Scaling Shiny apps with async programming

Joe Cheng
June 6, 2018

Bringing Shiny apps to production

- **Automated regression testing** for Shiny: shinytest
- New tools for improving **performance & scalability**:
  - Async programming: promises
  - Plot caching (coming soon)
- **Automated load testing** for Shiny: shinyloadtest (coming soon)

Async programming

Sound complicated?

It is!

But when you need it, you **really** need it.

Why would I need it?

R performs tasks one at a time (“single threaded”).

While your Shiny app process is busy doing a long running calculation, it can’t do anything else.

At all.

Example

```
# time = 0:00.000
trainModel(Sonar, "Class")
# time = 0:15.553, ouch!
```

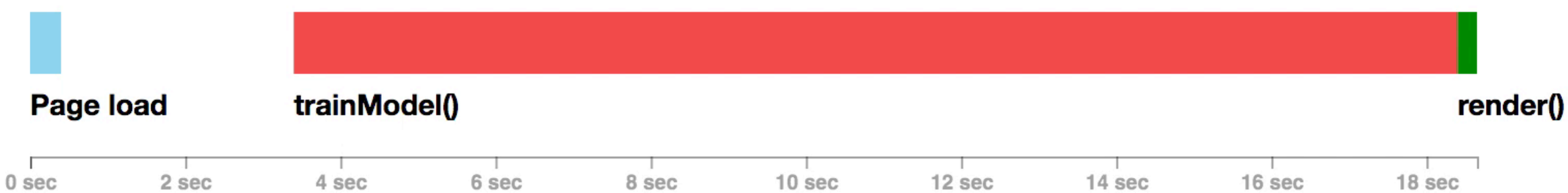
Example

```
ui <- basicPage(
  h2("Synchronous training"),
  actionButton("train", "Train"),
  verbatimTextOutput("summary"),
  plotOutput("plot")
)

server <- function(input, output, session) {
  model <- eventReactive(input$train, {
    trainModel(Sonar, "Class") # Super slow!
  })

  output$summary <- renderPrint({
    print(model())
  })

  output$plot <- renderPlot({
    plot(model())
  })
}
```

Demo

Synchronous

```
# time = 0:00.000
trainModel(Sonar, "Class")
# time = 0:15.553
```



Demo

Async to the rescue

Perform long-running tasks asynchronously: start the task but don’t wait around for the result. This leaves R free to continue doing other things.

We need to:

1. Launch tasks that run **away from the main R thread**

2. Be able to do something with the result (if success) or error (if failure), when the tasks completes, **back on the main R thread**

1. Launch async tasks

```
library(future)
plan(multiprocess)

# time = 0:00.000
f <- future(trainModel(Sonar, "Class"))
# time = 0:00.062
```

Potentially lots of ways to do this, but currently using the **`future`** package by Henrik Bengtsson.

Runs R code in a separate R process, freeing up the original R process.

1. Launch async tasks

```
library(future)
plan(multiprocess)

# time = 0:00.000
f <- future(trainModel(Sonar, "Class"))
# time = 0:00.062
value(f)
# time = 0:15.673
```

However, future's API for **retrieving** values (`value(f)`) is not what we want, as it is blocking: you run tasks asynchronously, but access their results synchronously

2. Do something with the results

The new **`promises`** package lets you access the results from async tasks.

A promise object represents the **eventual result** of an async task. It's an R6 object that knows:

1. Whether the task is running, succeeded, or failed

2. The result (if succeeded) or error (if failed)

Every function that runs an async task, should return a promise object, instead of regular data.

Promises

Directly inspired by JavaScript promises (plus some new features for smoother R and Shiny integration)

They work well with Shiny, but are generic—no part of promises is Shiny-specific

(Not the same as R's promises for delayed evaluation. Sorry about the name collision.)

Also known as tasks (C#), futures (Scala, Python), and CompletableFutures (Java 😂)

How <u>don't</u> promises work?

You **cannot** wait for a promise to finish

You **cannot** ask a promise if it's done

You **cannot** ask a promise for its value

How do promises work?

Instead of extracting the value out of a promise, you *chain* whatever operation you were going to do to the result, to the promise.

Sync (without promises):

```
query_db() %>%
  filter(cyl > 4) %>%
  head(10) %>%
  View()
```

How do promises work?

Instead of extracting the value out of a promise, you *chain* whatever operation you were going to do to the result, to the promise.

Async (with promises):

```
future(query_db()) %...>%
  filter(cyl > 4) %...>%
  head(10) %...>%
  View()
```

The promise pipe operator

```
promise %...>% (function(result) {
  # Do stuff with the result
})
```

The `%...>%` is the “promise pipe”, a promise-aware version of `%>%`.

Its left operand must be a promise (or, for convenience, a Future), and it returns a promise.

You don’t use `%...>%` to pull future values into the present, but to push subsequent computations into the future.

Asynchronous

```
# time = 0:00.000
future(trainModel(Sonar, "Class")) %...>%
  print() # time = 0:15.673
# time = 0:00.062
```

Demo

❌ Sync

```
# time = 0:00.000
trainModel(Sonar, "Class")
# time = 0:15.553
```

❌ Future

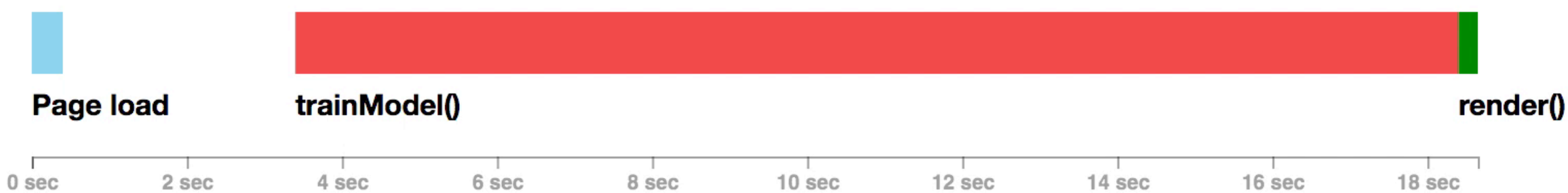
```
# time = 0:00.000
f <- future(trainModel(Sonar, "Class"))
# time = 0:00.062
value(f)
# time = 0:15.673
```

🙌 Future + promises

```
# time = 0:00.000
future(trainModel(Sonar, "Class")) %...>%
  print() # time = 0:15.673
# time = 0:00.062
```

Asynchronous

```
# time = 0:00.000
future(trainModel(Sonar, "Class")) %...>%
  print() # time = 0:15.673
# time = 0:00.062
```



Example 2

```
ui <- basicPage(
  h2("Asynchronous training"),
  actionButton("train", "Train"),
  verbatimTextOutput("summary"),
  plotOutput("plot")
)

server <- function(input, output, session) {
  model <- eventReactive(input$train, {
    future(trainModel(Sonar, "Class")) # So fast!
  })

  output$summary <- renderPrint({
    model() %...>% print()
  })

  output$plot <- renderPlot({
    model() %...>% plot()
  })
}
```

Demo

Current status

- The **`promises`** package is on CRAN
- Documentation at https://rstudio.github.io/promises
- **`shiny`** v1.1.0 is on CRAN, and is required for async apps
- Some downstream packages still need updates for async:

```
ramnathv/htmlwidgets
ropensci/plotly@async
rstudio/shinydashboard@async
rstudio/DT@async
```

A tour of the docs

- Why use promises?
- A gentle introduction to async programming
- Working with promises (API overview)
  - Additional promise operators
  - Error handling (promise equivalents to try, catch, finally)
- Launching tasks (a guide to using the **future** package)
- Using promises with Shiny
- Composing promises and working with collections of promises

Case study: **cranwhales**

Source: https://github.com/rstudio/cranwhales
Live: https://gallery.shinyapps.io/cranwhales

“As a web service increases in popularity, so does the number of rogue scripts that abuse it for no apparent reason.”

–Cheng’s Law of Why We Can’t Have Nice Things

Motivation

- RStudio runs the popular cloud.r-project.org CRAN mirror
- Who are the top downloaders each day?
  - What countries are they from?
  - How many downloads?
  - What packages?
  - Interesting access patterns?

