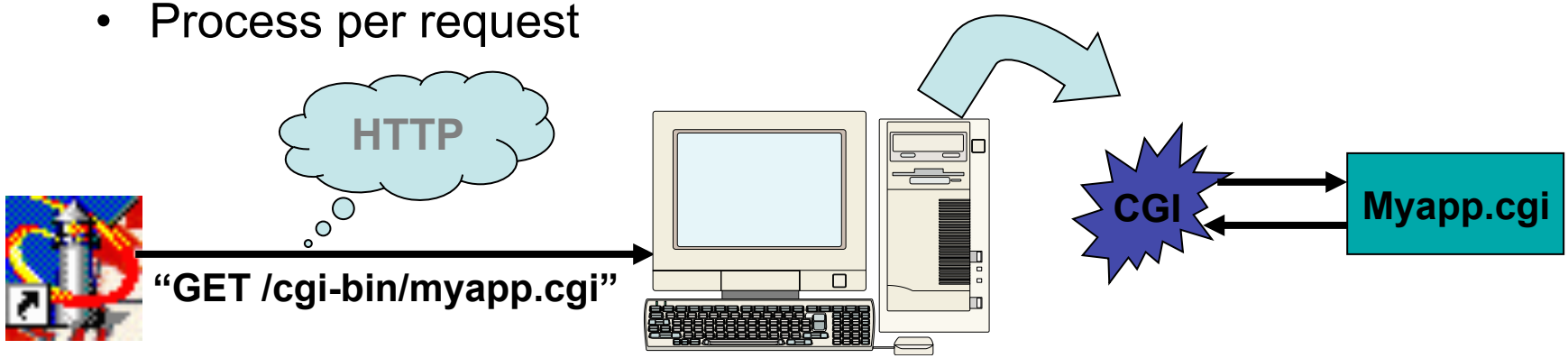
Web Runtime Architecture Revisited

Making the case for NodeJS

Recall our Web Application Architectures to date

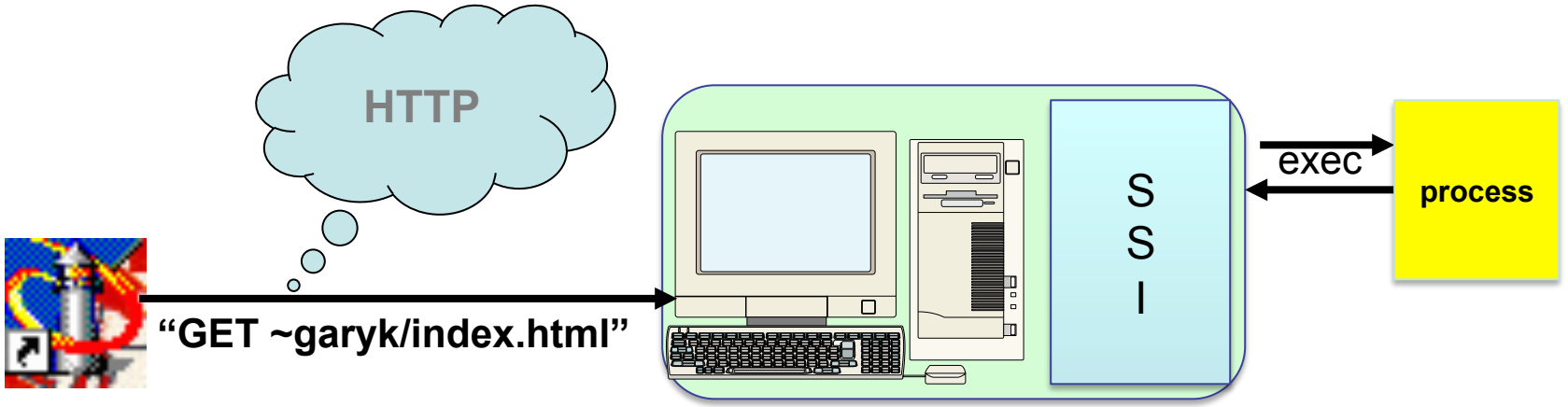
CGI

- Process per request



Server-side scripting

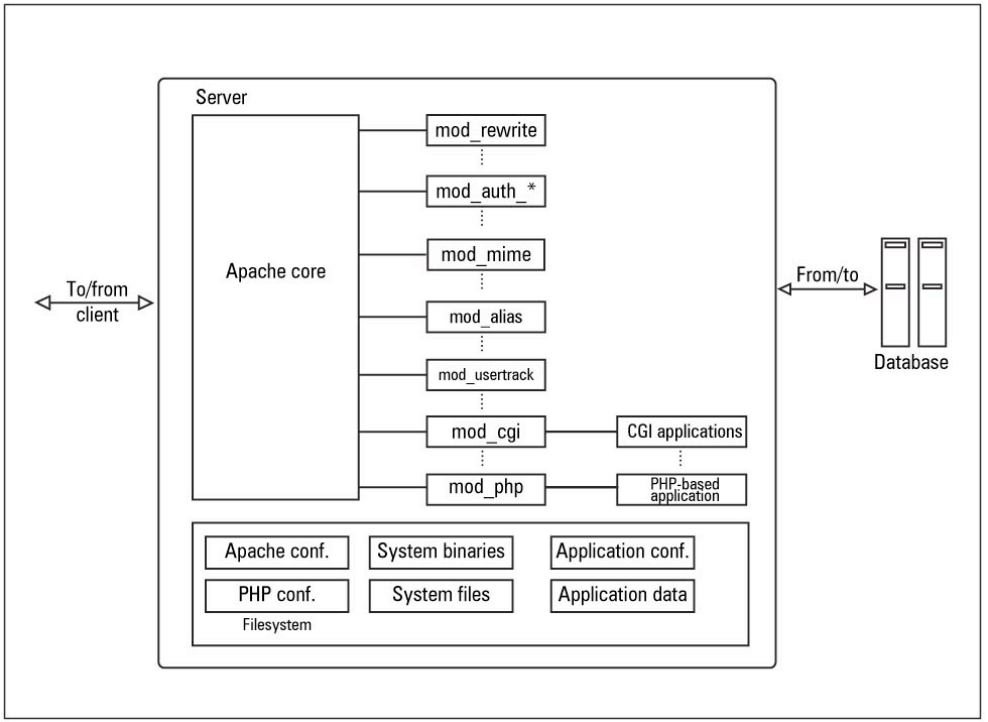
- Webserver process takes burden of processing scripts in process



