speedshoptune

prepared by **Nate Berkopec** exclusively for

Wiki Edu

Hello Ross and WikiEdu!

Thanks for having me take a look at your application. I've identified several areas for investment in performance. Some of the things I've discussed in this report are actually just simple configuration changes you can make today that will save you a lot of money. Some things I'll be able to fix over the next three weeks as I work with you. Others are more long-term projects and skills that you can improve on over the next six months.

This document is organized at the top level by the broad, organization-level **Objectives** I've followed in preparing this document. Underneath those headers are our desired **Outcomes**, which are my goals for your performance improvements over the next six months. Underneath that are specific **Recommendations** to achieve those outcomes. Each **Recommendation** has an associated cost and benefit, rated subjectively on a 5 point scale.

At the end of the document, I will present again the **Objectives, Outcomes,** and **Recommendations** without commentary. This intended to help you turn this document into action and assist during planning your sprints, etc.

I hope you enjoy this document and find it a useful guide for the next 6 months of performance work on your application.

[illegible]

Nate Berkopec
The Speedshop

Objective 1: Improve the customer experience for users of the dashboard.

The WikiEdu dashboard is a web application. This means that users experience latency as the difference between their input - usually a click or keypress - and the browser rendering an appropriate response and the page becoming “usable”. There are several components to this latency:

1. Network roundtrip to the server.
2. Request queue time, waiting for a free Passenger process (now visible in New Relic).
3. Server response time (this is what is visible in New Relic).
4. Parsing the document.
5. Downloading the CSS and JS resources necessary for page rendering.
6. Running JavaScript necessary for page rendering.
7. Sometimes other resources need to be loaded for the page to be usable: images, videos, etc.

What determines whether or not a page is “usable” depends on the page, because it depends on what the purpose of the page is and what the user is trying to accomplish. A page whose sole purpose is to display an image is not really usable until that image is loaded, for example. On most pages of the WikiEdu dashboard, the page is usable when all of the text on the page is visible. It is important to note that “the page becoming usable” may or may not correlate with existing browser events, such as load, DOMContentLoaded, First/Largest Contentful Paint, etc.

There are two main scenarios of interaction with the WikiEdu dashboard, each having very different performance characteristics: cold loading (no cache), and warm loading (cached, moving around the site).

Outcome: Decrease time-to-usable by ~35-40% for "cold" page loads on the homepage and /training.

Together, the homepage and /training make up about 85% of initial site visits to the dashboard according to Matomo. This means most of our effort for improving the initial page load experience should be concentrated here.

Like most pages, both of these pages are usable once the text is rendered. For pages like this, the Largest Contentful Paint event is a good proxy for "when the page is usable".

Since we don't have any real-user monitoring, we will be using synthetic benchmarks to analyze the performance of these pages.

According to webpagetest.org, Largest Contentful Paint fires in 1.8 seconds for the homepage. /training takes 1.578 seconds, but the way the pages load is very similar, so any difference in these metrics is probably just statistical variation. I would say this number is actually pretty good. The requirement for Google marking this metric as "green" as part of their Core Web Vitals program is 2.5 seconds or less.

However, there are some easy opportunities for improvement that I think we should take.

RECOMMENDATION: Use an HTTP/2-capable Content Delivery Network (CDN). Cost ?/5 Benefit 3/5

Currently, the entire site is served over HTTP/1.1. Generally, the best and easiest way to transition to an HTTP/2 enabled world is to use a Content Delivery Network, or CDN. Using a CDN will give the WikiEdu a number of performance benefits:

- HTTP-cacheable content (e.g. static assets like your JS and CSS bundles) will be served from the CDN's local point of presence rather than your server, greatly decreasing network round trip time, especially for the 5% of WikiEdu users outside of North America.
- HTTP-cacheable content will also only be served by your webserver once (when the CDN grabs it for the first time), and thereafter will be served by the CDN, decreasing load on your hardware.
- HTTP/2 is more efficient at prioritizing many downloads at once, which may improve Largest Contentful Paint times.
- CDNs serve assets with the most efficient compression available - usually Brotli, decrease file sizes.
- When using a CDN, SSL connections are set up to the CDN PoP, rather than your web server, decreasing the time required to initially connect to the dashboard.
- Also, since HTTP/2 only sets up one connection per domain, time to download JS and CSS will decrease, because we no longer have to set up a connection first before we can download those assets.