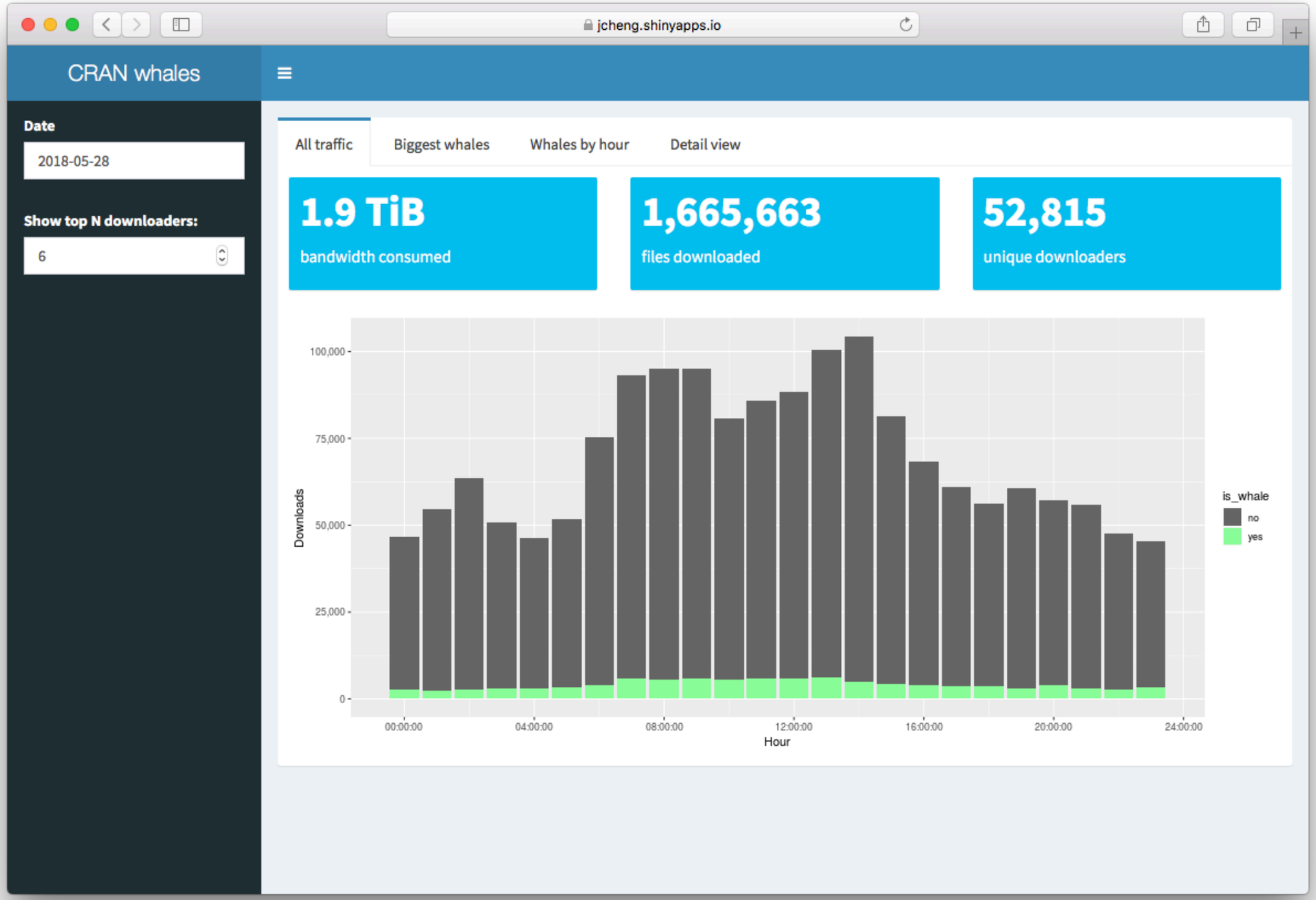
Data source

- RStudio CRAN mirror log files, available as gzipped CSV files at: http://cran-logs.rstudio.com/
- One log file for each day
- One row per download
- Anonymized IP addresses (each IP is converted to integer that is unique for the day)
- On a recent day (May 28, 2018):
  - 1,665,663 rows (downloads)
  - 23.4 MB download size, 137 MB uncompressed

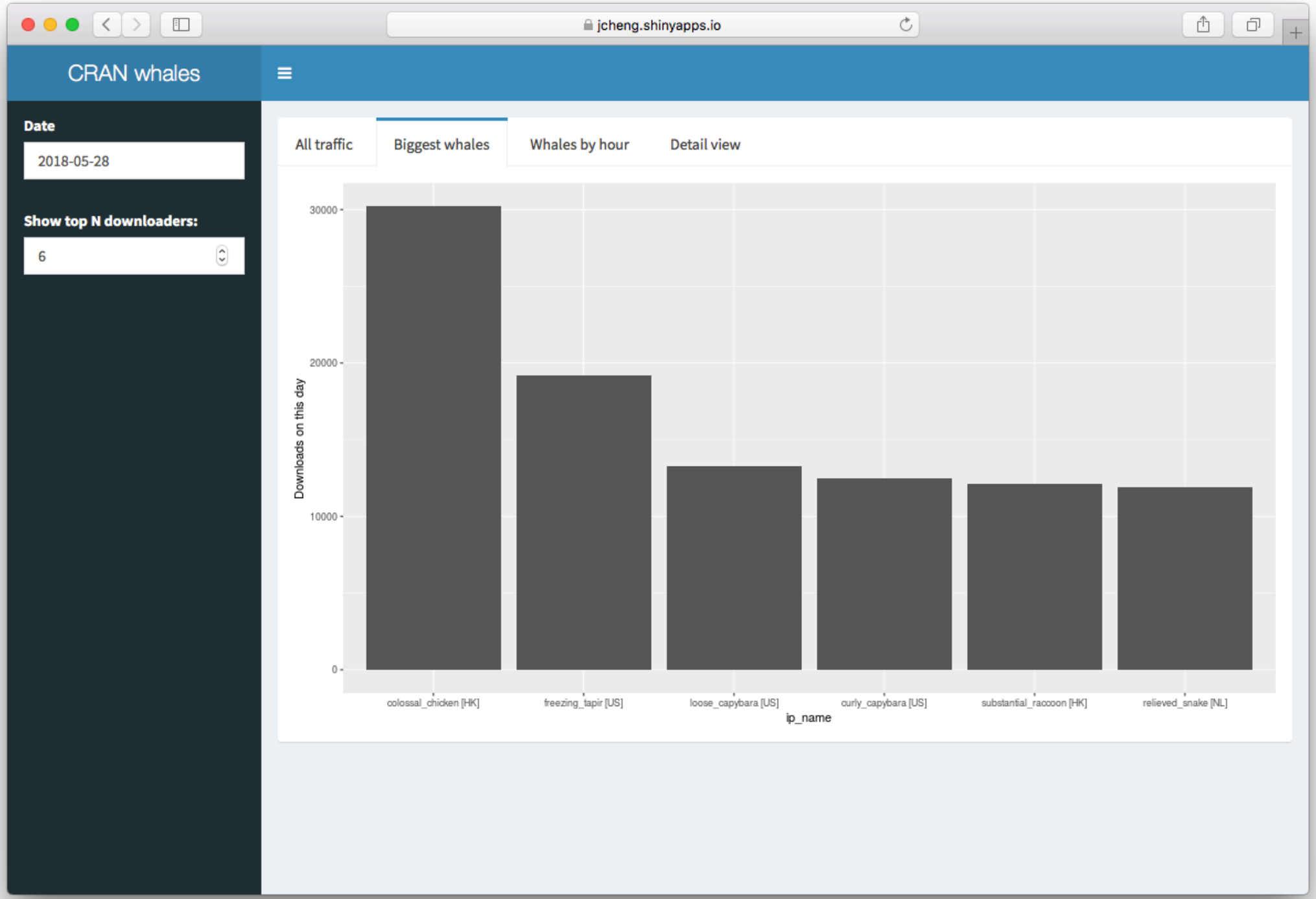
Data source

	date	time	size	r_version	r_arch	r_os	package	version	country	ip_id
1	2018-05-28	20:49:32	1061563	NA	NA	NA	jsonlite	1.0	NL	1
2	2018-05-28	20:49:39	1088934	3.3.3	x86_64	linux-gnu	zoo	1.8-1	AT	2
3	2018-05-28	20:49:40	5217525	3.4.4	x86_64	darwin15.6.0	caret	6.0-80	HK	3
4	2018-05-28	20:49:36	48598	3.5.0	x86_64	mingw32	whisker	0.3-2	US	4
5	2018-05-28	20:49:33	76419	3.5.0	x86_64	mingw32	glue	1.2.0	BR	5
6	2018-05-28	20:49:33	22597	3.5.0	x86_64	mingw32	pkgconfig	2.0.1	BR	5
7	2018-05-28	20:49:33	335282	3.5.0	x86_64	mingw32	R6	2.2.2	BR	5
8	2018-05-28	20:49:37	4507272	3.5.0	x86_64	mingw32	Rcpp	0.12.17	BR	5
9	2018-05-28	20:49:38	622343	3.4.1	x86_64	darwin15.6.0	tidyselect	0.2.4	BR	5
10	2018-05-28	20:49:44	4343639	3.3.3	x86_64	linux-gnu	LaplacesDemon	16.1.0	US	6
11	2018-05-28	20:49:39	2796780	3.4.4	x86_64	darwin15.6.0	HSAUR2	1.1-17	US	7
12	2018-05-28	20:49:36	122754	3.3.2	x86_64	darwin13.4.0	withr	2.1.2	US	4
13	2018-05-28	20:49:43	528	3.4.4	x86_64	linux-gnu	rlang	0.1.6	DE	8
14	2018-05-28	20:49:44	564001	3.4.4	x86_64	linux-gnu	crosstalk	1.0.0	HK	9
15	2018-05-28	20:49:45	1052067	3.4.4	x86_64	linux-gnu	GGally	1.4.0	HK	9
16	2018-05-28	20:49:47	1390861	3.3.3	i386	mingw32	webshot	0.5.0	HK	9
17	2018-05-28	20:49:40	2965818	3.3.3	i386	mingw32	curl	3.2	CA	10
18	2018-05-28	20:49:43	3572998	3.3.3	i386	mingw32	openssl	1.0.1	CA	10
19	2018-05-28	20:49:45	302589	3.3.3	i386	mingw32	httr	1.3.1	CA	10
20	2018-05-28	20:49:46	30082	3.3.3	i386	mingw32	memoise	1.1.0	CA	10
21	2018-05-28	20:49:46	65157	3.3.3	x86_64	linux-gnu	whisker	0.3-2	CA	10