Web archs (cont.)

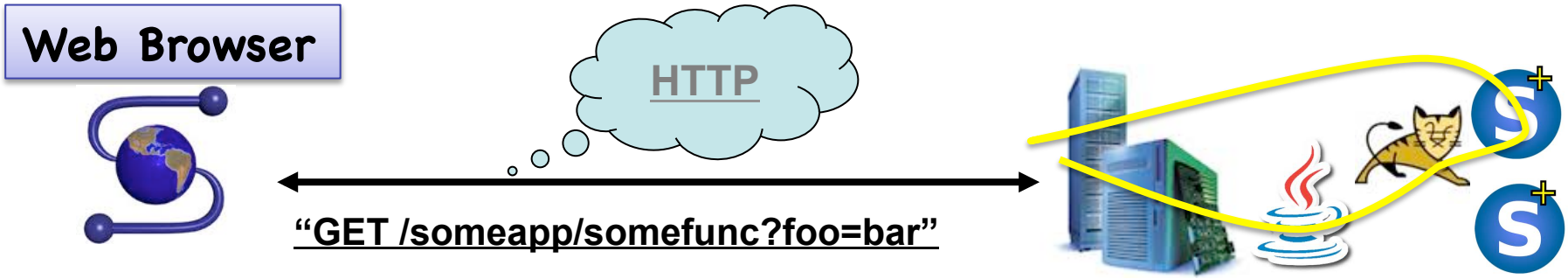
Apache mod handlers

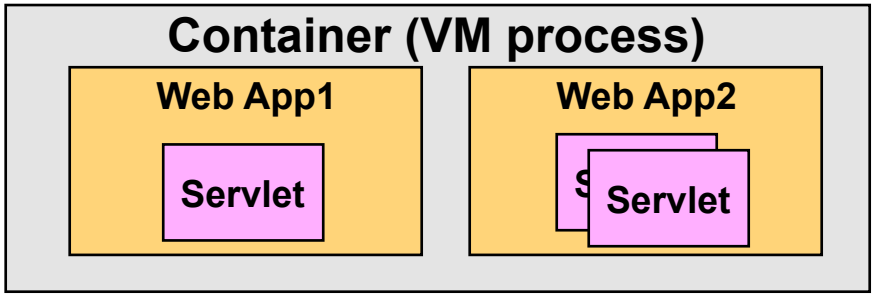
- Implicit invocation
- In-process delegation



Servlet container

- Multiple threads (workers)
- Component/container model





The case for a new model

So where are we (from a runtime perspective)?