Note how some of these benefits could be captured by simply setting up HTTP/2 on your Apache/Passenger servers. However, the benefits of the physical locations of CDN Points of Presence cannot be replicated.

Really, for me, the question is not IF you should use a CDN, but which, given the unique nature of your deployment environments. Wikimedia appears to have a CDN but it's not part of Cloud Services, so I don't think you can use it. Wikimedia Foundation appears to have worked with Cloudflare in the past. My preferred CDN vendors are Cloudflare and Fastly.

RECOMMENDATION: Mark as much Javascript as possible with the async attribute. Cost 2/5 Benefit 4/5

When building a webpage, when a browser's parser encounters a script tag, it must pause, download the script (if it has a src attribute), and execute it. If that script tag is inside the head tag, this means that the browser pauses and blocks before any of the actually important part of the DOM (what's inside the body tag) is ready.

There are several scripts in the head tag of the homepage and /training:

- i18n.js
- en.js
- vendors.js
- sentry.js
- jquery.js
- main.js

Every single one of these scripts must be downloaded and executed before any part of the DOM can render.

However, by adding the async attribute to a script tag, we can change this behavior. The async attribute allows the browser to unblock the parser and continue past the script tag. It's essentially saying to the browser: "download this in the background, and then execute it whenever you have finished downloading it."

I like to test if "async" behavior works for a script tag by using my browser's developer tools to block that request. If the page still looks OK and no errors are raised, it means that the script was unnecessarily blocking the rendering process. If we do this, we can see that every script (except Sentry, which is already using a sort of delayed/async mechanism internally) can be blocked, and the page still looks OK (sans rendering the upper toolbar, which I'll get to in a moment).

Given how the i18n system works, I believe this JS could only be "async'd" in the default case of a browser requesting the English version. For all other translations, the i18n scripts would have to be synchronous.

We will probably have to do some work to make each of these scripts async. Common problems include:

- Async does not guarantee order. Main.js could load before vendor.js, for example.
- Any script calls in the DOM would have to be made also async, to do deal with the fact that the functions they are trying to call may not yet be defined (for example, the way that you are initializing i18n).

With all of those scripts disabled, the homepage fires Largest Contentful paint in 850 milliseconds instead of 1350 milliseconds. That is about a 35% improvement.

RECOMMENDATION: Server-render the upper toolbar. Cost: 3/5 Benefit 1/5

The upper toolbar on the homepage is rendered client side via React. It's a very simple toolbar from the looks of things (at least it's starting state).

When marking the JS as async, this toolbar renders much later. Server-rendering it would prevent this "flash of unstyled content". This effect already occurs in the current design, but marking JS as async could make it worse.

One of the other Core Web Vitals is Cumulative Layout Shift. It measures how much the layout of the page changes after its initial render. The toolbar current renders and "pops" the entire page down, increasing the score for this metric. It makes the page less usable.

Outcome: Decrease time-to-usable for "cold" page loads on course pages by 35% or more.

Many of the same problems and solutions exist on the /courses pages as they do for the homepage, with the important difference being that these pages are client side rendered by React.

My recommendations for i18n hold for these pages as well. The Vega javascript can also be made asynchronous, as most pages do not actually require these scripts.

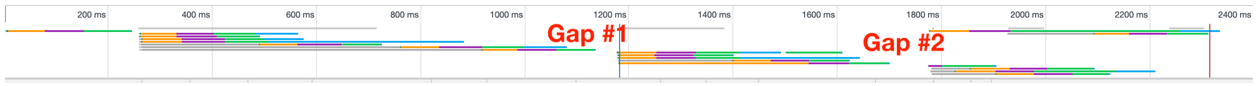
RECOMMENDATION: Optimize JS bundle sizing and composition to reduce JS downloads. Cost 3/5 Benefit 2/5

Merging the vendors Javascript and Jquery JS into the main Webpack bundle would allow us to more easily control total bundle size.