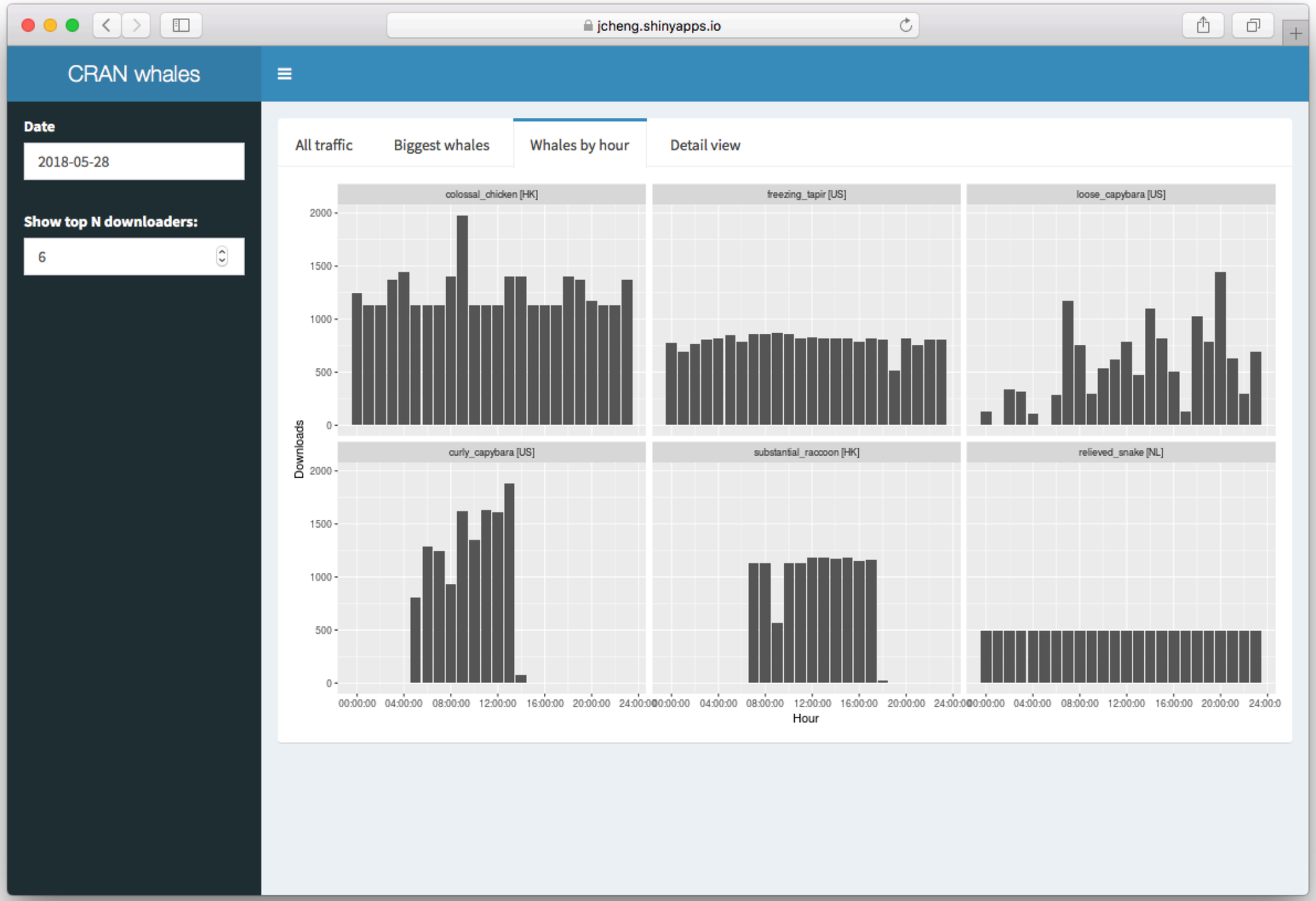
A tour of the app



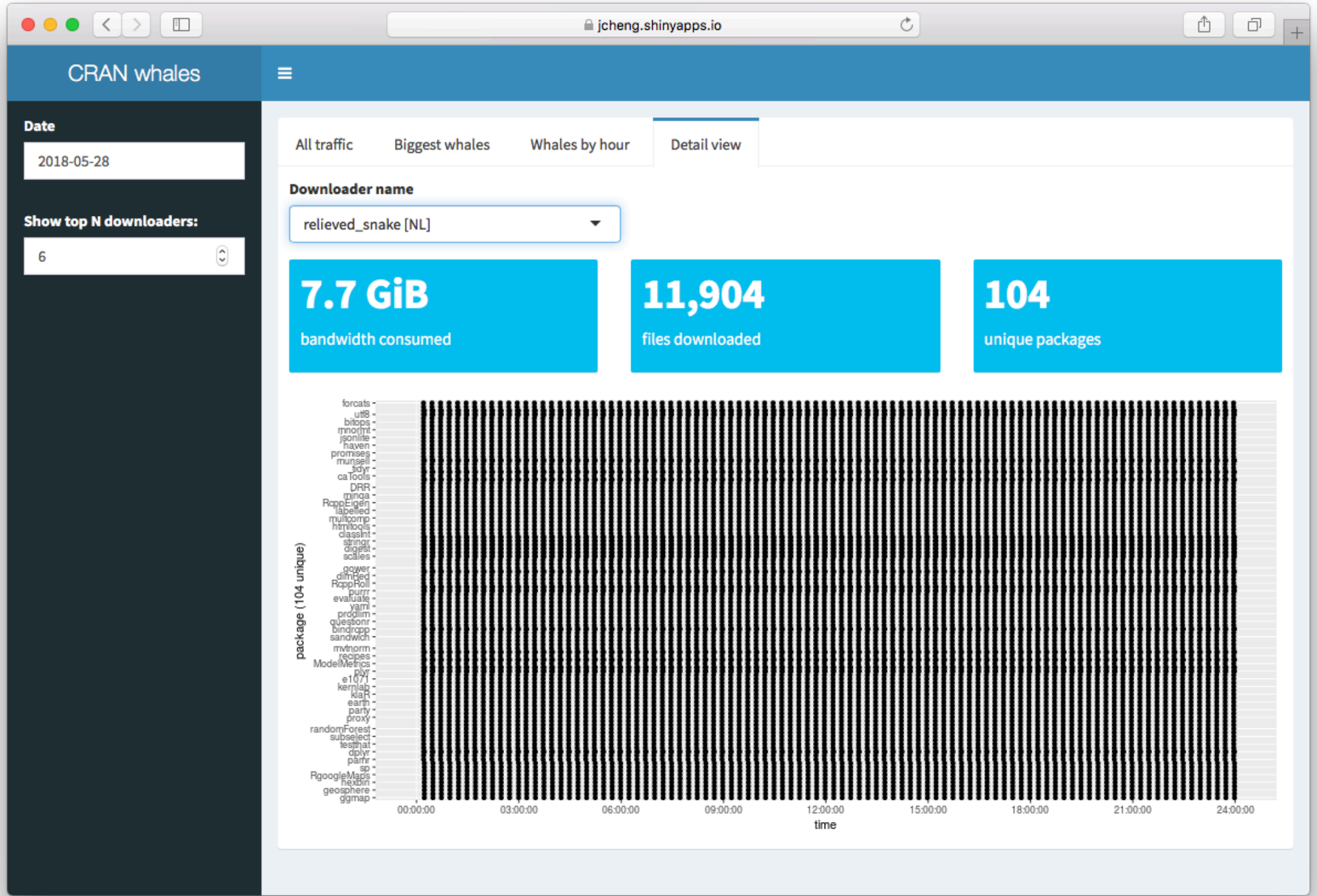
A tour of the app



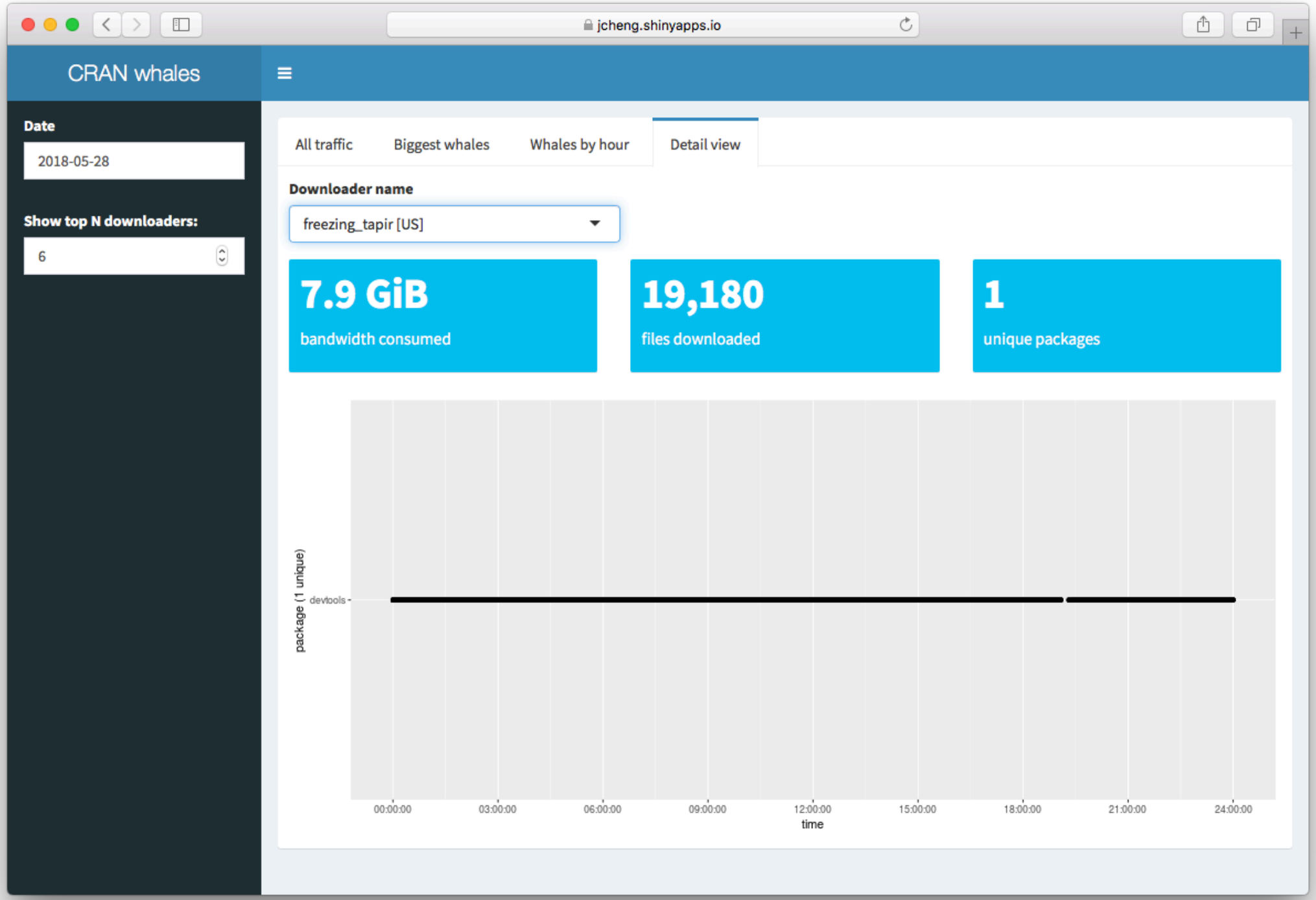
A tour of the app



A tour of the app



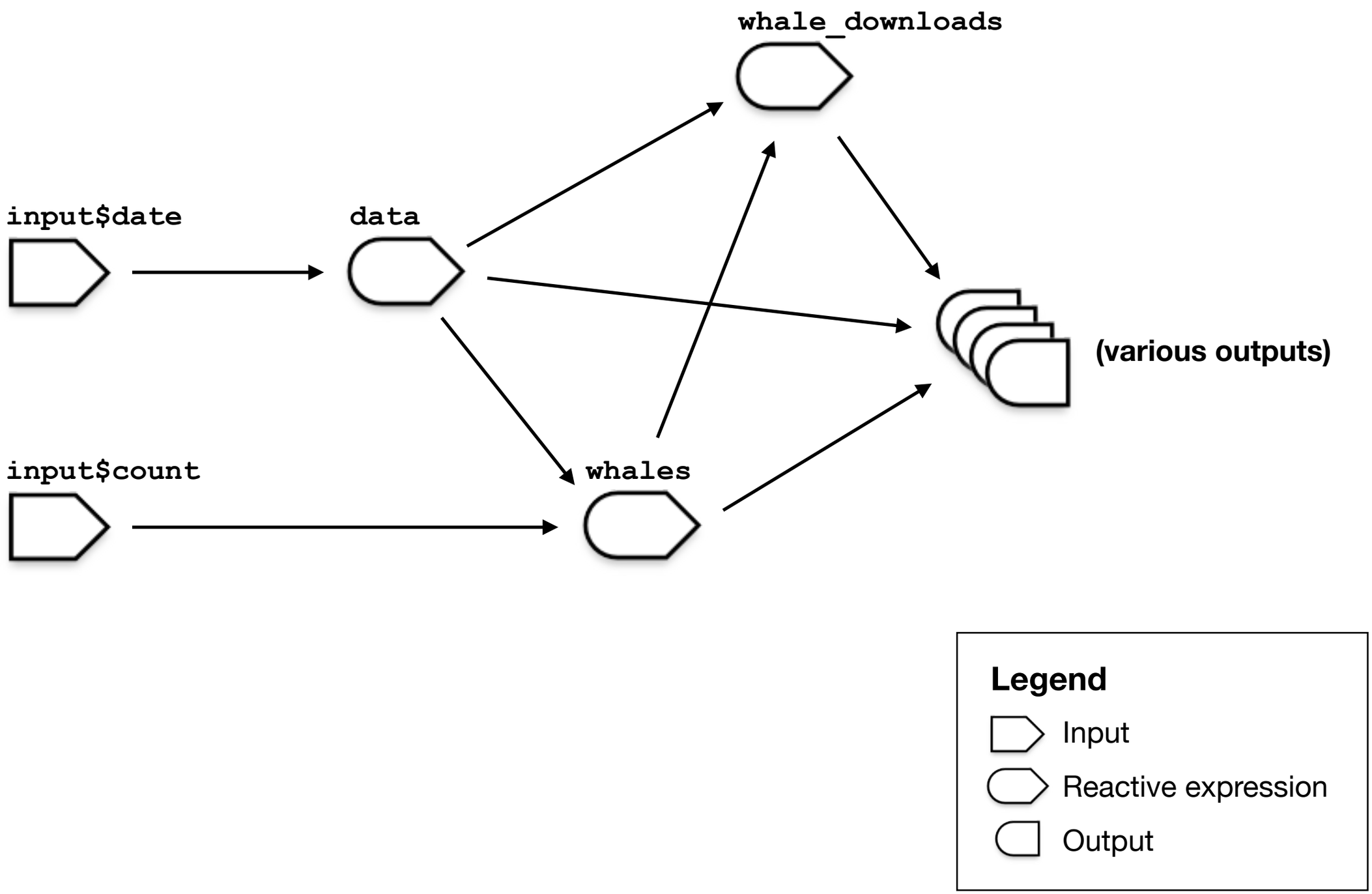
A tour of the app



A tour of the app

- Three main reactive expressions: **`data`**, **`whales`**, and **`whale_downloads`**
  - **`data`** is the raw data for the current day
  - **`whales`** is the top **`input$count`** downloaders. It returns the columns **`ip_id`**, **`ip_name`** (randomly generated) and **`country`**.
  - **`whale_downloads`** has the same columns as **`data`**, but the rows are filtered down to only include whales
- Side note: We'll purposely do minimal caching, to isolate the impact of async (within reason)

Reactive graph



Converting to async

1. Identify slow operations using **`profvis`**

2. Convert slow operations to async using the **`future`** package

3. Any code that was using the result of that operation, now needs to handle a promise (and any code that was using that code needs to handle a promise… etc…)

(Source: *Using promises with Shiny*)

Converting to async

1. Identify slow operations using **`profvis`**
2. Convert slow operations to async using the **`future`** package
3. Any code that was using the result of that operation, now needs to handle a promise (and any code that was using that code needs to handle a promise… etc…)

The `data` reactive: sync

```
data <- eventReactive(input$date, {
  date <- input$date  # Example: 2018-05-28
  year <- lubridate::year(date)  # Example: "2018"

  url <- glue("http://cran-logs.rstudio.com/{year}/{date}.csv.gz")
  path <- file.path("data_cache", paste0(date, ".csv.gz"))

  if (!file.exists(path)) {
    download.file(url, path)
  }

  read_csv(path, col_types = "Dti---c-ci", progress = FALSE)
})
```

Converting to async

1. Identify slow operations using `profvis`
2. Convert slow operations to async using the `future` package
3. Any code that was using the result of that operation, now needs to handle a promise (and any code that was using that code needs to handle a promise… etc…)

