task.trace();
        println!("TASK {i}:");
        println!("{trace}\n");
    }
}
```

## ▶ Sleep task dump

- A task that only awaits a sleep
- Backtrace starts at the inner poll

```
tokio::spawn(async {  
    tokio::time::sleep(Duration::from_millis(100))  
        .await;  
});
```

TASK 0:

```
- dump_sleep::main::{{closure}}::{{closure}} at src/bin/dump_sleep.rs:9:56  
└- <tokio::time::sleep::Sleep as core::future::future::Future>::poll at <crates.io>/tokio-1.33.0/src/time/sleep.rs:448:22  
    └- tokio::time::sleep::Sleep::poll_elapsed at <crates.io>/tokio-1.33.0/src/time/sleep.rs:404:16
```

## ▶ Sleep task dump (zoom)

```
- dump_sleep::main::{{closure}}::{{closure}}  
  └─ <tokio::time::sleep::Sleep as core::future::future::Future>::poll  
    └─ tokio::time::sleep::Sleep::poll_elapsed
```



## ▶ Deeper task dump

```
/// In main()
tokio::spawn(async { a().await });

async fn a() { b().await }
async fn b() { c().await }
async fn c() { tokio::time::sleep(Duration::from_millis(100)).await }
```

```
- dump_deeper::main::{{closure}}::{{closure}} at src/bin/dump_deeper.rs:18:13
  └─ dump_deeper::a::{{closure}} at src/bin/dump_deeper.rs:4:9
    └─ dump_deeper::b::{{closure}} at src/bin/dump_deeper.rs:8:9
      └─ dump_deeper::c::{{closure}} at src/bin/dump_deeper.rs:12:52
        └─ <tokio::time::sleep::Sleep as core::future::future::Future>::poll at <crates.io>/tokio-1.33.0/src/time/sleep.rs:448:22
          └─ tokio::time::sleep::Sleep::poll_elapsed at <crates.io>/tokio-1.33.0/src/time/sleep.rs:404:16
```

## ▶ Deeper task dump (zoom)

```
- dump_deeper::main::{{closure}}::{{closure}}
  └─ dump_deeper::a::{{closure}}
    └─ dump_deeper::b::{{closure}}
      └─ dump_deeper::c::{{closure}}
        └─ <tokio::time::sleep::Sleep as core::future::future::Future>::poll
          └─ tokio::time::sleep::Sleep::poll_elapsed
```

### ▶ Select task dump

```
let barrier = Arc::new(Barrier::new(2));
let (_tx, mut rx) = mpsc::channel::<u64>(2);

tokio::spawn(async move {
    // A task awaiting multiple futures will have a branched backtrace
    tokio::select! {
        _ = barrier.wait() => {}
        _ = tokio::time::sleep(Duration::from_millis(100)) => {}
        _ = rx.recv() => {}
    }
});
```

## ▶ Select task dump