I will also audit each component of the vendors JS bundle to ensure that it is still needed and cannot be easily replaced by a smaller dependency.

RECOMMENDATION: The "main" Webpack JS bundle should come first in the head tag. Cost 2/5 Benefit 3/5

One can clearly see a large gap in the download behavior of a course page:



Ideally, all resources required to render a page should download immediately, as soon as the document has been returned. However, we see two distinct gaps appearing after the initial document delivery. Let's talk about the first one.

The resources downloaded after Gap #1 are the handful of JS files downloaded by Webpack. The download of these files is kicked off by main.js.

However, before main.js can execute, the following events must happen:

- i18n.js must download and execute.
- en.js must download and execute.
- vendors.js must download and execute.
- JQuery.js must download and execute.
- vega JS must download and execute.

These events can take a long time - up to a whole second on my local machine. If main.js executed earlier, it could start off the download for these additional Webpacked resources, required for rendering the Course page, far sooner.

RECOMMENDATION: Trigger JSON downloads sooner. Cost 2/5 Benefit 2/5

This is the second gap. After downloading and executing the Webpacked resources, a few JSON resources are requested by the now-booted React app. These resources are strictly necessary for the rendering of the page.

I believe the right move here is to use some resource hints on the course pages to trigger the download of these resources as soon as possible.

Probably the best alternative would be to use link tags with the rel=preload attribute. I'll have to be careful implementing this, as we want to be sure we don't over-prioritize these resources and accidentally slow the page down.

As an example, the course homepage requests users.json, timeline.js, course.json, and campaigns.json. You already know that the browser will request these resources at the time of rendering the route, so adding a preload resource hint in Rails will be easy.

Outcome: Interactions post-first-load should feel fast and snappy in the “courses” React app.

The React parts of the frontend experience - that is, the Javascript which runs when navigating from one tab to another, for example - are extremely simple. Latency in interactions post-first-page load is almost entirely down the backend response.

RECOMMENDATION: Remove the external service call from destroying assignments. Cost: 1/5 Benefit 1/5

The assignments destruction action makes external service calls to Wikipedia to update state there. Instead of waiting on Wikipedia to respond, we should move this task into a background job and return a 200 response to the user ASAP.

RECOMMENDATION: Reduce the average response time of 4 “unusable” endpoints. Cost: 3/5 Benefit 3/5

There are 4 endpoints whose response times are so slow on average that they are basically unusable:

- Surveys#results
- Ores_plot#campaign_plot
- Courses#manual_update
- Survey_assignments#index

Reducing response times here will make these actions actually a pleasure to use, rather than something you have to wait 10 seconds or more for!

Each one of these actions suffers from catastrophic N+1s, sometimes on the order of 10s of thousands of database calls.

RECOMMENDATION: Prefetch the other JSON resources inside a course. Cost 1/5 Benefit 2/5

We can use the prefetch resource hint to indicate an optional download which may be required on the next navigation. Using this hint on course pages will allow us to request data for the other tabs of the course, so when the user clicks them, zero latency occurs and the content can be rendered immediately.

RECOMMENDATION: Reduce the average response time of a handful of "bad" endpoints. Cost 3/5 Benefit 3/5

These are some endpoints with poor (>500 millisecond) response times that are called frequently:

- CampaignsController#articles
- CampaignsController#users
- AssignmentsController#create
- RevisionFeedbackController#index
- FeedbackFormResponsesController#create

None of these controllers have N+1 issues, so installing rack-mini-profiler on production will be required to see what's going on.

RECOMMENDATION: Fix weird behavior with ActionDispatch calling itself 30 times. Cost 1/5 Benefit 1/5

Just something I noticed on New Relic: almost every response shows ActionDispatch::MiddlewareStack::InstrumentationProxy#call being called 30 times. This is not typical. It doesn't take up a lot of time but I think it's something that should be investigated and fixed.

RECOMMENDATION: Create “alarm bells” for N+1s in development mode. Cost 2/5 Benefit 4/5

Most of the Dashboard’s most serious performance problems revolves around N+1s. A development process that reproduces these problems on your local machine will make them more visible and less likely to ever be deployed for production.

