Profiling and Optimizing Performance in the Cloud

MASTER THESIS

Oscar Rosner

Submitted on 12 October 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisors:
Dr. Andreas Kaufmann
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 12 October 2021

License

Erlangen, 12 October 2021

Abstract

Internet users love fast websites and hate slow ones. They use, revisit and pay for sites with good performance and abandon bad performing sites forever.

High responsiveness and optimized resource utilization are therefore essential prerequisites for a website's popularity and financial success.

In this thesis, we profiled and optimized performance in a cloud-based web application. We improved the performance of 15 backend endpoints and doubled the frontend's overall performance score. Optimized usage of a database abstraction framework, an aggressive caching strategy, increased batch calls, and bundle size reductions had the most significant impact on performance.

We present detailed examples of performance bottlenecks in a complex web application and show how we remedied them.

Contents

Acronyms

CAQDAS	Computer-Assisted Qualitative Data Analysis Software
CDN	Content Delivery Network
CLS	Cumulative Layout Shift
CRUD	Create, Read, Update, and Delete
FAU	Friedrich-Alexander-Universität Erlangen-Nürnberg
FCP	First Contentful Paint
GAE	Google App Engine
GCP	Google Cloud Platform
JDO	Java Data Objects
JSON	JavaScript Object Notation
JS	JavaScript
LCP	Largest Contentful Paint
LOC	Lines of Code
NYT	Nailing your Thesis
PWA	Progressive Web Application
QDA	Qualitative Data Analysis
RPC	Remote Procedure Call
RTCS	Real-time Collaboration Service
SPA	Single-Page Application
TBT	Total Blocking Time
TTI	Time To Interactive

1 Introduction

Scientific research has shown that users love fast websites (Kuan et al., 2005; Liu & Arnett, 2000; Novak et al., 2000; Palmer, 2002). Numerous practitioner reports give examples. Pinterest reduced the perceived loading times of their website by 40%, which increased search engine traffic and sign-ups by 15%[1]. Google Maps decreased their homepage's size from 100 KB to ~75 KB and measured a traffic increase by 25% in the following three weeks[2]. Vodafone (IT) reduced the load time of their homepage's most prominent visible element by 31% and measured 8% more total sales[3]. Swappie reduced the average load time of their website by 23% and saw a revenue increase of 42% in the following three months[4].

Users hate slow websites (Abels et al., 1997; Egger et al., 2012; Zhang & Yang, 2009). A survey by Kissmetrics revealed that 79% of e-commerce customers who had trouble with site performance were less likely to buy from the same site again[5]. Additionally, Google's SOASTA report from 2017 showed that the probability of a *bounce* (user leaves the site) increases by 32% when page load time goes from one second to three seconds[6]. These examples reveal that even minor performance optimizations hold great opportunities and show the risks that arise when performance is neglected.

Cost models of cloud providers have increased in number and are highly complex (Buyya et al., 2011; Dutta & Dutta, 2019; Kavis, 2014; Mell & Grance, 2011). Choosing the right one can be difficult but is needed to avoid unnecessary operational costs. Therefore, a company needs to analyze and understand its application's resource usage to select the appropriate cost model.

In this thesis, we profile performance in QDAcity, a cloud-based web application for qualitative research, and use our findings to increase overall responsiveness and optimize resource utilization.

[1]https://bit.ly/39ufiY2
[2]https://zd.net/3u4wxsz
[3]https://web.dev/vodafone
[4]https://web.dev/swappie
[5]https://bit.ly/3u3O1Fk
[6]https://bit.ly/3kvoMZd

2 QDAcity

2.1 Qualitative Data Analysis

Qualitative Data Analysis (QDA) is a form of inquiry in which researchers look to answer their question about a phenomenon of interest.

> A *phenomena of interest* is something we want our research to understand, predict, explain, or describe (Rappaport, 1987).

The analyzed data is qualitative, consisting of words, descriptions, expressions, or concepts; any information that can't be reduced to a numerical representation and helps to answer the research question.

Researchers can gather qualitative data from various sources like face-to-face or telephone interviews, focus or Delphi groups, observations, videos, internet sites, and more (Bazeley, 2013; Grbich, 2013; Saldaña, 2013). The most common way to collect qualitative data is through in-depth interviews with one or more people that are or have been involved with the phenomenon of interest (Kaufmann & Riehle, 2015).

QDA is commonly embedded in an iterative process, shown in Figure 2.1, in which the researchers conduct interviews until the subsequent analysis yields no new insights that could move the theory forward (e.g., when researchers hear the same response again and again). At this point, data saturation is reached, and the data collection ends.

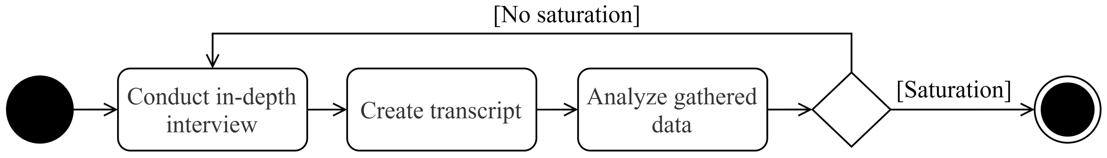


Figure 2.1: Iterative research process of collecting and analyzing data

It can take numerous interviews until data saturation is reached, and researchers can consequently end up with hundreds of pages of transcripts that require rigorous analysis. Many software products support researchers with their inquiry by assisting with data management, the coding process (explained in Section 2.2), or project coordination (Wickham & Woods, 2005). These programs are commonly referred to as Computer-Assisted Qualitative Data Analysis Software (CAQDAS).

2.1.1 Coding

Coding is a method to analyze and manage qualitative data and a fundamental skill for qualitative analysis (Bazeley, 2013). During coding, researchers assign codes to text chunks of varying sizes: words, phrases, sentences, or whole paragraphs (Miles & Huberman, 1994). Codes are based on the researcher's understanding of a text passage and describe a concept or theme in the collected data.

Supposing that we are interviewing a group of programmers, and a respondent says, "I feel stressed all the time, and I keep missing sleep". We might code this as *Programmer Burnout* and annotate the passage accordingly (in QDAcity, such an annotation is called a *Coding*). By applying *Programmer Burnout* in our analysis, we group burnout-related text chunks, which helps us to

- see how symptoms and causes of burnout vary between different respondents (data analysis),
- quickly find a quote to support a point we want to make later (data management).

2.2 Functional Overview

QDAcity is a CAQDAS and a cloud-based web application, that was developed to support the dissertation of Kaufmann (2021). It's currently developed by the Professorship for Open Source Software at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)[1]. The application has two user groups: Researchers and students. While researchers use QDAcity for their qualitative research, students are required to use QDAcity for their QDA related assignments when attending courses like Nailing your Thesis (NYT)[2] at FAU.

The app offers both user groups a wide range of functionality to assist with qualitative research but, at its core, enables users to:

1. Create or join a project with other collaborators,
2. upload RTF and PDF documents (e.g., interview transcripts),
3. annotate the documents with codings that link text segments to codes which are created and refined during the analysis to build the code system, a hierarchical structure of categories and concepts (codes), and
4. from the code system and coded data synthesize a theory that answers their research question.

2.2.1 Frontend Pages

QDAcity is accessible at https://qdacity.com. First-time visitors can create a new QDAcity account with their email and password or sign in through a third-party service such as Google or Facebook. Most functionality in QDAcity is accessible through three main pages: Personal Dashboard, Project Dashboard, and the Coding Editor. We introduce them here explicitly because they will reappear in later chapters as the target of performance optimizations.