```

- dump_select::main::{{closure}}::{{closure}} at src/bin/dump_select.rs:11:9
  ↳ tokio::future::poll_fn::PollFn<F> as core::future::future::Future>::poll at <crates.io>/tokio-1.33.0/src/future/poll_fn.rs:58:9
    # MPSC Channel receive
    | dump_select::main::{{closure}}::{{closure}}::{{closure}} at <crates.io>/tokio-1.33.0/src/macros/select.rs:524:49
    |   ↳ tokio::sync::mpsc::bounded::Receiver<T>::recv::{{closure}} at <crates.io>/tokio-1.33.0/src/sync/mpsc/bounded.rs:230:42
    |     ↳ tokio::future::poll_fn::PollFn<F> as core::future::future::Future>::poll at <crates.io>/tokio-1.33.0/src/future/poll_fn.rs:58:9
    |       ↳ tokio::sync::mpsc::bounded::Receiver<T>::recv::{{closure}}::{{closure}} at <crates.io>/tokio-1.33.0/src/sync/mpsc/bounded.rs:230:22
    |         ↳ tokio::sync::mpsc::chan::Rx<T,S>::recv at <crates.io>/tokio-1.33.0/src/sync/mpsc/chan.rs:246:16
    # Sleep
    | dump_select::main::{{closure}}::{{closure}}::{{closure}} at <crates.io>/tokio-1.33.0/src/macros/select.rs:524:49
    |   ↳ tokio::time::sleep::Sleep as core::future::future::Future>::poll at <crates.io>/tokio-1.33.0/src/time/sleep.rs:448:22
    |     ↳ tokio::time::sleep::Sleep::poll_elapsed at <crates.io>/tokio-1.33.0/src/time/sleep.rs:404:16
    # Barrier wait
    ↳ dump_select::main::{{closure}}::{{closure}}::{{closure}} at <crates.io>/tokio-1.33.0/src/macros/select.rs:524:49
      ↳ tokio::sync::barrier::Barrier::wait::{{closure}} at <crates.io>/tokio-1.33.0/src/sync/barrier.rs:129:10
        ↳ tokio::util::trace::InstrumentedAsyncOp<F> as core::future::future::Future>::poll at <crates.io>/tokio-1.33.0/src/util/trace.rs:81:46
          ↳ tokio::sync::barrier::Barrier::wait_internal::{{closure}} at <crates.io>/tokio-1.33.0/src/sync/barrier.rs:183:36
            ↳ tokio::sync::watch::Receiver<T>::changed::{{closure}} at <crates.io>/tokio-1.33.0/src/sync/watch.rs:715:55