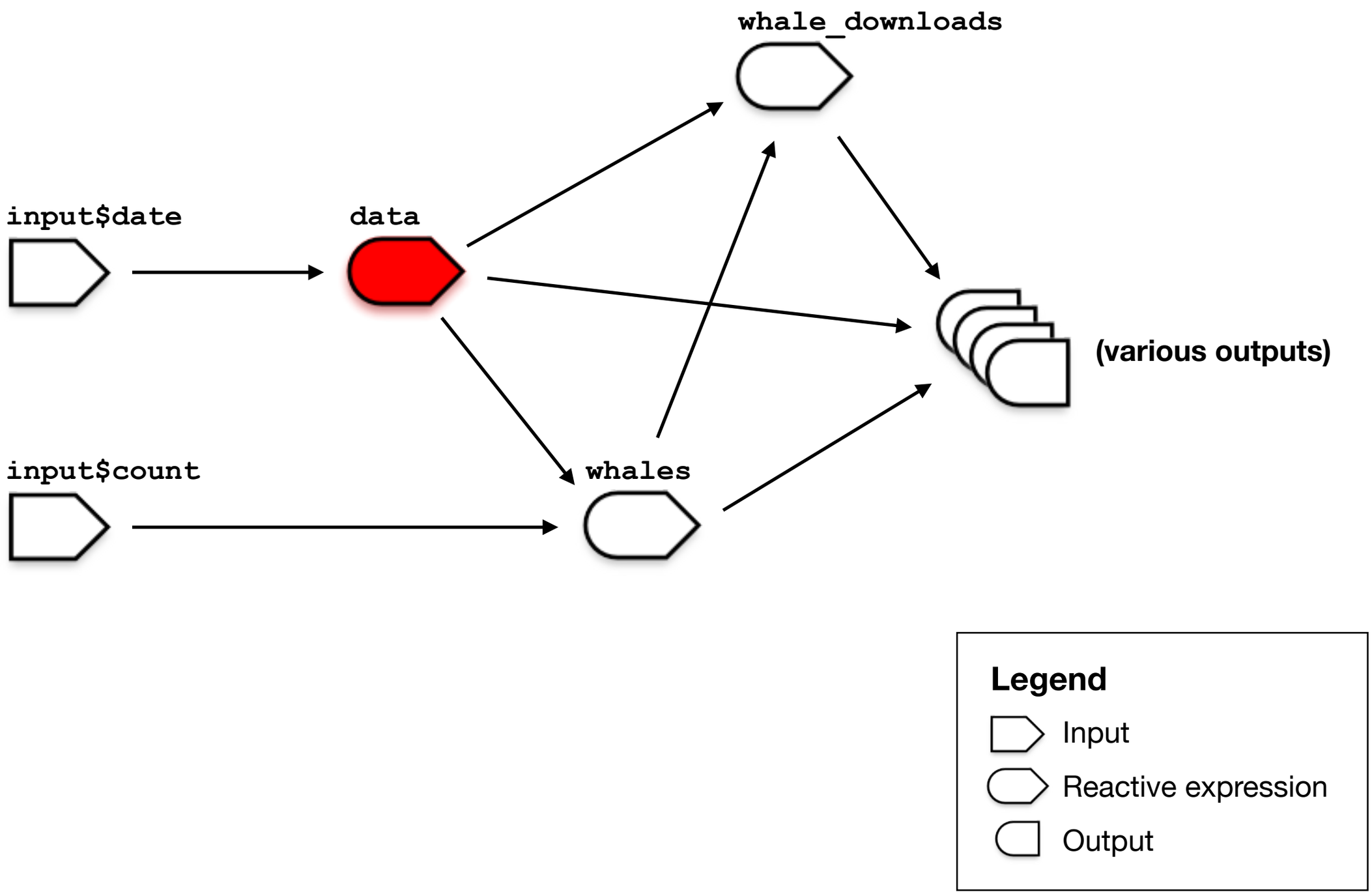
The `data` reactive: async

```
data <- eventReactive(input$date, {
  date <- input$date  # Example: 2018-05-28
  year <- lubridate::year(date)  # Example: "2018"

  url <- glue("http://cran-logs.rstudio.com/{year}/{date}.csv.gz")
  path <- file.path("data_cache", paste0(date, ".csv.gz"))

  future({
    if (!file.exists(path)) {
      download.file(url, path)
    }

    read_csv(path, col_types = "Dti---c-ci", progress = FALSE)
  })
})
```

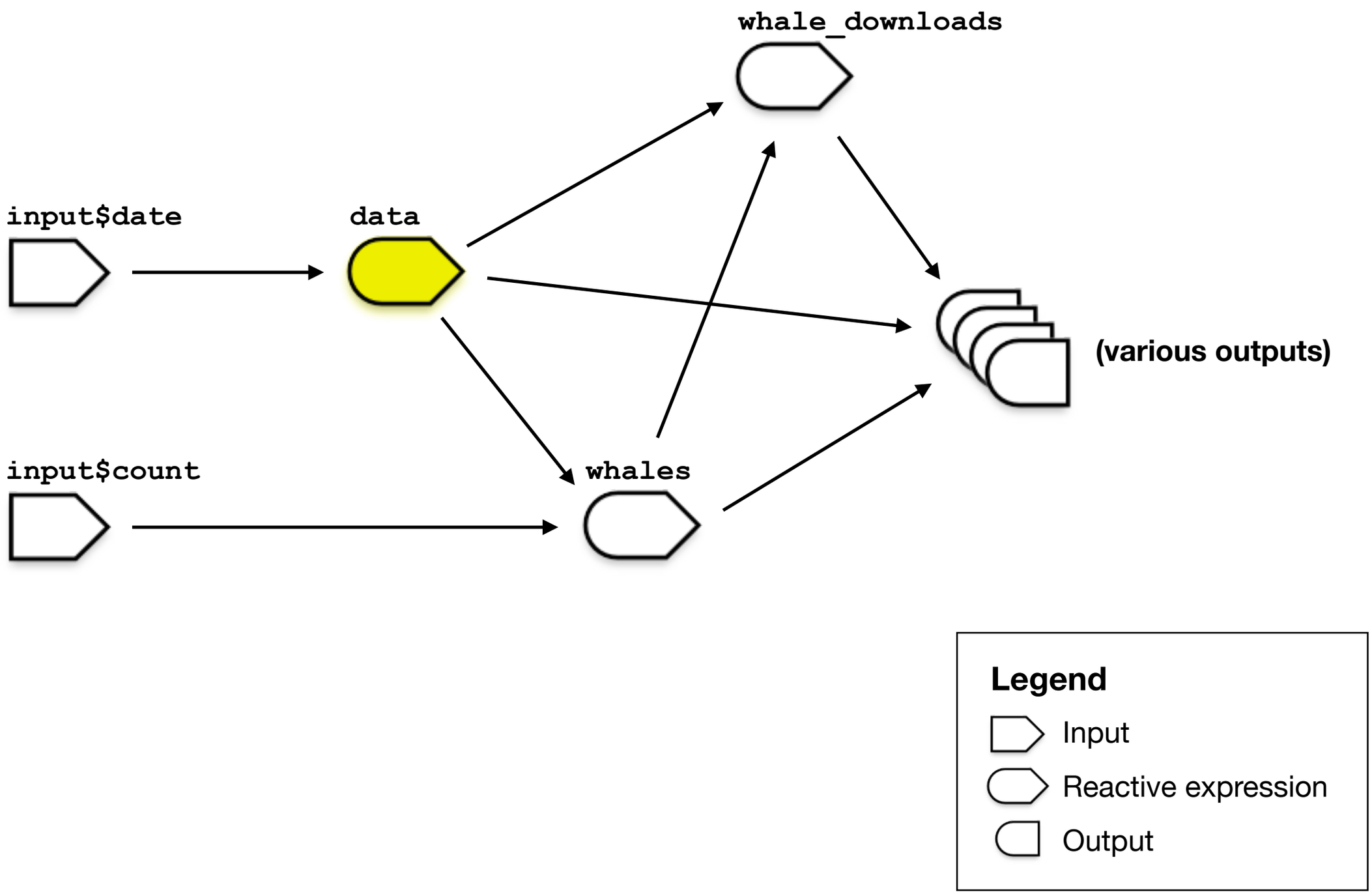
Converting to async

1. Identify slow operations
2. Convert slow operations to async using the `future` package
3. Any code that was using the result of that operation, now needs to handle a promise (and any code that was using *that* code needs to handle a promise… etc…)