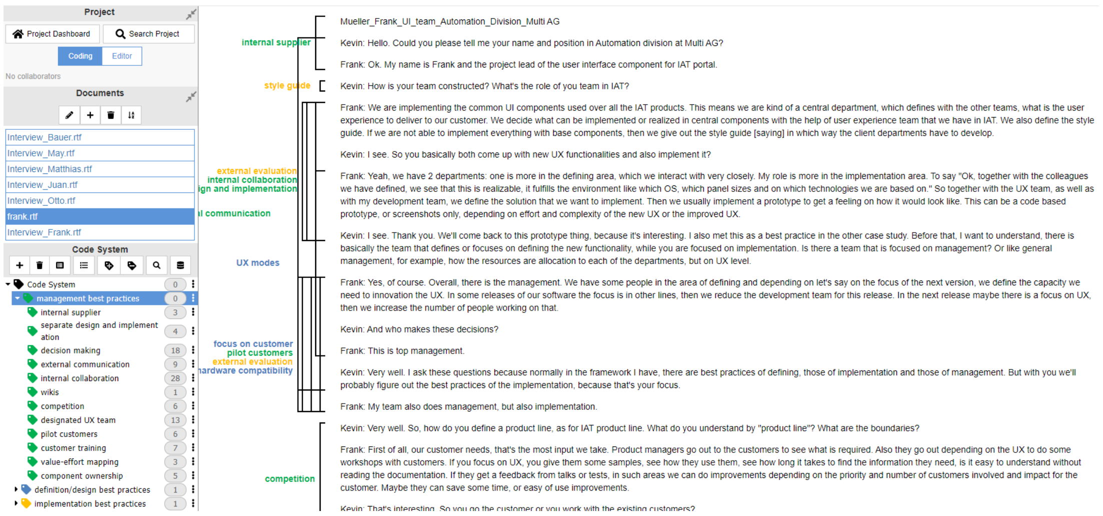
Personal Dashboard: After a successful login, this page welcomes the users and displays their projects, courses, and notifications. The dashboard's primary purpose is to manage user projects. Users can create a project themselves or join an existing one. The latter could be the case for an NYT student who received an invitation from a teaching staff member.

Project Dashboard: Clicking on a project in the list of projects (Personal Dashboard) opens the project dashboard. This page displays project properties such as name, description, settings, members, and number of documents and codes.

[1]https://oss.cs.fau.de/
[2]https://oss.cs.fau.de/teaching/specific/nyt

Coding Editor: This page is the heart of QDAcity; it's where users analyze qualitative data through coding (see Section 2.1.1). Project members maintain their codes in a shared code system. A code in the code system resembles a *code book entry* after MacQueen et al. (1998). It includes the code, a brief definition, an extended definition (code memo), guidelines for when to use the code, and guidelines for when not to use it. The code book makes the analysis process more transparent and ensures that all researchers apply the codes in the same way.

Figure 2.2 shows the coding editor. The page lists the code system's entries on the left sidebar and displays the annotated interview transcript on the right. This particular transcript was used for practice in the NYT winter term of 2020/21.



Code System	0
management best practices	0
internal supplier	3
separate design and implementation	4
decision making	18
external communication	9
internal collaboration	28
wikis	1
competition	6
designated UX team	13
pilot customers	6
customer training	7
value-effort mapping	3
component ownership	5
definition/design best practices	1
implementation best practices	1

Mueller_Frank_UI_team_Automation_Division_Multi AG

Kevin: Hello. Could you please tell me your name and position in Automation division at Multi AG?

Frank: Ok. My name is Frank and the project lead of the user interface component for IAT portal.

Kevin: How is your team constructed? What's the role of you team in IAT?

Frank: We are implementing the common UI components used over all the IAT products. This means we are kind of a central department, which defines with the other teams, what is the user experience to deliver to our customer. We decide what can be implemented or realized in central components with the help of user experience team that we have in IAT. We also define the style guide. If we are not able to implement everything with base components, then we give out the style guide [saying] in which way the client departments have to develop.

Kevin: I see. So you basically both come up with new UX functionalities and also implement it?

Frank: Yeah, we have 2 departments: one is more in the defining area, which we interact with very closely. My role is more in the implementation area. To say "Ok, together with the colleagues we have defined, we see that this is realizable, it fulfills the environment like which OS, which panel sizes and on which technologies we are based on." So together with the UX team, as well as with my development team, we define the solution that we want to implement. Then we usually implement a prototype to get a feeling on how it would look like. This can be a code based prototype, or screenshots only, depending on effort and complexity of the new UX or the improved UX.

Kevin: I see. Thank you. We'll come back to this prototype thing, because it's interesting. I also met this as a best practice in the other case study. Before that, I want to understand, there is basically the team that defines or focuses on defining the new functionality, while you are focused on implementation. Is there a team that is focused on management? Or like general management, for example, how the resources are allocation to each of the departments, but on UX level.

Frank: Yes, of course. Overall, there is the management. We have some people in the area of defining and depending on let's say on the focus of the next version, we define the capacity we need to innovation the UX. In some releases of our software the focus is in other lines, then we reduce the development team for this release. In the next release maybe there is a focus on UX, then we increase the number of people working on that.

Kevin: And who makes these decisions?

Frank: This is top management.

Kevin: Very well. I ask these questions because normally in the framework I have, there are best practices of defining, those of implementation and those of management. But with you we'll probably figure out the best practices of the implementation, because that's your focus.

Frank: My team also does management, but also implementation.

Kevin: Very well. So, how do you define a product line, as for IAT product line. What do you understand by "product line"? What are the boundaries?

Frank: First of all, our customer needs, that's the most input we take. Product managers go out to the customers to see what is required. Also they go out depending on the UX to do some workshops with customers. If you focus on UX, you give them some samples, see how they use them, see how long it takes to find the information they need, is it easy to understand without reading the documentation. If they get a feedback from talks or tests, in such areas we can do improvements depending on the priority and number of customers involved and impact for the customer. Maybe they can save some time, or easy of use improvements.

Kevin: That's interesting. So you go the customer or you work with the existing customers?

Figure 2.2: Coding in QDAcity

2.3 Technical Overview

QDAcity is a Single-Page Application (SPA) with a JavaScript (JS) client that communicates with two backend services for data storage and real-time collaboration. Figure 2.3 illustrates how these three components are wired together. QDAcity is also a Progressive Web Application (PWA) because it installs a ServiceWorker[3] that intercepts and caches network requests making the frontend fast and independent of the network (Majchrzak et al., 2018; Steiner, 2018).

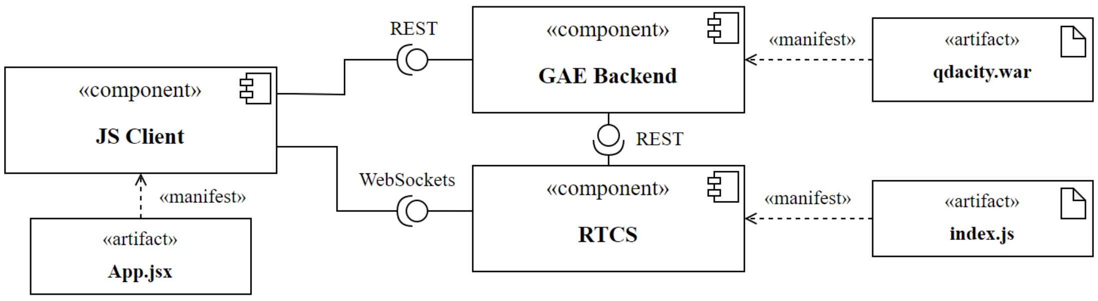


Figure 2.3: Building block view of QDAcity's components

2.3.1 JS Client

QDAcity's frontend is made with React[4], which has been the second most used web framework, behind jQuery[5], for the last two years[6][7]. React allows us to encapsulate application logic into reusable components and compose them to build a complex UI. React components are written in JSX, an extension to JavaScript, and can be implemented as a simple function or an ES6 class (ES6 is a JavaScript specification released in 2015, often called ECMAScript2015). If the component uses state, e.g., to remember its data has and its modifications, it must be implemented as a class or a function with React Hooks[8].