There are couple of things we can do to fix this:

1. Install query-counters in development mode that surface the number of queries that ran on a given page or endpoint.
2. Use production data in development, so that the local database is as complex as the production one.

I will work with you to come up with a process that works for you and future WikiEdu developers.

OBJECTIVE: Understand and increase capacity “headroom” for additional traffic.

Web application backends are queueing systems. When clients connect to Apache/Passenger, they are placed in a single queue. Passenger “worker” processes pull from this queue whenever they are free and not processing any other requests.

There are several simple equations, developed by Agner Krarup Erlang and the operations researcher John Little, which help us to understand the utilization and behavior of systems such as these.

In a web application backend, we wish to minimize queueing while maximizing utilization. That is, we want to keep the average time per request spent waiting in queues low, while keeping hardware usage as low as possible. An app managed in this way meets its performance requirements with the minimum amount of cost and complexity.

OUTCOME: Understand current traffic, utilization and available headroom

For a queueing system, Erlang (the guy, not the language) showed that we can say that the average number of requests being processed in parallel by the system is equal to:

*Average number of requests processing in parallel over last 60 minutes = Average response time * Average request arrival rate*

For example, during periods of high usage, the Dashboard receives about 250 requests per minute. Each request takes about 50 milliseconds on average.

This means that the average number of requests being processed in parallel is 0.050 (seconds per request) * 4.16 (requests per second). Note how the units cancel out, leaving us with a dimensionless quantity of 0.208. That means that, during periods of peak web load, your server is processing 0.208 requests in parallel. We call this the "carried traffic".

We can do the same calculation for the non-web (background job) side. In this case, the average time per job was 2 seconds, and about 1 job was performed every second. That's a carried traffic value of 2.

So, this means that you're actually processing 10 times as much background job load as you are frontend/web load.

Since you're running the background jobs, web, and even databases on a single node, that means that we can expect your long-term CPU load to about 2.2 as well. This server is a 16GB/6vCPU server. Since there are 6 vCPUs available, and we are using 2.2, our long term utilization will hover around 20%.

In most systems, utilization of 50-80% provides the best tradeoff between high utilization and low queueing times. That means we could easily afford to double usage on this instance without any decrease in service quality.

We can see already that queue times are very low. After installing the X-Request-Start header on our requests, New Relic shows that request queue times are very low (less than a few milliseconds). Once this number increases (say, to 20 milliseconds or more), we would know that we are running out of capacity for web requests.

RECOMMENDATION: Create alarms around web request queue length and certain job queue latencies. Cost 2/5 Benefit 3/5

In this model of scaling a system, the output variable is the average time spent in the queue for a web request or background job, and the independent variable is utilization. Because utilization is simply carried traffic divided by the "offered" traffic available, we decrease utilization by increasing offered traffic.

In the case of a web request, offered traffic is simply the total number of Passenger worker processes available to process a request. In the case of background jobs, it is the number of Sidekiq processes available to process a request.

Typically, we simply configure the number of processes per "box" to a number appropriate for the underlying CPU and memory of the box, and then add boxes to the setup when utilization gets high and queue times increase. You have a single node setup, a proposed multi-node setup will be described later.