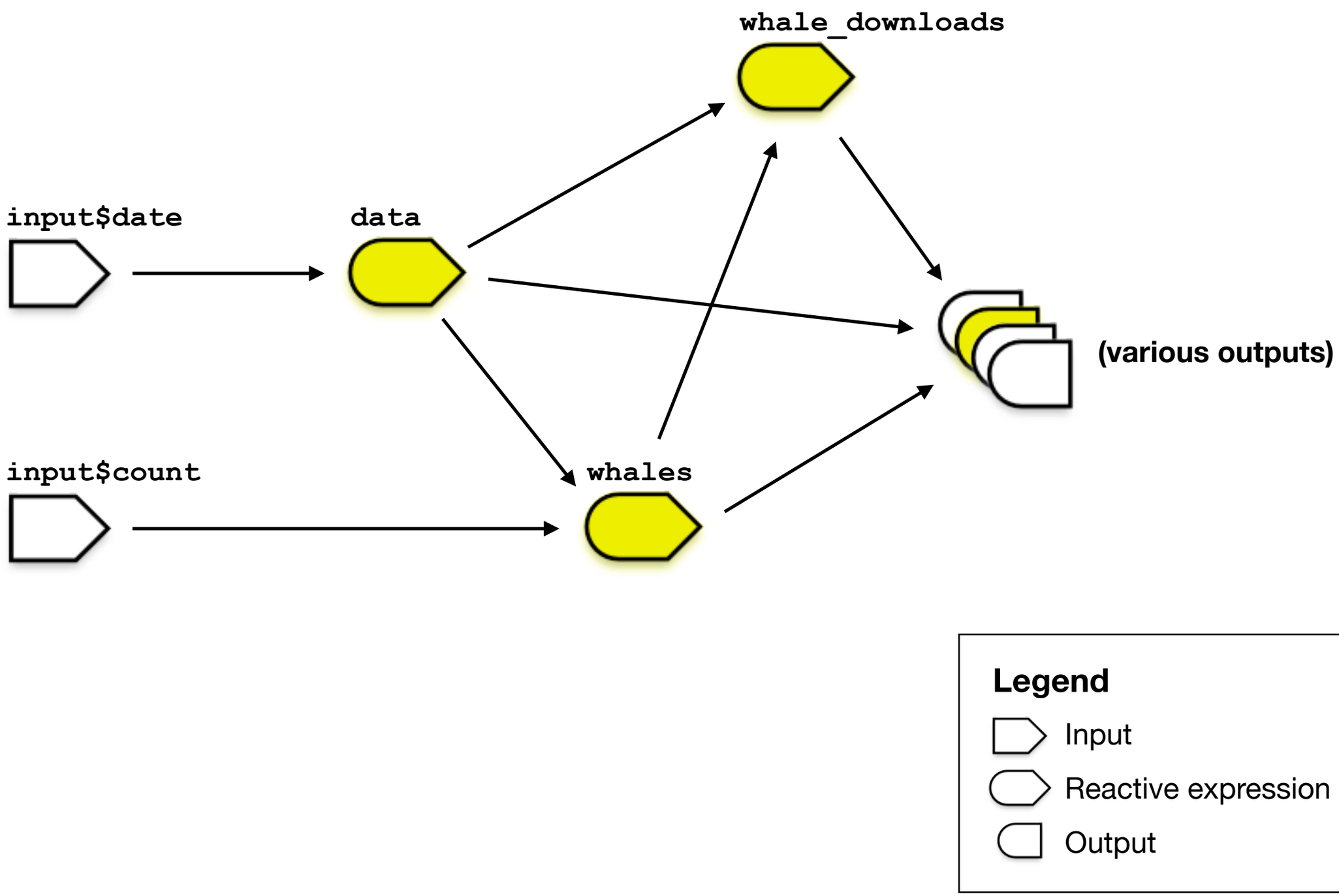
Reactive graph



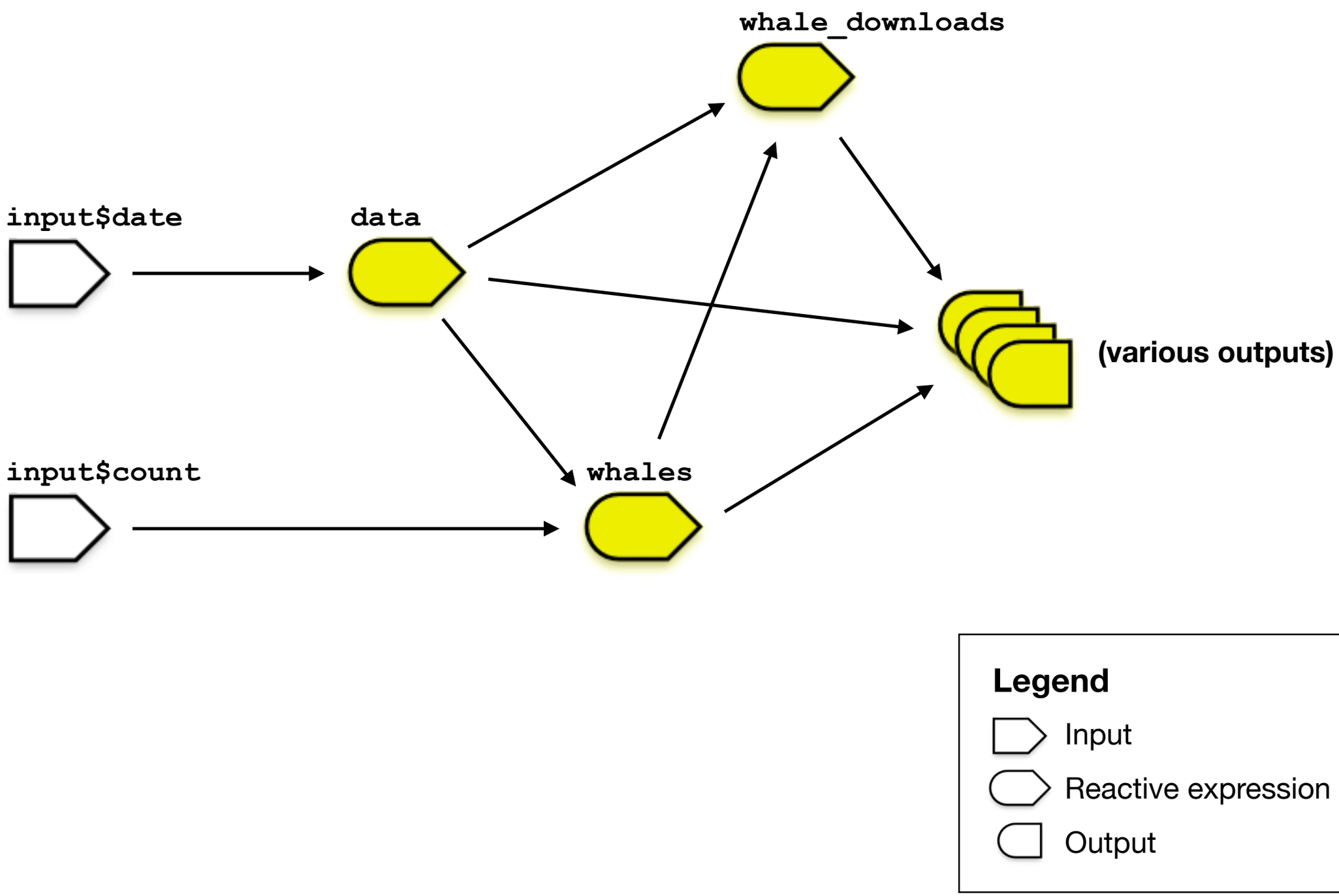
Reactive graph



Reactive graph



Reactive graph



The `whales` reactive: sync

```
whales <- reactive({
  data() %>%
    count(ip_id) %>%
    arrange(desc(n)) %>%
    head(input$count)
})
```

The `whales` reactive: async

```
whales <- reactive({
  data() %...>%
    count(ip_id) %...>%
    arrange(desc(n)) %...>%
    head(input$count)
})
```

Pattern 1: promise pipe

- As simple as find-and-replace
- Only works if the promise object is at the head of the pipeline
- Only works if you are only dealing with one promise object at a time
- Surprisingly common—applied to 59% of reactive objects in this app

The `whale_downloads` reactive: sync

```
whale_downloads <- reactive({
  data() %>%
    inner_join(whales(), "ip_id") %>%
    select(-n)
})
```

The `whale_downloads` reactive: async

```
whale_downloads <- reactive({
  data() %...>%
    inner_join(whales(), "ip_id") %...>%
    select(-n)
})
```

The `whale_downloads` reactive: async

```
whale_downloads <- reactive({
  promise_all(d = data(), w = whales()) %...>% with({
    d %>%
      inner_join(w, "ip_id") %>%
      select(-n)
  })
})
```

Pattern 2: gather

- Necessary when you have multiple promises
- Use **`promise_all`** to wait for all input promises
- **`promise_all`** returns a promise that succeeds when all its input promises succeed; its value is a named list
- Use **`with`** to make the resulting list's elements available as variable names

ggplot2 outputs: sync

```
output$downloaders <- renderPlot({
  whales() %>%
    ggplot(aes(ip_name, n)) +
    geom_bar(stat = "identity") +
    ylab("Downloads on this day")
})
```

ggplot2 outputs: async

```
output$downloaders <- renderPlot({
  whales() %...>% {
    whales_df <- .
    ggplot(whales_df, aes(ip_name, n)) +
    geom_bar(stat = "identity") +
    ylab("Downloads on this day”)
  }
})
```

Pattern 3: promise pipe + code block

- Inside the code block, the “dot” is the result of the promise
- More flexibility than simple pipeline, which is needed when working with “untidy” functions, or if your result object needs to be used somewhere besides the first argument
- Very useful for regular (non-async) `%>%` operators too

Complete diff

GitHub, Inc.

```
94        need(input$count <= 25, "Too many downloaders; 25 or fewer please")
95      )
96  -   data() %>%
97  -     count(ip_id, country) %>%
98  -     arrange(desc(n)) %>%
99  -     head(input$count) %>%
100       mutate(ip_name = factor(ip_id, levels = ip_id,
101 -       labels = glue("{random_name(length(ip_id), input$date)} [{country}]"))) %>%
102       select(-country)
103   })
104
105   # data(), filtered down to the downloads that are by the top `input$count`
106   # downloaders
107   whale_downloads <- reactive({
108 -   data() %>%
109 -     inner_join(whales(), "ip_id") %>%
110 -     select(-n)
111   })
112
113
```

```
@@ -116,77 +123,82 @@ server <- function(input, output, session) {
116   #### "All traffic" tab ---------------------------------------
117
118   output$total_size <- renderValueBox({
119 -   data() %>%
120 -     pull(size) %>%
121 -     as.numeric() %>%  # Cast from integer to numeric to avoid overflow warning
122 -     sum() %>%
123 -     humanReadable() %>%
124       valueBox("bandwidth consumed")
125   })
126
127   output$total_count <- renderValueBox({
128 -   data() %>%
129 -     nrow() %>%
130 -     format(big.mark = ",") %>%
131       valueBox("files downloaded")
132   })
133
134   output$total_uniques <- renderValueBox({
135 -   data() %>%
```

```
98        need(input$count <= 25, "Too many downloaders; 25 or fewer please")
99      )
100 +   data() %...>%
101 +     count(ip_id, country) %...>%
102 +     arrange(desc(n)) %...>%
103 +     head(input$count) %...>%
104       mutate(ip_name = factor(ip_id, levels = ip_id,
105 +       labels = glue("{random_name(length(ip_id), input$date)} [{country}]"))) %...>%
106       select(-country)
107   })
108
109   # data(), filtered down to the downloads that are by the top `input$count`
110   # downloaders
111   whale_downloads <- reactive({
112 +   promise_all(data = data(), whales = whales()) %...>%
113 +     with({
114 +       data %>%
115 +         inner_join(whales, "ip_id") %>%
116 +         select(-n)
117 +     })
118   })
119
120
123   #### "All traffic" tab ---------------------------------------
124
125   output$total_size <- renderValueBox({
126 +   data() %...>%
127 +     pull(size) %...>%
128 +     as.numeric() %...>%  # Cast from integer to numeric to avoid overflow warning
129 +     sum() %...>%
130 +     humanReadable() %...>%
131       valueBox("bandwidth consumed")
132   })
133
134   output$total_count <- renderValueBox({
135 +   data() %...>%
136 +     nrow() %...>%
137 +     format(big.mark = ",") %...>%
138       valueBox("files downloaded")
139   })
140
141   output$total_uniques <- renderValueBox({
142 +   data() %...>%
```

Measuring performance: Did async help?