QDAcity has 265 such class components, written with modern JavaScript syntax. During our build process, Babel[9] transpiles these classes to browser-friendly ES5 functions, and webpack[10] bundles the result together with all necessary dependencies to create a single, minified JavaScript file as the output. Figure 2.4 shows this simplified version of the frontend build process in QDAcity.

[3]https://developers.google.com/web/fundamentals/primers/service-workers
[4]https://reactjs.org/
[5]https://jquery.com
[6]https://insights.stackoverflow.com/survey/2020
[7]https://insights.stackoverflow.com/survey/2019
[8]https://reactjs.org/docs/hooks-intro.html
[9]https://babeljs.io
[10]https://webpack.js.org

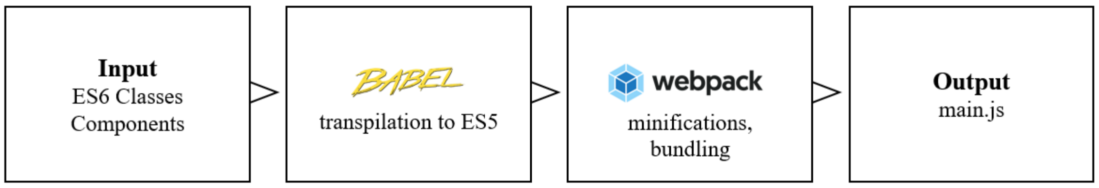


Figure 2.4: Frontend build process in QDAcity (simplified)

2.3.2 GAE Backend

A typical SPA like QDAcity has a heavy, feature-rich client that communicates with a light backend whose primary purpose is to provide an interface to the database, protected by user authentication. In QDAcity, this backend is a Java 8 application hosted on Google App Engine (GAE). GAE is an application-hosting platform running on Google's cloud infrastructure called Google Cloud Platform (GCP).

The Java application uses Cloud Endpoints[11] for generating a RESTful web API that exposes Create, Read, Update, and Delete (CRUD) operations for our data models. Cloud Endpoints routes an incoming request to the corresponding application code and returns the response as JavaScript Object Notation (JSON). The mapping between request and code is established with Java annotations on classes, methods, and parameters. The generated API consists of 226 methods from 30 endpoint classes, each class named after the data model it controls (e.g., **Project**Endpoint, **Code**Endpoint, or **User**Endpoint). To separate security concerns from business logic and avoid always-repeating boilerplate code in the endpoint methods, QDAcity installs a custom authentication interceptor that sits between an incoming request and the destined endpoint. The interceptor extracts user credentials from the request header and injects them into the next call, wrapped in a user object which is `null` if the credentials are invalid or missing.

Code 2.1 shows an endpoint class with Cloud Endpoints annotations in lines 1, 3, and 4. The code comes from the real QDAcity repository but was significantly attenuated to make the point.

```
1  @Api(authenticators = {QdacityAuthenticator.class}, /* ... */)
2  public class ProjectEndpoint {
3      @ApiMethod(name = "getProject", path = "project")
4      public Project getProject(@Named("id") Long id, User user) {
5          // Authorize user; query for project; return project.
6      }
7  }
```

Code 2.1: Cloud Endpoints annotations in endpoint class

[11] https://cloud.google.com/endpoints

For persistence, the GAE backend uses a NoSQL document database called Firestore in Datastore mode[12]. The endpoints access the datastore in two ways: through a low-level API from Google[13] or with Java Data Objects (JDO).

JDO in QDAcity

The App Engine SDK includes an implementation of JDO, a data access interface that provides three main features for QDAcity.

1. Automatic mapping between java objects and datastore entities,
2. consistent data schema and type safety for the schemaless NoSQL datastore,
3. abstraction of the datastore, making it easier to move to another cloud provider and avoid vendor lock-in.

The App Engine SDKs implementation of JDO is based on the open-source software called DataNucleus AccessPlatform[14], which is installed in QDAcity as a maven plugin.

2.3.3 RTCS

A unique selling point of QDAcity is the real-time collaboration feature that allows multiple project members to work on the same document simultaneously while having every change to the code system or the codings shared between them.

Such real-time collaboration is a technical challenge as it requires full-duplex communication between client and server and a connection that can last for hours. Both of these requirements cannot be met by the GAE backend since it communicates exclusively via HTTP and has a maximum request deadline of 60 seconds. QDAcity uses a second backend service instead, the RTCS, a NodeJS[15] application running on a virtual machine on Compute Engine[16].

When loading the coding editor, project members automatically initiate the communication with the RTCS through a WebSocket connection that remains open for the duration of the coding session. The RTCS doesn't forward the incoming messages to the other project members directly but instead publishes them to a Redis Room[17] which can be subscribed to by other RTCS instances. This indirection is necessary to allow the RTCS to scale horizontally during times of high

[12]https://cloud.google.com/datastore/docs
[13]https://cloud.google.com/appengine/docs/standard/java/datastore/api-overview
[14]https://www.datanucleus.org/products/accessplatform_6_0/index.html
[15]https://nodejs.org/en/
[16]https://cloud.google.com/compute
[17]https://github.com/socketio/socket.io-redis-adapter

traffic volume while maintaining the communication between project members connected to different instances. Figure 2.5 illustrates the dynamic view of the RTCS for the internal scenario: “Project member adds a code to the code system with two collaborators and two RTCS instances”.

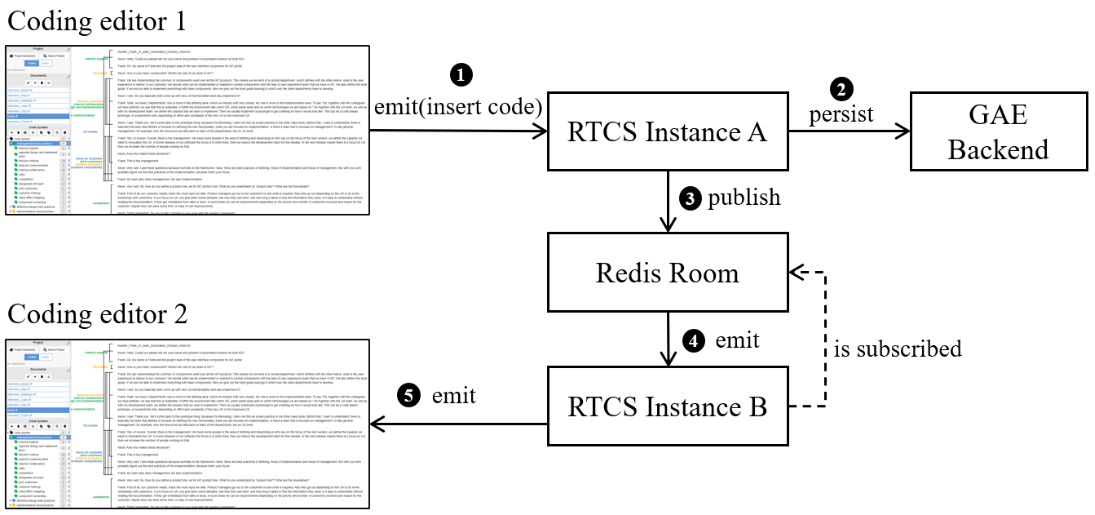


Figure 2.5: Dynamic view of the RTCS

2.3.4 Source Code Distribution in QDAcity

Although we have described the JS Client, GAE Backend, and the RTCS in the previous sections in similar length, this does not reflect their share in the repository and complexity. In Table 2.1, we aggregated the Lines of Code (LOC) that each of the three parts contributes to the total size of the repositories source code. We included this ratio to clarify where most of the working hours have gone since the start of QDAcity.

Component	**LOC**	**Rel. LOC**
JS Client	56,155 (77% JSX, 22% JS)	68%
GAE Backend	25,122 (100% Java)	30%
RTCS	1,686 (100% JS)	2%

Note 1: LOC is the sum of lines from all files in “Directory”, including comments and blank lines.