              ↳ tokio::sync::watch::changed_impl::{{closure}} at <crates.io>/tokio-1.33.0/src/sync/watch.rs:881:18
                ↳ tokio::sync::notify::Notified as core::future::future::Future>::poll at <crates.io>/tokio-1.33.0/src/sync/notify.rs:1108:9
                  ↳ tokio::sync::notify::Notified::poll_notified at <crates.io>/tokio-1.33.0/src/sync/notify.rs:100

```

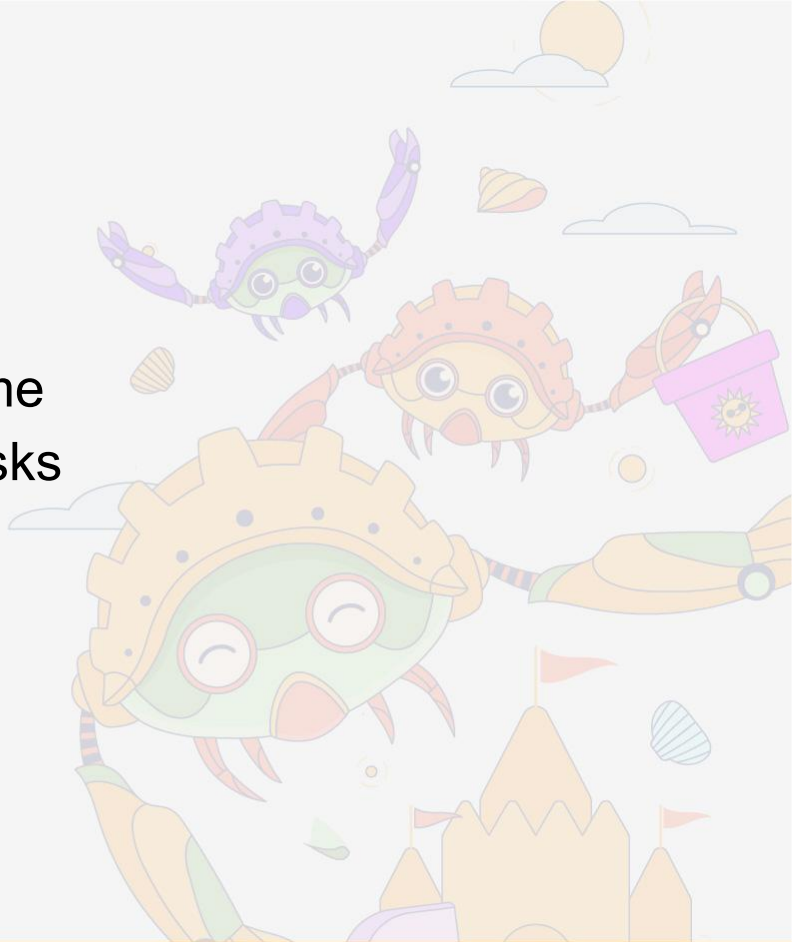
## ► Caveats

- Task dumps are work in progress
- Creating a task dump is expensive
  - But enabling the feature isn't!
- Not much metadata available
  - Task names and IDs
  - Function parameters
- Only supported on Linux



▶ Summary: Task dumps

- New Tokio feature
- Capture a snapshot of a runtime
- Useful for pinpointing stuck tasks



Observing Tokio

# Present & Future

So, what's next?

► Observability tools

<b>Tool</b>	<b>Level of Detail</b>	<b>Use in Production</b>
<b>Metrics</b>	Low	Yes!
<b>Tokio Console</b>	Medium	Careful!