Load testing Shiny apps

- Shiny applications work using a combination of HTTP requests (to load the app's HTML page, plus various CSS/JavaScript files) and WebSockets (for communicating inputs/outputs)

- Because of WebSockets, custom tools are needed for load testing

- `shinyloadtest` tools (coming soon):

  - **Record** yourself using the app (resulting in HTTP and WebSocket traffic)

  - Then **play back** those same actions against a server, multiplied by X

  - **Analyze** the timings generated by the playback

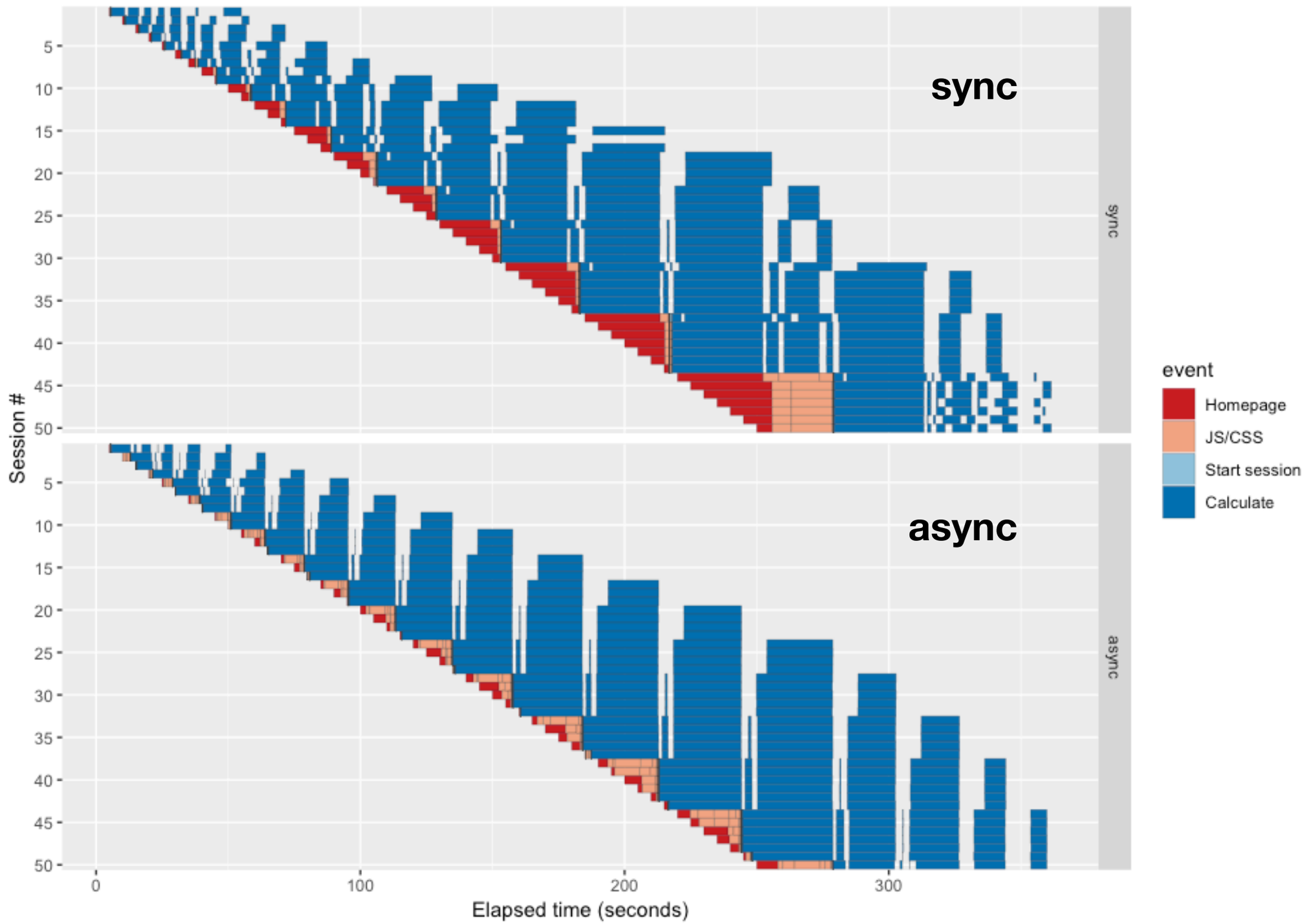
Measuring performance

- **Reducing HTTP times** is especially important, as these reflect the initial page load time. Users are much more sensitive to latency here!

- I recorded a **40 second test script**, and for each test, played it back **50 times**, with a **5 second wait** between each start time.

- Tested against a single R process; everything running on my MacBook Pro

Initial results



Mixed results

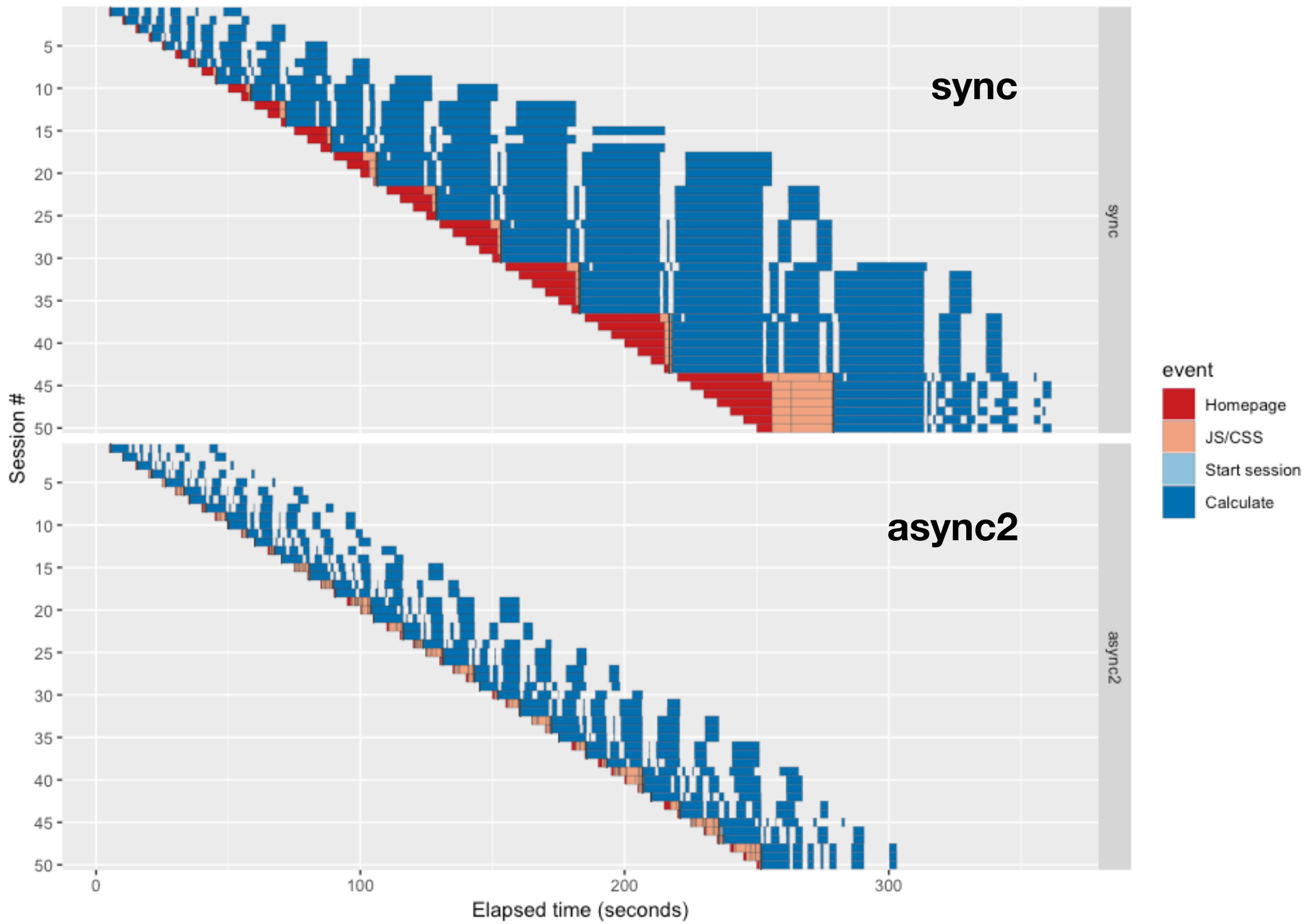
- **The Good:** HTTP latency significantly reduced = faster initial load times
- **The Bad:** WebSocket latency has not improved, might even be worse

Why isn’t the async version faster?