Note 2: Percentage values were rounded to the nearest integer (5.92% → 6%, 10.2% → 10%).

Table 2.1: Lines of Code in QDAcity

The JS Client contains 68% of the total LOC, which emphasizes our point from Section 2.3.2 that the complexity in QDAcity lies in the frontend.

3 Performance in the GAE Backend

We profiled the code of the GAE backend to identify existing performance bottlenecks and build up a set of re-usable code-tuning strategies for optimization. We present the results with the highest potential for performance improvements in sections 3.1.1 - 3.1.7.

Building on the knowledge we gained from the profiling, we defined quality requirements to optimize performance in the most frequently used endpoints. Section 3.2 presents an overview of the optimizations we applied and describes a few specific cases where we achieved significant performance improvements.

In Section 3.3, we compare the implementation results from 3.2 with the requirements from 3.1 and describe additional maintainability improvements which were not directly related to performance.

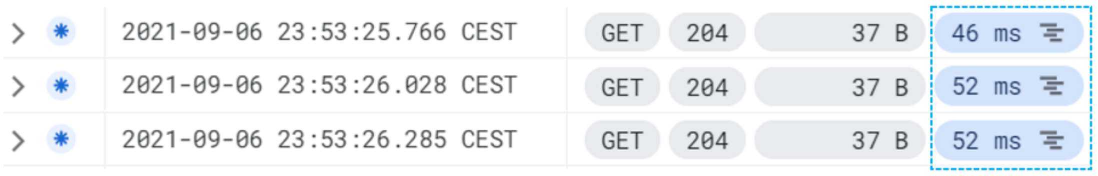
3.1 Profiling Results

Method

We profiled an App Engine F1 instance located in Iowa (us-central1) that runs Java 8 in the standard environment. We conducted the experiments in the qda-city testing environment, a separate GCP project that mirrors the production environment, including its GCP services (App Engine, Compute Engine, ...) and its datastore entities.

We looked at two indicators to measure request performance: Latency and the number and type of Remote Procedure Calls (RPCs) each endpoint makes during execution. The latency is the request processing time on the instance, from when the request was received until the response was sent. It does not include the time it took for the request to travel back and forth from our Postman client[1] to App Engine and is therefore only a fraction of the round-trip time.

App engine logs the latency of every request and includes it in the request log that contains other meta information such as the HTTP method or the status code. Request logs are displayed in the Cloud Logging[2] console from where they can be examined, filtered, and eventually downloaded.



```
2021-09-06 23:53:25.766 CEST   GET   204   37 B   46 ms
2021-09-06 23:53:26.028 CEST   GET   204   37 B   52 ms
2021-09-06 23:53:26.285 CEST   GET   204   37 B   52 ms
```

Figure 3.1: Example logs in Cloud Logging

To measure the execution time of individual lines, we wrote timestamps into the standard output, from where App Engine picks them up and adds them to the request log. We tried to use other instrumentation clients, such as Google's recommended solution, OpenCensus[3]. Still, it did not work because it uses gRPC[4] to transmit tracing data to Cloud Logging, a framework not supported in the standard environment[5] of App Engine.

We did not include instrumentation code in the code snippets of the profiling results but highlighted the lines to indicate such instrumentation. We use this convention to make the code examples easier to understand while not neglecting any methodological details (Code 3.1 shows this convention).

[1] https://www.postman.com/product/rest-client/
[2] https://cloud.google.com/logging
[3] https://opencensus.io
[4] https://grpc.io
[5] https://opencensus.io/integrations/google_cloud/google_cloud_appengine_standard

```
Long start = System.currentTimeMillis();
/* code under test */
Long end = System.currentTimeMillis();
logToStdout("Time = " + (end - start));
```

```
/* code under test */
```

Code 3.1: Convention for shortening instrumented code

App Engine automatically creates a cloud trace[6] for every nth request (the sample rate is about 5% - 1 out of 20 requests gets traced). The trace contains every remote procedure call made during endpoint execution (see Figure 3.2).

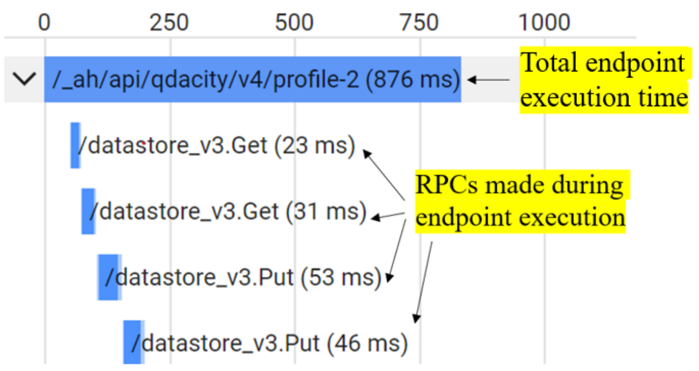

0
250
500
750
1000
/_ah/api/qdacity/v4/profile-2 (876 ms)
Total endpoint execution time
/datastore_v3.Get (23 ms)
/datastore_v3.Get (31 ms)
/datastore_v3.Put (53 ms)
/datastore_v3.Put (46 ms)
RPCs made during endpoint execution

Figure 3.2: Example cloud trace with two remote procedure calls

The endpoints of the GAE backend typically made these requests to access distributed App Engine services such as the datastore. The traces helped us to understand what was happening behind the scenes and improve the efficiency of these RPCs based on this understanding.

Confounding Parameters

Latency Fluctuation:
To smooth out sample-to-sample fluctuation of request latencies, we increased the sample size to 200 and took the median value to ignore above- and below-average latency spikes caused by unexpected network issues. So when we present the "request latency" or "response time" of an endpoint in the following sections, we always mean the median time of 200 requests.

Loading Requests:
When a newly created instance receives a request, App Engine has to load the necessary application code to process the request, leading to response times up to 20 times higher than usual (see Section 3.1.7). To keep the response time of the initial *loading request* out of our sample data, we sent an additional request

[6]https://cloud.google.com/trace

prior to each test run to trigger the initialization tasks and make the instance as ready as possible.

App Engine Auto Scaling:
If the GCP load balancer receives 200 requests in a short time, App Engine's auto-scaling mechanism will be triggered and create new instances to handle the traffic spike. We limited the number of instances to 1, preventing any auto-scaling and the resulting loading requests.

3.1.1 The Cost of a Datastore Write

This section shows that it takes longer to write entities to the datastore than to read them. We will use the results to support our points in later sections.

Experiment

Code 3.2 uses our JDO implementation DataNucleus to load a project from the datastore (the read operation) and updates its name (the write operation). We measured the latency of these two operations and highlighted the responsible lines below.

```
// Get PersistenceManager as 'pm'

Project project = pm.getObjectById(Project.class, /* id */); // Read entity.

project.setName(/* new name */); // Write entity.
```

Code 3.2: Datastore GET and PUT with DataNucleus

Table 3.1 shows that the second line, containing the write, takes more than twice as long as the first one. Note: To calculate the *Percentage Change* we used the formula in Appendix A.

Time to Read	Time to Write	Percentage Change
10 ms	23 ms	130% INCREASE

Table 3.1: Datastore write takes twice as long as a read

Financial Cost

A datastore write is also more expensive than a datastore read. A single write to a datastore located in North America costs \$0.0000018, which is three times more costly than a datastore read[7]. The cloud computing industry is highly competitive, so these prices and thresholds frequently change.

[7]https://cloud.google.com/datastore/pricing

3.1.2 Taking Advantage of Transparent Persistence

When a persistence manager loads an object from the datastore, the object becomes *attached* to that persistence manager. It remains in this state until it's actively detached with `pm.detach(object)` or until the persistence manager is closed with `pm.close()`. When the application sets data on an attached object, DataNucleus automatically saves these changes to the database without a previous explicit call to an output method like `makePersistent()`. This mechanism is called *transparent persistence*[8] and is essential for separating the application layer from the database layer.