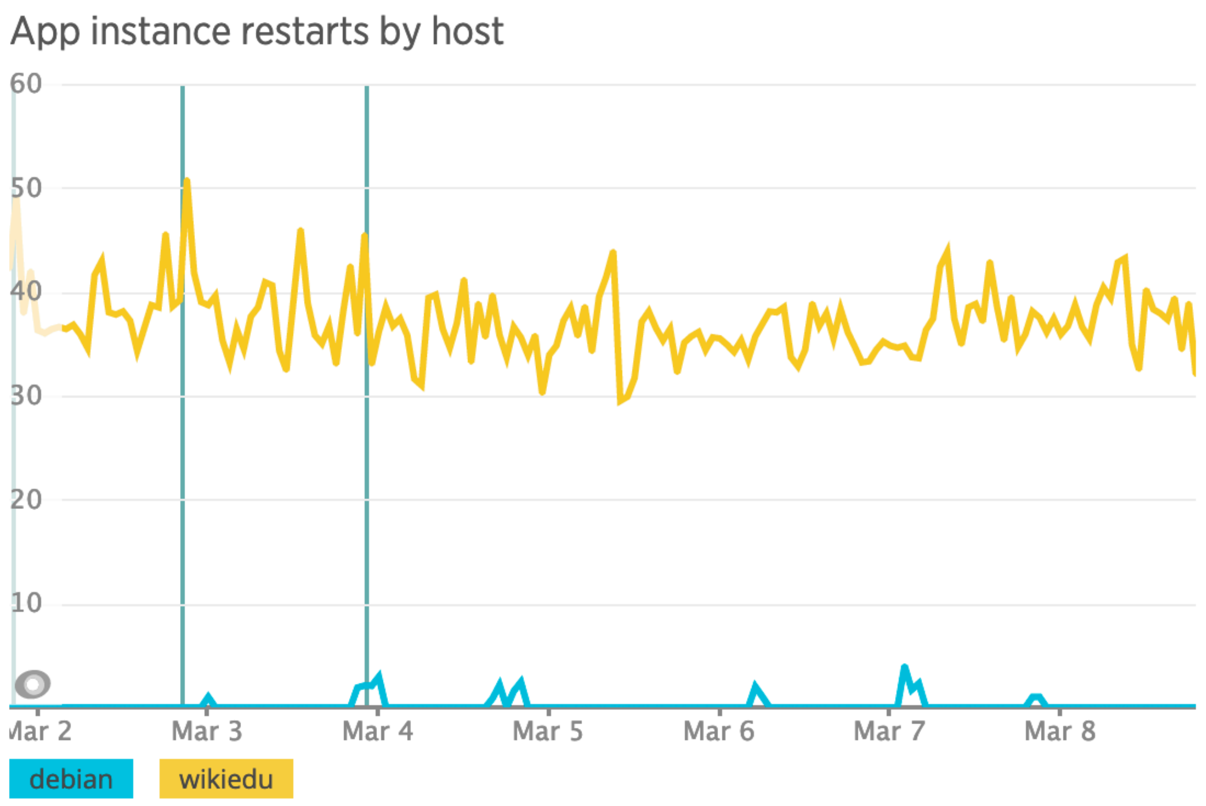
Still, we want to know when queueing is approaching bad levels.

I have already created an alert in New Relic when request queue time exceeds 20 milliseconds. We'll need to work on reporting similar metrics to NewRelic regarding how long background jobs spend in queues.

RECOMMENDATION: Create better logging around CPU, memory utilization. Cost 2/5 Benefit 3/5

We need a strategy for monitoring and reporting CPU and memory utilization. This may simply involve installing New Relic's infrastructure monitoring or something else. I'll have to work with you on what works for your situation. We want to make sure these resources do not run out under periods of maximum load.

RECOMMENDATION: Fix whatever is restarting web instances. Cost: 1/5 Benefit: 3/5



Something is restarting your web processes at the rate of about 40 per hour. We need to figure out what this is.

Restarts at this frequent of a rate increase response times, sometimes to a very large extent.

OUTCOME: Decrease capacity use.

The easiest way to scale a single-node setup like your own is to reduce the average response time of jobs and web responses, decreasing carried traffic.

As already laid out, background jobs make up 90% of your load (carried traffic of 2). For example, if we could reduce background job response time by 50%, carried traffic would be reduced by 50% as well, greatly decreasing utilization.

RECOMMENDATION: Fix N+1s across the application, particularly in background jobs. Cost 3/5 Benefit 3/5

N+1s in background jobs are increasing carried traffic far beyond what they really need to be. Reducing background job processing time by reducing N+1s would decrease capacity use to the greatest extent, far more than focusing on web requests.

In terms of reducing capacity use, there is only one job that matters: CourseDataUpdateWorker, which makes up 98% of load. Reducing the response time of this job will be the primary task here.

RECOMMENDATION: Set Sidekiq concurrency to appropriate values. Cost 1/5 Benefit 2/5

Since Sidekiq uses multiple threads, we can say that the offered traffic of a single Sidekiq process is actually greater than 1. Because of the Global VM Lock, the exact number is dependent on the amount of I/O wait that the job performs, and what our concurrency is set to:

I/O Wait	\`concurrency\`	Offered Traffic
5% or less	1	1
25%	5	1.25
50%	10	2
75%	16	3
90%	32	8
95%	64	16

This is a mathematical relationship created by Amdahl's Law.