Pros:

1. Evolved to a mature component/container architecture
2. Flexibility to distribute processing across an “enterprise architecture”
3. Can still stay simple with SSS approach

Cons:

1. That whole component/container thing is pretty complicated
2. It isn’t as flexible as we thought – brittle deployments
3. The web is moving toward a different model (front-end centric)
4. Overkill for most web applications
5. Troubleshooting distributed multithreaded apps is non-trivial
6. Impedance mismatch

The case for a new model

What can we do?

- *What if we had an approach that got rid of the threads, container, components – the lot of it? Can we just go lightweight?*
- Wouldn't that bring us back to SSS?
  - In one sense, yes! Many web-dev shops ran back to SSS-type technologies such as PHP, RoR, Python (Django)
  - However, in reality there were a lot of factors driving this, such as a rejection of verbose coding in Java. (See Grails, Scala)
- These architectures still fundamentally have a thread-per-request (or process-per-request) architecture – nah!

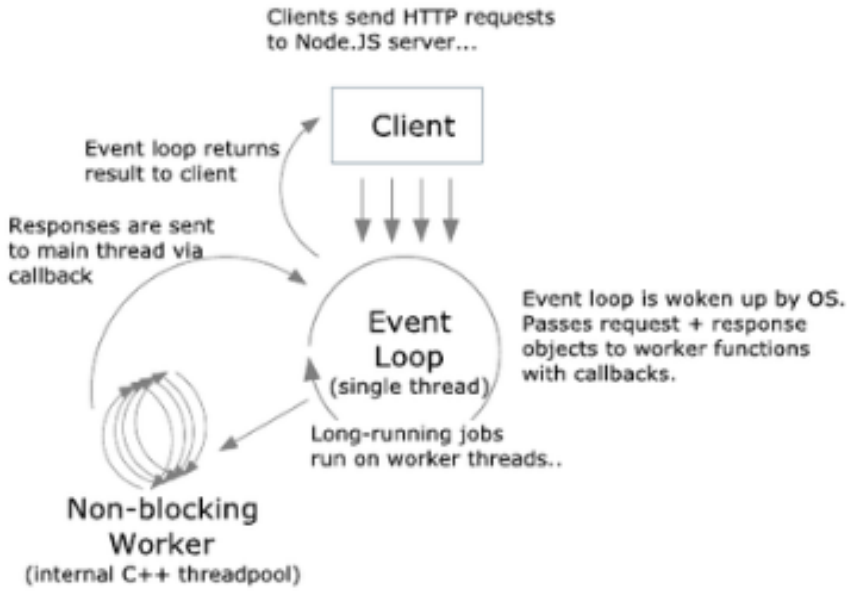
Enter NodeJS

- Uses an asynchronous process model
- Leverages Javascript, which full-stack devs prefer
- Provides a straightforward path to *containerization* and *microservices*

Single-threaded asynchronous processes

A new model

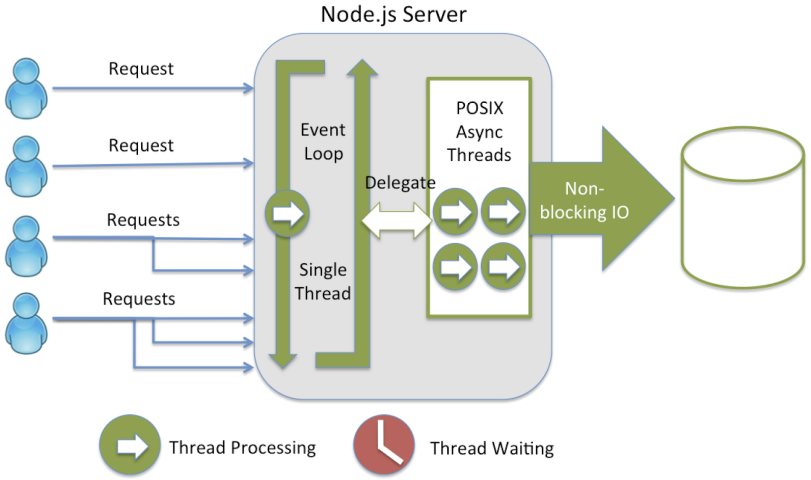
- NodeJS works by processing all incoming requests on a single thread
  - How can this scale?
  - The processing is done in a presumed non-blocking, quick computation fashion



What happens with blocking requests?