Experiment

In this section, we measure the overhead caused by an explicit call to `makePersistent()`. Code 3.3 shows two request methods. The left one contains the unnecessary call; the second one does not. Except for this slight difference, both methods are identical. They obtain an instance of a persistence manager, load a project, and set its name to a new value.

```
/* Get PersistenceManager
   Load existing project */

project.setName(/* new value */);
pm.makePersistent(project);
```

```
/* Get PersistenceManager as 'pm'
   Load existing project */

project.setName(/* new value */);
```

Code 3.3: Comparing: Calling makePersistent with an attached object.

Our profiling results show that the call to `makePersistent()` is unnecessary and causes overhead that negatively impacts response time. By omitting the call, we reduced the median request latency from 49.46 ms ($N = 200$) to 47.47 ms ($N = 200$); this is a decrease of 4% (see Table 3.2).

Time Previous	Time Optimized	Percentage Change
49.46 ms	47.47 ms	4% DECREASE

Table 3.2: Optimized request latency without explicit makePersistent call

Taking 2 ms off a request will not impact the perceived response time. Still, the optimization is justified because it takes advantage of transparent persistence to eliminate unnecessary code.

[8] https://wiki.c2.com/?TransparentPersistence

3.1.3 Multiple Changes to Attached Object

In the previous section we described transparent persistence, a mechanism that automatically causes a datastore write whenever data is set to an attached object. This implies that multiple succeeding data manipulations cause the same number of datastore writes which are sent to the datastore one after the other. This section proves that DataNucleus behaves exactly as described and presents an optimization technique to avoid transparent persistence.

Experiment

Code 3.4 loads a project from the datastore and calls a setter method three times with new values in each case.

```
// Load project as 'project' with persistence manager.

project.setName(/* new value */); // 1st datastore write.
project.setDescription(/* new value */); // 2nd datastore write.
project.setCodeSystemId(/* new value */); // 3rd datastore write.
```

Code 3.4: Three subsequent calls to setter method on an attached object

Figure 3.3 shows the network trace that this code generates. As expected, we see three consecutive datastore writes, each of which lasts about 40 ms.

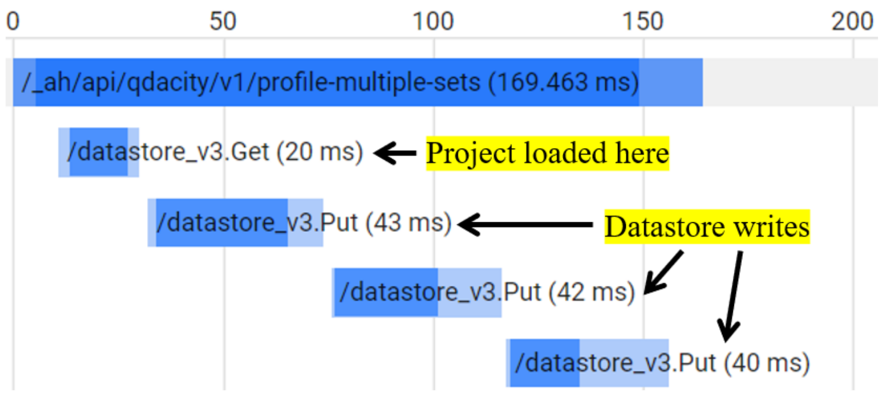


Figure 3.3: Network trace of multiple sets to attached object

These results are particularly problematic when combined with the results from Section 3.1.1, which show that datastore writes are slower and more costly than other operations.

Optimization

To stop transparent persistence and, as a result, reduce the number of sequential datastore writes, we detach the project from the control of the persistence manager with `makePersistent()`. The method returns a detached copy of the original project without any strings to an underlying persistence mechanism. We can safely use the setter methods of the detached object and follow it up with an explicit call to `makePersistent()` when we're done (see Code 3.5).

```
// Load project as 'project' with persistence manager 'pm'.

Project detachedCopy = pm.detachCopy(project);

detachedCopy.setName(/* new value */); // No datastore write.
detachedCopy.setDescription(/* new value */); // No datastore write.
detachedCopy.setCodeSystemId(/* new value */); // No datastore write.

pm.makePersistent(detachedCopy); // 1st datastore write.
```

Code 3.5: Detaching an object before setting its data

The network trace in Figure 3.4 shows the desired result; a single datastore write caused by the explicit call to `makePersistent()` instead of three.

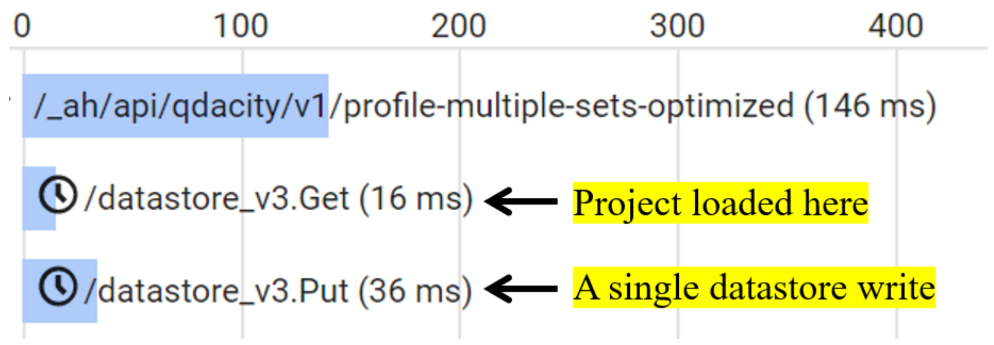

0
100
200
300
400
/_ah/api/qdacity/v1/profile-multiple-sets-optimized (146 ms)
/datastore_v3.Get (16 ms)
Project loaded here
/datastore_v3.Put (36 ms)
A single datastore write

Figure 3.4: Network trace of multiple sets to detached object

3.1.4 Batch Operations

DataNucleus allows us to persist and delete entities in batches which is a crucial optimization technique to reduce the number of remote procedure calls made during endpoint execution. This section shows their impact on efficiency.

Experiment

Code 3.6 fills a list with 500 new project objects and persists them with a call to the batch version of `makePersistent()`.

```java
// Get persistence manager as 'pm'

List<Project> projects = new ArrayList<>();
// Fill the list with 500 new projects

pm.makePersistentAll(projects); // Persist all projects at once
```

Code 3.6: Batch call for persisting multiple projects

Our profiling results show an underwhelming result. It appears that the maximum batch size is limited to ten entities per RPC to /datastore_v3.Put and so the batch call from above causes 50 (500/10) datastore writes (see Figure 3.5).

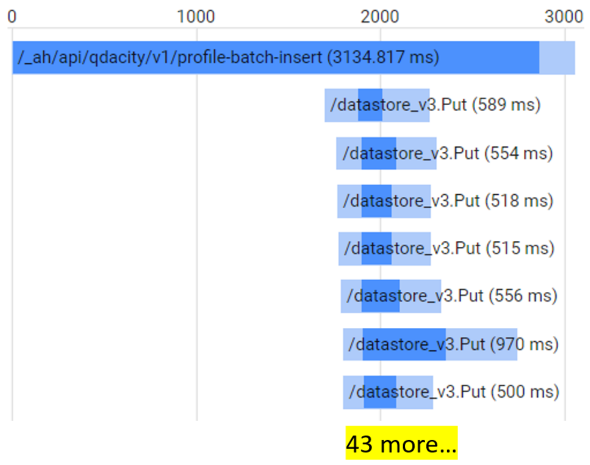


Figure 3.5: Batch version of makePersistent causes 50 RPC calls

For batch deletes, the limit is increased to 100, and deleting the entities above with `deletePersistentAll()` therefore caused five (500/100) datastore writes. The DataNucleus documentation mentions the *batchLimit* configuration parameter for changing the default batch sizes for RDMBS systems but such a parameter does not exist for the datastore. We ran the same experiment with the batch APIs of the memcache but could not detect a similar limit for batch sizes. Caching, retrieving, and deleting 500 objects from memcache in three separate calls resulted in three RPCs.