<b>Task Dump</b>	High	Not Yet!



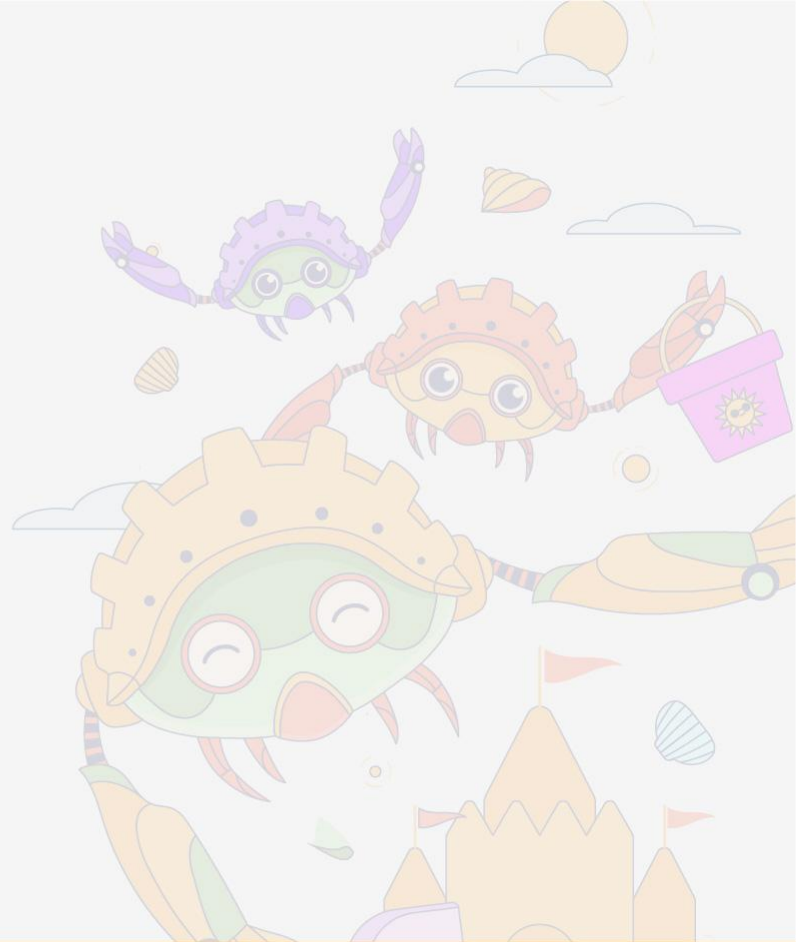
## ▶ The Future: Metrics

- Additional metrics
- Simpler collector integration



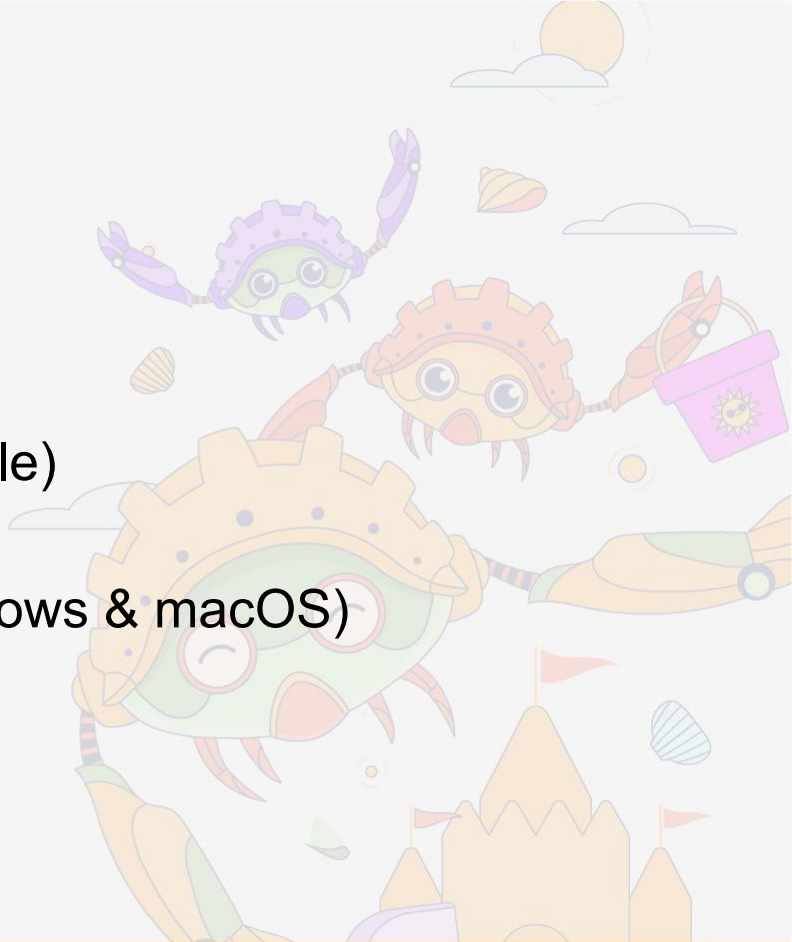
## ▶ The Future: Tokio Console

- More lints
- More usability features
  - Filtering
  - Custom span integration



## ▶ The Future: Task Dump

- Task identifiers
  - Task ID
  - Name (like in Tokio Console)
- More backtrace metadata
- Support more platforms (Windows & macOS)



## ▶ The Future: Ecosystem

- Better integration
  - Metrics and task dumps in Tokio Console
  - Or together elsewhere!
- Standardize runtime instrumentation
  - Instrument other async runtimes

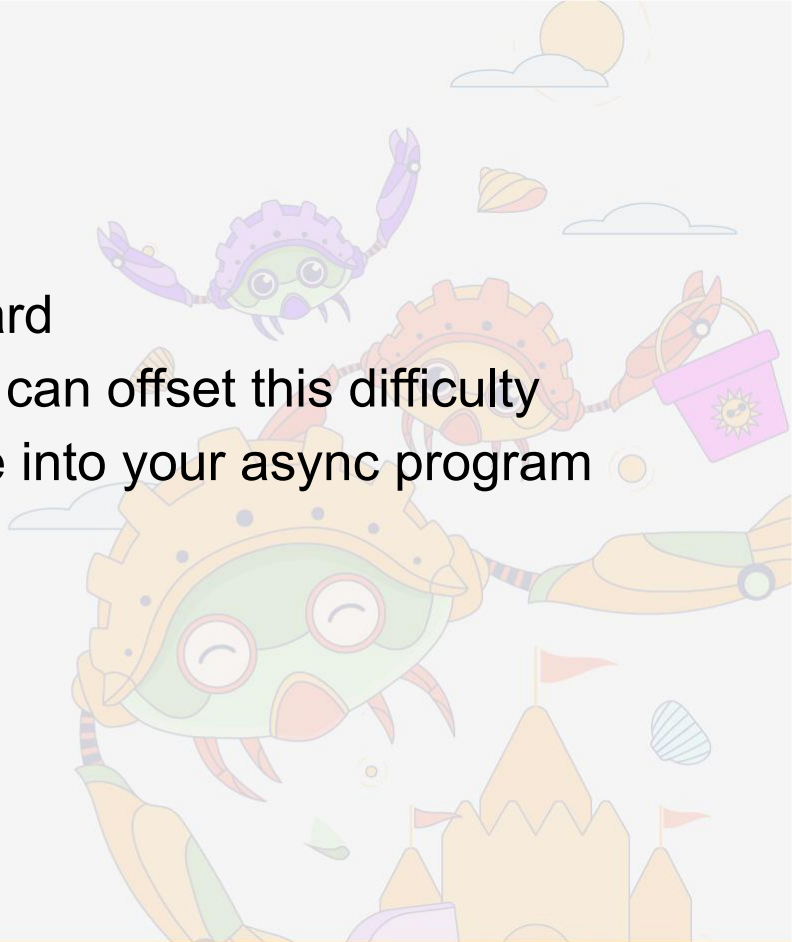
## ▶ The Future: Ecosystem

- You!
  - All these tools are open source
  - Many are developed in maintainer “free time”
  - Want to help?
    - Join us on the [Tokio Discord](#)

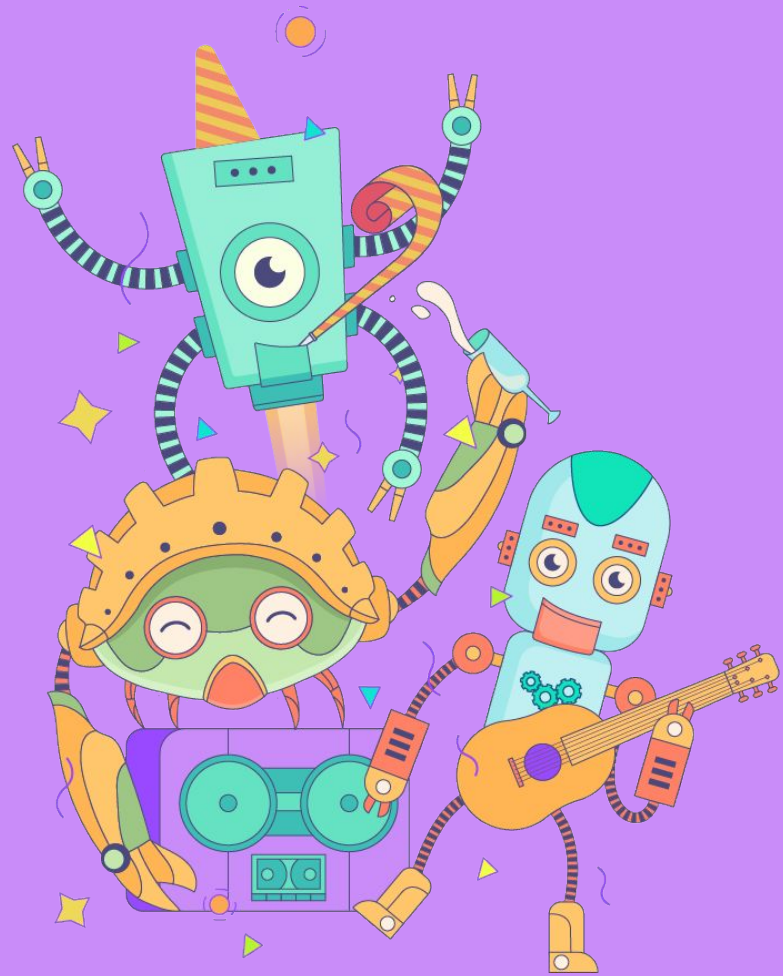