- Well these are bad
  - We try to avoid these
  - But if we can't we delegate them off to other threads w/ callbacks

Top image from blogs.msdn.com
Bottom image from strongloop.com

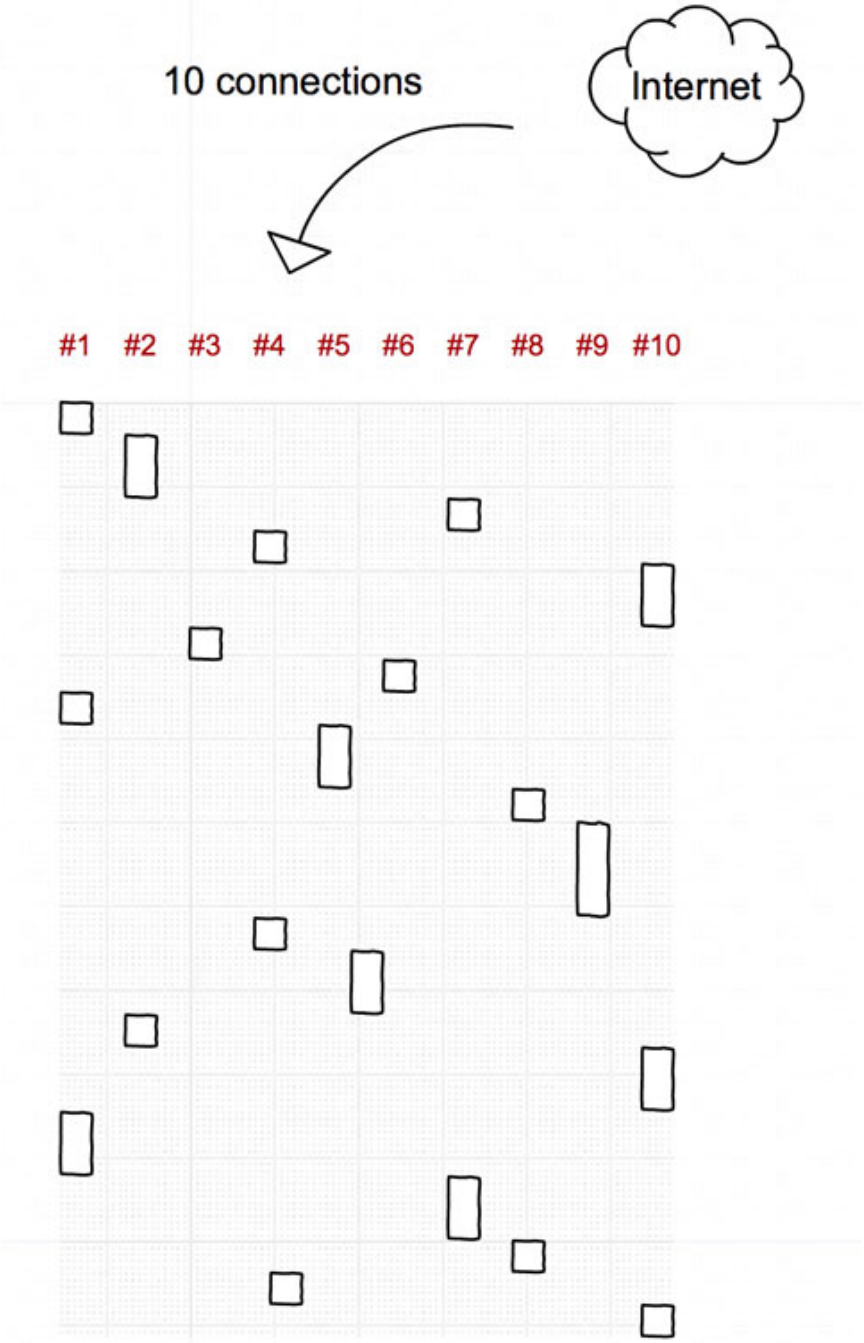


NodeJS Processing

This figure shows a timeline (going down the y-axis) of 10 HTTP requests being processed by a NodeJS server