The primary job that you're processing, CourseDataUpdateWorker, spends about 50% of its time in I/O. However, concurrency is only set to 2. Increasing concurrency to 10 would increase the offered traffic per process to 2, effectively doubling capacity.

OUTCOME: Understand and prevent future downtime on the global dashboard.

As you described to me, the global dashboard had some problems during periods of high traffic.

RECOMMENDATION: Create the logging infrastructure necessary to understand CPU and memory load on Global. Cost: 3/5 Benefit: 3/5

We simply need the same instrumentation on Global as we have on Wiki Ed. We'll have to work together on creating something that can give us the data that we need. Here's what we'll need from Global:

- Average response time (jobs and web)
- Average request rate (jobs and web)
- Queue times (jobs and web)
- CPU load and utilization
- Memory utilization

If I had to guess, you are simply outstripping the capacity of the node. The capacity utilization reduction efforts I'll be making on the app will help, but it would be helpful to know just how much headroom there is.

OUTCOME: Create a plan for future traffic growth, understanding what bottlenecks exist and approximately when they will need to be addressed.

Once we've got the current situation under control, we can start thinking about the future.

RECOMMENDATION: Create a plan for multi-node scaling. Cost: 3/5 Benefit 4/5

The most obvious thing is to create a plan for going multi-node. You'll need something that will work on both Wikimedia Cloud Services and on Linode.

The general architecture would be something like:

- N nodes for Apache/Passenger
- N nodes for Sidekiq
- 1 node for the MariaDB database
- 1 node for Redis
- 1 node for the load balancer

Probably the most important and finicky part of the setup will be the load balancing. Splitting off Sidekiq and Redis may be the easiest and the largest benefit, and the part that should be tackled first.

This recommendation and task will involve understanding private networking in both of your deployment environments, then creating scripts or other deployment architecture to actually provision these environments.

RECOMMENDATION: Optimize Sidekiq queue structure so that job latencies are addressed and documented. Cost 1/5 Benefit 1/5

You really only have two Sidekiq jobs: CourseDataUpdateWorker, and everything else.

However, as you specified to me, different courses actually have different priorities for how often they need to be synced. For each major job type, we need to document the maximum acceptable latency from the time it is enqueued until when it has finished executing, and then make sure our queue structure and worker configuration helps us to achieve those targets.

OUTCOME: Understand and improve the throughput of the course update process (CourseDataUpdateWorker), improving data synchronization latency (up to 10 minutes/update for editathons)

CourseDataUpdateWorker, and the frequency at which it is run, is intimately linked with user experience. The more often we can run it, the lower the data sync latency with Wikipedia, and the better the customer experience.

The primary impediment to this is the load that the CourseDataUpdateWorker imposes on the server, currently accounting for almost 90% of total load. By decreasing average response time, we decrease this load.

RECOMMENDATION: Fix "long" Article SELECT statements in this worker. Cost: 2/5 Benefit 2/5

This worker suffers from a few Article SELECT statements which take a long time, particularly a count triggered in the duplicate article deleter (articles_grouped_by_title_and_namespace):

```
SELECT COUNT(*) AS count_all, `articles`.`title` AS
articles_title, `articles`.`namespace` AS articles_namespace
FROM `articles` WHERE `articles`.`title` IN
(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?) AND
`articles`.`wiki_id` = ? GROUP BY `articles`.`title`,
`articles`.`namespace`
```

We need either an alternative strategy or a better query here. 2.5 million + rows are getting scanned in most cases.

RECOMMENDATION: Fix N+1s in this worker on TrainingModulesUsers, Revisions. Cost: 2/5 Benefit 3/5

I'll need to dig into this worker to understand what it does and what it loops over. Most traces of the worker show many N+1s, particularly for Revisions. Removing these will decrease job exec time.