3.1.5 Query Entities in Superclass Table

With DataNucleus, the developers can decide how to persist classes of an inheritance tree by defining their *inheritance strategy*. For most classes, the GAE backend used the *new-table* strategy where each class has its own table in the datastore. Figure 3.6 illustrates the result of the new-table strategy with the example of the BaseProject class and its derivatives.

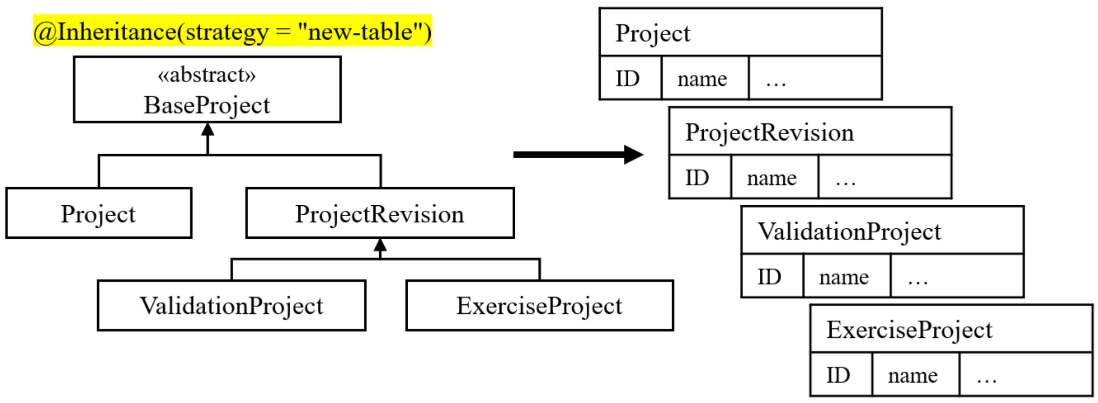
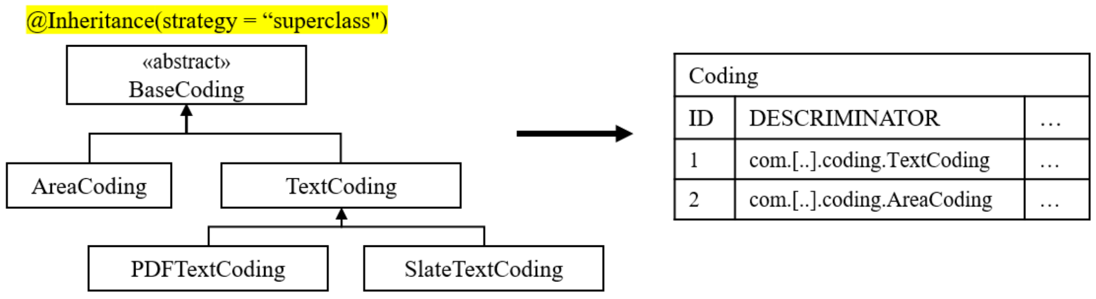


Figure 3.6: Result of new-table inheritance strategy

To persist codings and documents, the GAE backend used a different inheritance strategy called *superclass*. With superclass, DataNucleus stores the entities of all subclasses in a single table with an additional discriminator column that identifies the object type (see Figure 3.7).



ID	DESCRIMINATOR	…
1	com.[..].coding.TextCoding	…
2	com.[..].coding.AreaCoding	…

Figure 3.7: Result of superclass inheritance strategy

This section shows that while executing a query on a superclass table, DataNucleus makes a remote procedure call for every class in the inheritance tree of the table's entries.

Experiment

Code 3.7 uses a DataNucleus query to retrieve all coding entities with a particular project id from a superclass table called “Coding”.

```
// Get persistence manager as 'pm'

Query query = pm.newQuery(BaseCoding.class, "projectId == :arg0");
List<BaseCoding> codings = query.execute(/* existing project id */);
```

Code 3.7: DataNucleus query to retrieve codings

Figure 3.8 shows the code’s trace, which lists six calls to /datastore_v3.RunQuery, executed in sequential order.

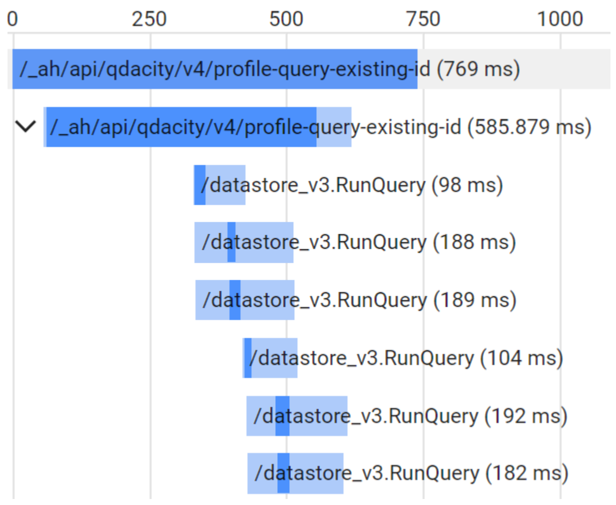


Figure 3.8: Trace of DataNucleus query on superclass table

We repeated the experiment with a different superclass table which made us realize that the number of RPCs was equal to the number of classes in the inheritance tree. This explains why the above query made six calls: *five* coding subclasses and *one* base class.

Making multiple separate calls to query entities from a single table is obviously less efficient than a single call and slower because, as the trace shows, DataNucleus doesn’t execute the calls in parallel. Additionally, the query’s RPC usage doesn’t scale well either because it will execute another RPC for each new subclass added to the tree.

Optimization: We rebuilt the above query with the low-level datastore API and executed it on the same table. Instead of six, the new API only needed a single call to retrieve the same entities from the superclass table. The substitution of DataNucleus with the low-level datastore API is therefore a viable optimization technique when searching superclass tables.

3.1.6 Datastore vs. Memcache

App Engine has a distributed memory cache service known as *memcache*, named and modeled after the original memcached[9]. Memcache stores key-value pairs in memory (RAM) to allow fast read and write access to objects. Because even google servers can crash[10], memory is volatile, and memcache is therefore not meant for long-term storage like the datastore. Instead, it's intended for short-term storage of commonly queried datastore entities.

To test how fast memcache is, we measured the latency of a *cache hit* (value is found) and compared it against a datastore read.

Code 3.8 loads the same project, first from the datastore and then from memcache. The two calls differ because the memcache service requires a key that must not be longer than 250 bytes. We created such a key for this example by concatenating the project's id with its class name.

```
/* Get persistence manager as 'pm'
   Get memcache service as 'mc' */

Project p_ds = pm.getObjectById(Project.class, /* id */);
Project p_mc = (Project) mc.get(/* id.classname */);
```

Code 3.8: Loading a project from datastore and memcache

We ran this code 200 times and saw that with memcache, the project was retrieved 55% faster than with the datastore query (see Table 3.3).

Time with Datastore	Time with Memcache	Percentage Change
9 ms	4 ms	55% DECREASE

Note: Standard deviation was three times higher for the datastore read than for the cache hit (3.3 → 1).

Table 3.3: Comparing cache hit against datastore read

The results will be used in Section 3.2 to justify an aggressive caching strategy in our optimizations.