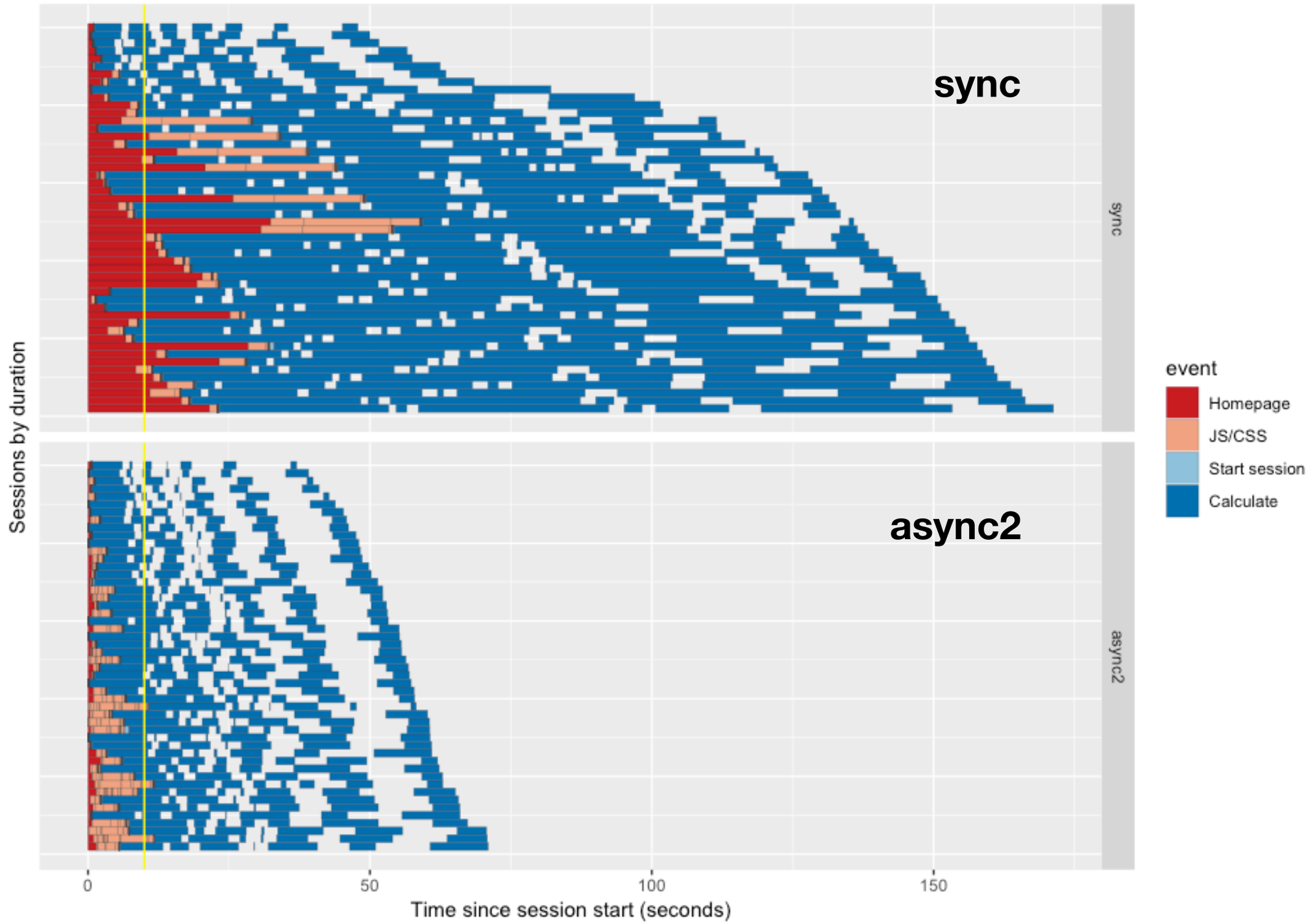
Futures have their own overhead

- Async futures run in separate R processes
- Each future's result value must be copied back to the parent (Shiny) process, and part of this happens while blocking the parent process
  - This copying can be as time consuming as the **`read_csv`** operation we're trying to offload!
- We can reduce the overhead by doing more work in the future, and returning less data back to the parent
  https://github.com/rstudio/cranwhales/compare/async...async2

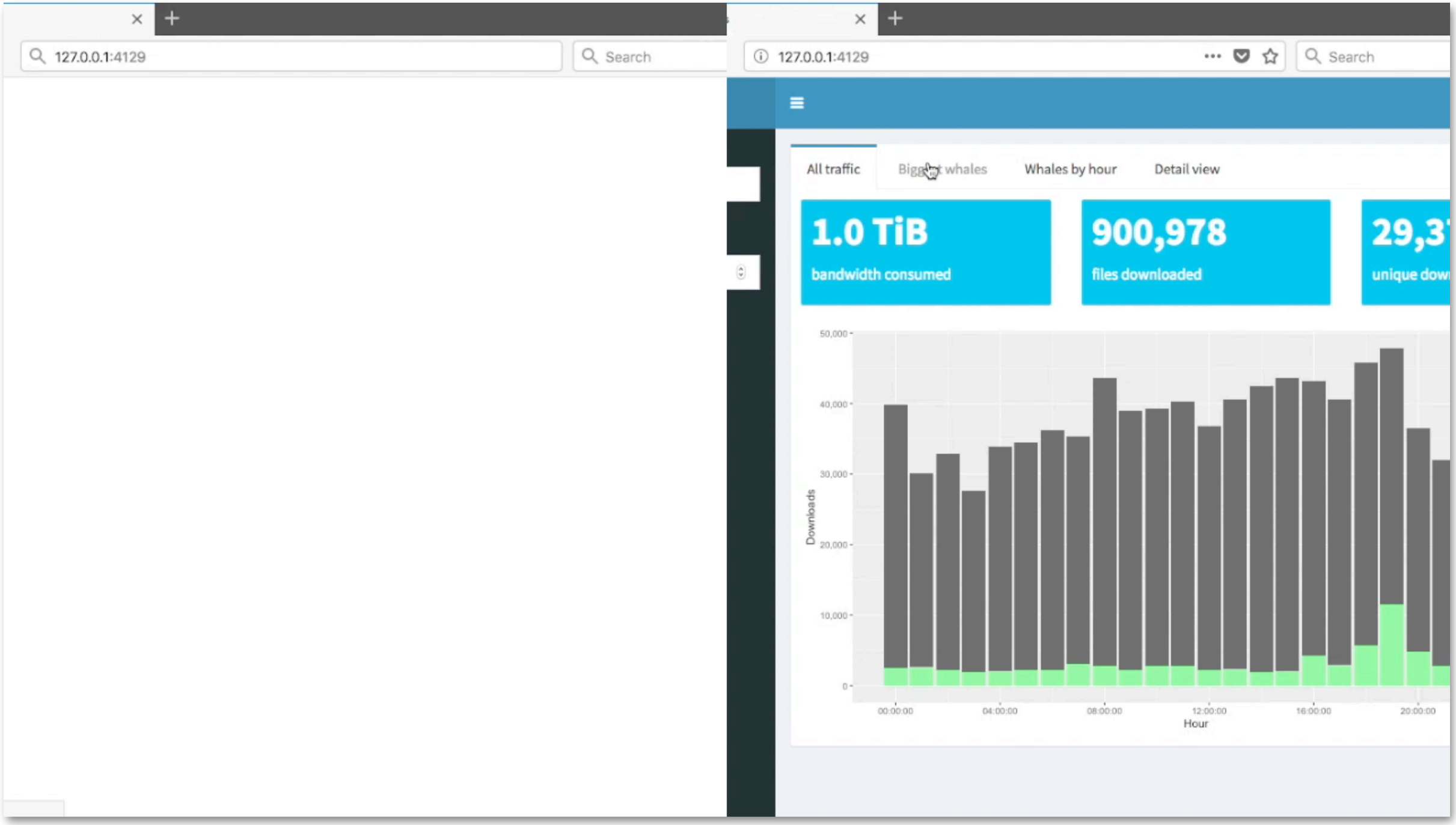
New results


sync
async2
Session #
Elapsed time (seconds)
event
Homepage
JS/CSS
Start session
Calculate

New results (left-aligned, sorted by duration)



Head to head comparison (video link)



Limitations of async

- Few advantages for single sessions (i.e. no concurrency)
  - Latency doesn’t decrease
  - Not specifically intended to let you interact with the app while other tasks for your session proceed in the background (details)—but I’ll publish workarounds soon

Limitations of async

- Other techniques can have much more dramatic impact on performance, for both single and multiple sessions
  - Precompute (summarize/aggregate/filter) ahead of time and save the results (i.e. Extract-Transform-Load)
  - Cache results when possible

Thank you

https://speakerdeck.com/jcheng5/async-webinar