RECOMMENDATION: Understand how this worker scales in relation to sync latency, and create a plan relating number of courses, sync latencies, and number of workers. Cost: 1/5 Benefit 3/5

So, once the average processing times are fixed, we can start thinking about how much load each course in the system creates. Then, we can make capacity planning decisions: X number of courses running at Y updates per hour means Z Sidekiq workers. That will help you to understand how quickly you need to proceed with multi-node plans, and what sync rates you can choose.

WikiEdu Dashboard: Tune Summary

I. OBJECTIVE: Improve the customer experience for users of the dashboard.
   A. OUTCOME: Decrease time-to-usable by 35-40% for “cold” page loads on the homepage
      1. RECOMMENDATION: Use an HTTP/2-capable Content Delivery Network (CDN). Cost ?/5 Benefit 3/5
      2. RECOMMENDATION: Server-render the upper toolbar. Cost: 3/5 Benefit 1/5
      3. RECOMMENDATION: Mark as much Javascript as possible with the async attribute. Cost 2/5 Benefit 4/5
   B. OUTCOME: Decrease time-to-usable for “cold” page loads on course pages by 35% or more.
      1. RECOMMENDATION: Optimize JS bundle sizing and composition to reduce JS downloads. Cost 3/5 Benefit 2/5
      2. RECOMMENDATION: The “main” Webpack JS bundle should come first in the head tag. Cost 2/5 Benefit 3/5
      3. RECOMMENDATION: Trigger JSON downloads sooner. Cost 2/5 Benefit 2/5
   C. OUTCOME: Interactions post-first-load should feel fast and snappy in the “courses” React app.
      1. RECOMMENDATION: Remove the external service call from destroying assignments. Cost: 1/5 Benefit 1/5
      2. RECOMMENDATION: Reduce the average response time of 4 “unusable” endpoints. Cost: 3/5 Benefit 3/5
      3. RECOMMENDATION: Prefetch the other JSON resources inside a course. Cost 1/5 Benefit 2/5
      4. RECOMMENDATION: Reduce the average response time of a handful of “bad” endpoints. Cost 3/5 Benefit 3/5

5. RECOMMENDATION: Fix weird behavior with ActionDispatch calling itself 30 times. Cost 1/5 Benefit 1/5
6. RECOMMENDATION: Create “alarm bells” for N+1s in development mode. Cost 2/5 Benefit 4/5

II. OBJECTIVE: Understand and increase capacity “headroom” for additional traffic.

A. OUTCOME: Understand current traffic, utilization and available headroom.
1. RECOMMENDATION: Create alarms around web request queue length and certain job queue latencies. Cost 2/5 Benefit 3/5
2. RECOMMENDATION: Create better logging around CPU, memory utilization. Cost 2/5 Benefit 3/5
3. RECOMMENDATION: Fix whatever is restarting web instances. Cost: 1/5 Benefit: 3/5

B. OUTCOME: Decrease capacity use.
1. RECOMMENDATION: Fix N+1s across the application, particularly in background jobs. Cost 3/5 Benefit 3/5
2. RECOMMENDATION: Set Sidekiq concurrency to appropriate values. Cost 1/5 Benefit 2/5

C. OUTCOME: Understand and prevent future downtime on the global dashboard.
1. RECOMMENDATION: Create the logging infrastructure necessary to understand CPU and memory load on Global. Cost: 3/5 Benefit: 3/5

D. OUTCOME: Create a plan for future traffic growth, understanding what bottlenecks exist and approximately when they will need to be addressed.
1. RECOMMENDATION: Create a plan for multi-node scaling. Cost: 3/5 Benefit 4/5
2. RECOMMENDATION: Optimize Sidekiq queue structure so that job latencies are addressed and documented. Cost 1/5 Benefit 1/5

E. OUTCOME: Understand and improve the throughput of the course update process (CourseDataUpdateWorker), improving data synchronization latency (up to 10 minutes/update for editathons)

1. RECOMMENDATION: Fix “long” Article SELECT statements in this worker. Cost: 2/5 Benefit 2/5
2. RECOMMENDATION: Fix N+1s in this worker on TrainingModulesUsers, Revisions. Cost: 2/5 Benefit 3/5
3. RECOMMENDATION: Understand how this worker scales in relation to sync latency, and create a plan relating number of courses, sync latencies, and number of workers. Cost: 1/5 Benefit 3/5