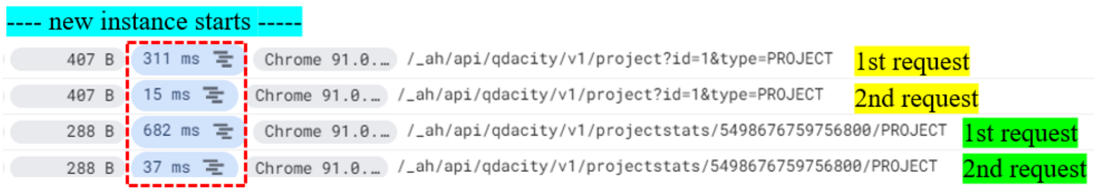
[9]https://memcached.org
[10]https://en.wikipedia.org/wiki/2020_Google_services_outages

3.1.7 The Effect of Loading Requests

The first request issued to a newly created instance typically takes significantly longer than the subsequent requests handled by the same instance. Such a *loading request* happens because the instance needs to perform app-specific initialization tasks like loading application code or to establish connections to distributed services.

Experiment

With our postman client, we sent four requests to a newly created instance, issuing the first two requests to the project endpoint and the last two to the project stats endpoint. Both endpoints belonged to two different Java classes, Project.java and ProjectStats.java. The request logs in figure 3.9 show that in each case, the first request took significantly longer than the next one, up to 20 times longer for the project endpoint.



```
---- new instance starts -----
407 B   311 ms   Chrome 91.0...   /_ah/api/qdacity/v1/project?id=1&type=PROJECT              1st request
407 B   15 ms    Chrome 91.0...   /_ah/api/qdacity/v1/project?id=1&type=PROJECT              2nd request
288 B   682 ms   Chrome 91.0...   /_ah/api/qdacity/v1/projectstats/5498676759756800/PROJECT  1st request
288 B   37 ms    Chrome 91.0...   /_ah/api/qdacity/v1/projectstats/5498676759756800/PROJECT  2nd request
```

Figure 3.9: Request logs with two loading requests

This result is concerning because it illustrates that a loading request happens not only once per instance, but each time an endpoint from a new Java class is executed. This means that the first user who uses a newly created instance experiences loading requests for most endpoints resulting in horrific response times.

3.2 Quality Requirements

Glossary:

- *loading request*: A request in which an App Engine instance performs initialization tasks that lead to significantly longer response times than subsequent requests.
- *attached object*: A Java object that is managed by a DataNucleus persistence manager.
- *superclass table*: A datastore table containing entities of different subclasses belonging to the same inheritance tree. Such a table is the result of the "superclass" inheritance strategy of DataNucleus.

Based on the knowledge we gained from the profiling results, we defined the following six quality requirements to enhance the existing GAE backend:

GAE REQ 1: If a user sends the first request to a newly created instance, that request shall not be a loading request.

The results in Section 3.1.7 show that a loading request can take up to 20 times longer than subsequent requests handled by the same instance. To prevent a real-user from experiencing these response times, the instance should perform initialization tasks before handling user traffic.

GAE REQ 2: If an object exists in memcache and in the datastore, the endpoints shall load that object from the memcache.

Memcache is a distributed cache service from which objects can be loaded 55% faster than from the datastore (see Section 3.1.6). The memcache should therefore be preferred over the datastore if an entity exists in both storages.

GAE REQ 3: The endpoints shall not call `makePersistent()` explicitly after setting data to an attached object.

If data was set on an attached object, an explicit call to `makePersistent()` has no effect and increases request latency by 4% (see Section 3.1.2).

GAE REQ 4: The endpoints shall detach an attached object from the persistence manager before setting its data more than once in a row.

Setting data repeatedly to an attached object triggers an equal number of RPCs that write the changes to the datastore. By detaching the object beforehand, we can persist all changes with a single RPC.

GAE REQ 5: If the endpoints operate on independent entities, they shall use the batch APIs of memcache and the datastore.

Although Section 3.1.4 shows that DataNucleus limits the batch size of datastore writes to 10 and to 100 for deletes, combining individual calls into batches is still beneficial for performance and efficiency. For the batch APIs of the memcache, we did not find any size limitations.

GAE REQ 6: The endpoints shall use the low-level datastore API instead of DataNucleus to query superclass tables.

We found that DataNucleus makes (1 + *number of subclasses*) RPCs when executing a query on a superclass table such as “Coding” or “Document”. After replacing DataNucleus with the low-level datastore API, the query only caused a single RPC (see Section 3.1.5).

3.2.1 Endpoints for Optimization

We narrowed our optimizations to the endpoints we thought would most frequently be invoked by QDAcity users. We deliberately use the word *would* here since the user base was too small to derive representative functional demands. For example, if we only tracked the behavior of NYT students, who use QDAcity for university assignments instead of independent qualitative research projects, we would get a distorted picture of what other users do, want, or ignore.

We applied our optimizations to so-called *hot* endpoints that

- are automatically invoked by the frontend upon rendering the personal dashboard, project dashboard, or coding editor,
- handle common operations in the coding editor, such as adding, modifying, or deleting a code or the associated codings.

These two criteria were matched by 21 hot endpoints, which amounted to ~9% of total endpoints in the GAE backend. Optimizing only 9% of endpoints may seem insufficient, but the code size says nothing about its share of the program's total execution time, as research has shown:

- In 1997, the Standish group[11] analyzed 100 applications and found that only 20% of the application features were used often, while the remaining 80% were rarely or never used (Standish Group, 2010).
- In his "Industrial software metrics top 10 list", Boehm (1987) writes that 20% of a program's modules consume 80% of the program's entire execution time, corresponding to the classic Pareto principle, which states that 80% of the outcomes are due to 20% of causes (Pareto, 1897).
- In his paper "An empirical study of Fortran projects," Knuth (1971) found out that, on average only 4% of a program's code was responsible for 50% of its runtime.

[11]https://www.standishgroup.com/

3.3 Implementation

Using the criteria from Section 3.2.1, we selected 21 endpoints for optimization, which are shown in Table 3.4.

Page	**Endpoint URL**	**Optimizations**
Personal Dashboard	GET /user	[0, 2]
Personal Dashboard	GET /projects	[0, 1]
Personal Dashboard	GET /validationproject	[0, 2]
Personal Dashboard	GET /usergroup.listUserGroups	[0, 1]
Personal Dashboard	GET /todos	[0]*
Personal Dashboard	GET /notification.listNotifications	[0, 1, 2]*
Personal Dashboard	GET /listTermCourseByParticipant	-
Personal Dashboard	GET /userTutorialState	-
Personal Dashboard	GET /courses	-
Project Dashboard	GET /projectstats/{p-id}/{p-type}	[0, 3]
Project Dashboard	GET /validationreport/{p-id}	[0]
Project Dashboard	GET /userlist?projectID={p-id}	[0, 1, 2]
Project Dashboard	GET /projectrevision/{p-id}	[0]
Project Dashboard	GET /project?id={p-id}&type={p-type}	[0, 2]
Coding Editor	GET /collectionresponse_code/{cs-id}	-
Coding Editor	GET /collectionresponse_textdocument/ {p-id}/{p-type}	-
Coding Editor	GET /listCodingsForProject?projectId= {p-id}&projectType={p-type}	-
Coding Editor	POST /batchProcess	[0, 1, 2]
Coding Editor	POST /code/{pc-id}	[0, 5]
Coding Editor	PUT /code	[0, 4, 5]
Coding Editor	DELETE /code/{c-id}	[0, 5]

[0]: Refactoring and cleanup without noteworthy performance improvements

[1]: Increased usage of memcache and datastore batch APIs (*GAE REQ 5*)

[2]: Improved utilization of memcache (*GAE REQ 2*)

[3]: Query superclass table with low-level datastore API instead of DataNucleus (*GAE REQ 6*)

[4]: JDO-object detached before multiple changes (*GAE REQ 4*)

[5]: Omitted explicit call to makePersistent after changing jdo-object (*GAE REQ 3*)

[*]: We applied these optimizations indirectly via code review on GitLab

Table 3.4: Overview of applied optimizations to 21 hot endpoints

3.3.1 Core Refactorings

In this section, we describe two endpoint optimizations in detail; both are prime examples of the significant performance benefits of the implementation of GAE REQ 5 and GAE REQ 6.