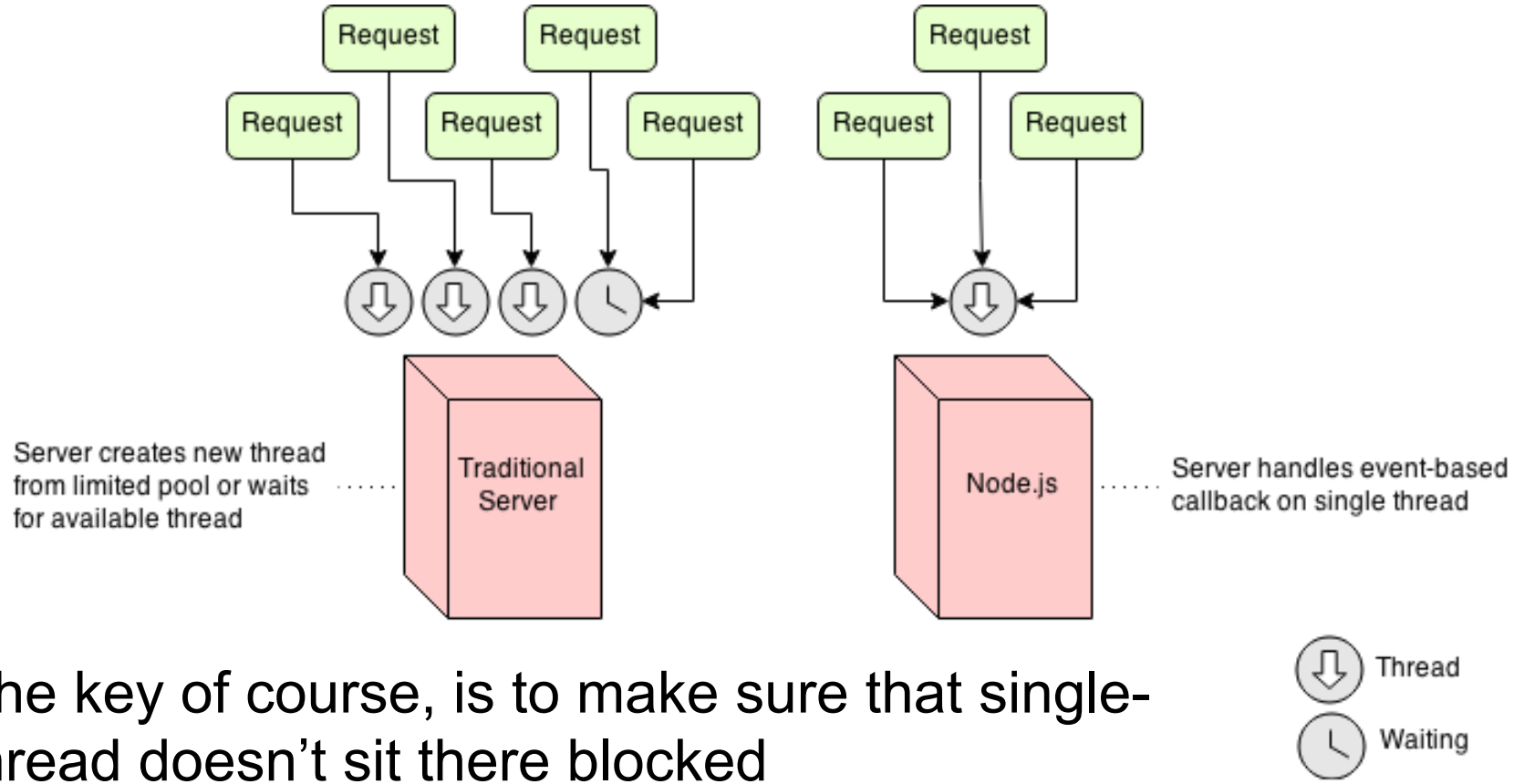
Note there is no overlap, no multiple threads – instead we have a time-slice type of round-robin processing

In order to keep the merry-go-round going, each participating request handler must make short, non-blocking calls. To return immediately, we use asynchronous calls and instead of return values we process callbacks when the work is done



bytearcher.com/articles/100-new-requests-while-still-serving-the-first-one

Comparing NodeJS to Traditional Servers



The key of course, is to make sure that single-thread doesn't sit there blocked

- Coding in an asynchronous model may not seem natural at first. It is like pseudo-multithreading
- The key is that the "hamster on the treadmill", i.e. the main event processing thread, must not stop
  - This isn't much different than Android programming!