## ► Final thoughts

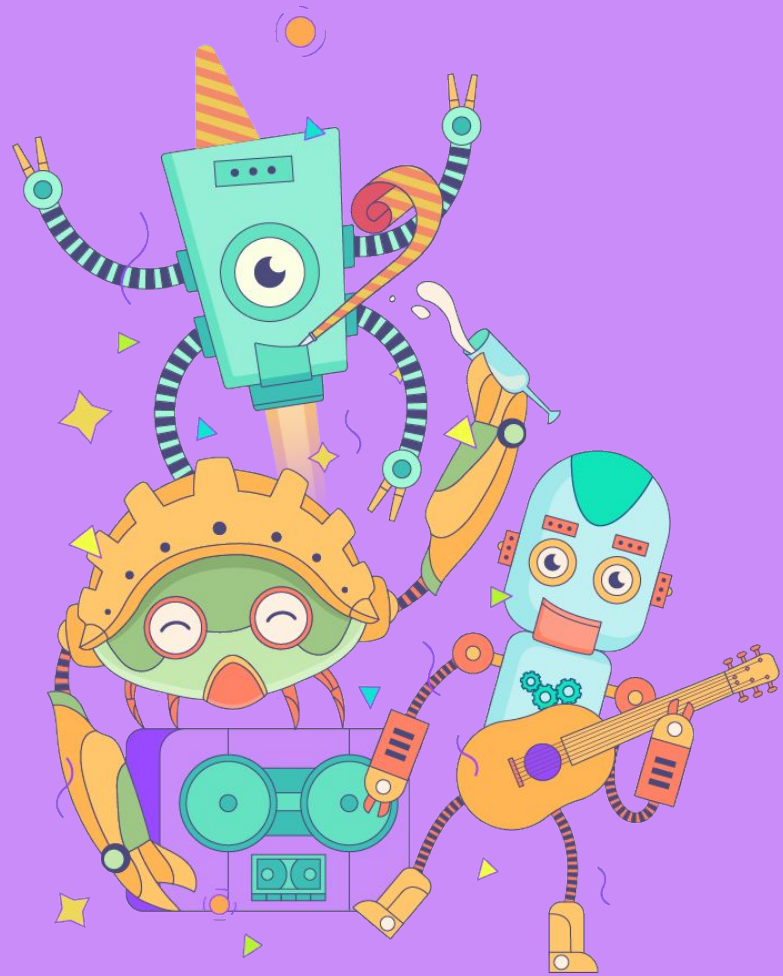
- Async programming can be hard
- Instrumentation of the runtime can offset this difficulty
- There are many options to see into your async program
- We're not there yet



# Thank-you for listening



# Questions?





Links

▶ Tokio

- [Web-site](#)
- [Code](#)
- [docs.rs](#)
- [Discord server](#)



Links

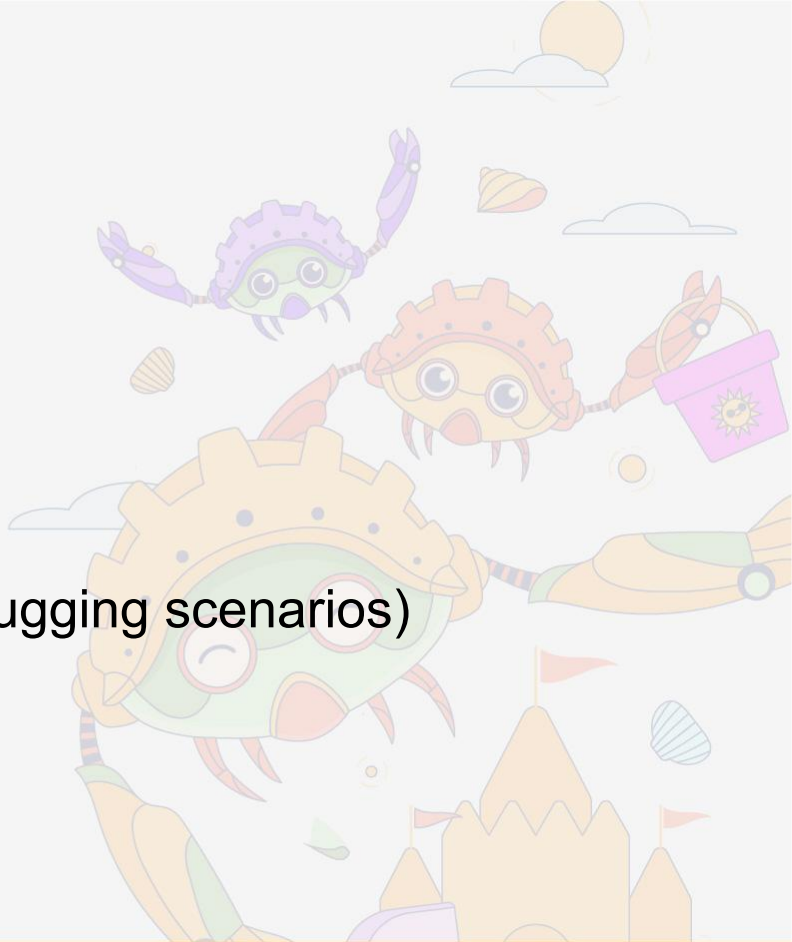
## ▶ Other async runtimes

- [Async-std](#)
- [Smol](#)
- [Embassy](#)



## ▶ Metrics

- Tokio crate
  - [RuntimeMetrics](#)
- [tokio-metrics crate](#)
  - [RuntimeMonitor](#)
  - [TaskMonitor](#) (detailed debugging scenarios)



Links

## ▶ Tokio Console

- [console code](#)
- [console-subscriber on docs.rs](#)
- [tokio-console on docs.rs](#)



Links

## ▶ Task Dump

- [Handle::dump\(\)](#)
- [task dump stabilization issue](#)



Links

## ▶ Hayden Stainsby

- [Email](#)
- [Web-site](#)
- [GitHub](#)
- [Mastodon](#)
- [Rust for Lunch](#)