CodingEndpoint → batchProcess

This endpoint is responsible for persisting a list of codings in the datastore. To reiterate: A coding is an annotation applied to a document and a central part of the QDA process (see Section 2.2). Therefore, `batchProcess` is one of the most important and most frequently used endpoints in the GAE backend.

In the previous implementation, `batchProcess` iterated over the list of codings to:

1. Load the document to which the coding was applied (1 datastore read).
2. Authorize the user for the document (1 memcache get, assuming that the user was found in memcache).
3. Persist the coding (1 datastore write or delete).

Our optimization made two changes:

- Because documents and codings have a 1 - N relationship, the codings passed to `batchProcess` typically belonged to the same document. The previous implementation, therefore, loaded and authorized the same document multiple times. To prevent this redundancy, we only retrieved distinct documents and moved the authorization before the actual processing.
- We used both batch APIs of DataNucleus to insert and delete all codings at once and comply with GAE REQ 5.

Table 3.6 shows the savings in outgoing datastore or memcache requests as a result of the optimized version. For this experiment, we passed 17 codings to `batchProcess`, all belonging to the same text document.

Outgoing Request	**Calls Previous**	**Calls Optimized**
/datastore_v3.Get	17	1
/datastore_v3.Put	17	2
/memcache.Get	17	1

Table 3.5: Optimized datastore and memcache call frequency in batchProcess

Although the optimized version persists the codings with a single call to the batch API of DataNucleus, we saw two datastore writes in our traces (marked with blue in Table 3.6). This is because the upper size of batch inserts is limited to 10 by DataNucleus (profiling result from Section 3.1.4).

ProjectStatsEndpoint → getProjectStats

This endpoint returns the number of documents, codings, and other project statistics that are displayed on the user's project dashboard (Figure 3.10).

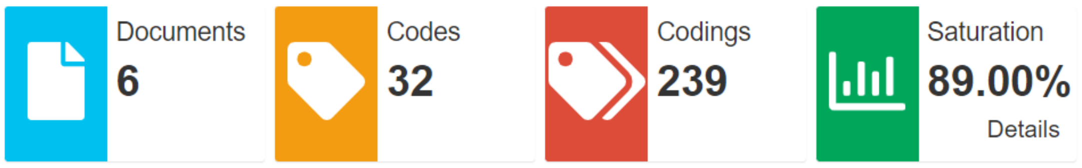


Figure 3.10: Project stats displayed on the project dashboard

The previous implementation used DataNucleus to query the "Coding" and "Document" datastore tables that hold entities of different subclasses. Section 3.2.1 shows that executing a query on a superclass table results in (1 + number of subclasses) datastore reads that are not performed in parallel.

To comply with GAE REQ 6, we refactored the endpoint to use the low-level datastore API. Additionally, because both queries run on independent tables, we used Java's executor service[12] to run them in parallel. We implemented the counting of entities and the parallel execution of queries as facade methods that hide the complexity of the low-level datastore API from the endpoint.

Using queries of the low-level datastore API that run in parallel and only return entity keys, we achieved a 78% decrease in request latency (see Table 3.6). We also reduced the number of datastore reads on the superclass tables from 10 ((1 + 5 coding subclasses) + (1 + 3 document subclasses)) to 2.

Time Previous	Time Optimized	Percentage Change
301 ms	67 ms	78% DECREASE

Table 3.6: Optimized request latency of getProjectStats

[12]https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

3.3.2 Preventing User Facing Loading Requests

This section describes our attempts to preload application code into a newly created instance, thereby protecting users from experiencing loading requests (see Section 3.1.7) and fulfilling GAE REQ 1.

User-facing loading requests are such a common problem that App Engine offers a feature called warmup requests. If this feature is enabled, App Engine issues a request to /_ah/warmup immediately after a new instance was created.

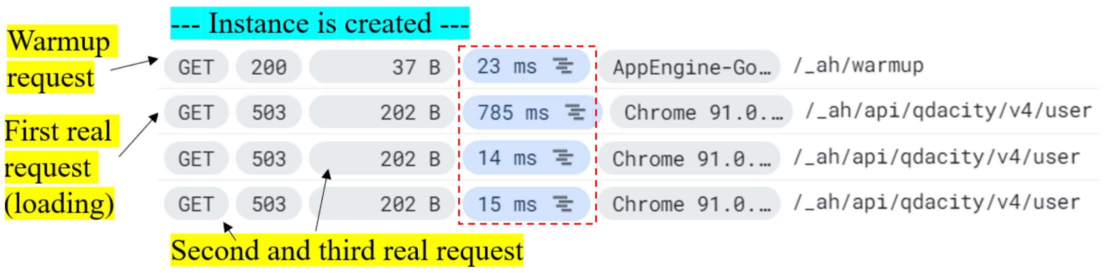
In QDAcity, we respond to this request with a custom warmup servlet that initializes and calls the most frequently used endpoints with dummy data (see Code B.2).

```
if (requestUrl.startsWith("/_ah/warmup") {
    UserEndpoint userEndpoint = new UserEndpoint();
    userEndpoint.getUser("fake-id", null);
    /* ... */
}
```

Code 3.9: Custom servlet responding to a warmup request

Our goal with this servlet was to load the application code into the fresh instance, establish connections with distributed App Engine services such as the datastore or memcache, and perform App Engine specific meta data validation for all classes managed by DataNucleus. This is an excellent solution on paper, but in practice, it does not prevent the first user request request from being a loading request. Figure 3.11 shows the logs of an experiment in which we created a fresh instance, triggered our custom servlet, and sent three user requests afterward to the same endpoint.

Even though the warmup servlet called the same method that handled the subsequent requests, it did not prevent the first user request from taking significantly longer than the second and third request.


--- Instance is created ---
Warmup request
First real request (loading)
Second and third real request

```
GET  200   37 B   23 ms   AppEngine-Go…   /_ah/warmup
GET  503  202 B  785 ms   Chrome 91.0.…   /_ah/api/qdacity/v4/user
GET  503  202 B   14 ms   Chrome 91.0.…   /_ah/api/qdacity/v4/user
GET  503  202 B   15 ms   Chrome 91.0.…   /_ah/api/qdacity/v4/user
```

Figure 3.11: Custom warmup servlet does not prevent loading request

Send Warmup Requests from Distributed Service

This section presents an alternative to the ineffective warmup servlet. Instead of calling the endpoints directly, the servlet issues a “warmup-qdacity” request to a distributed *warmup service*. Immediately after receiving the request, the warmup service issues a handful of user requests to essential QDAcity endpoints such as /user or /project. Figure 3.12 shows the communication between App Engine, the GAE backend, and the warmup service.

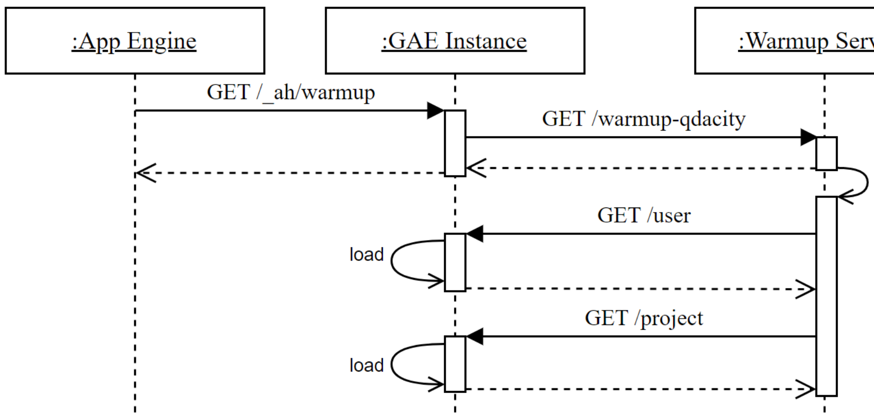
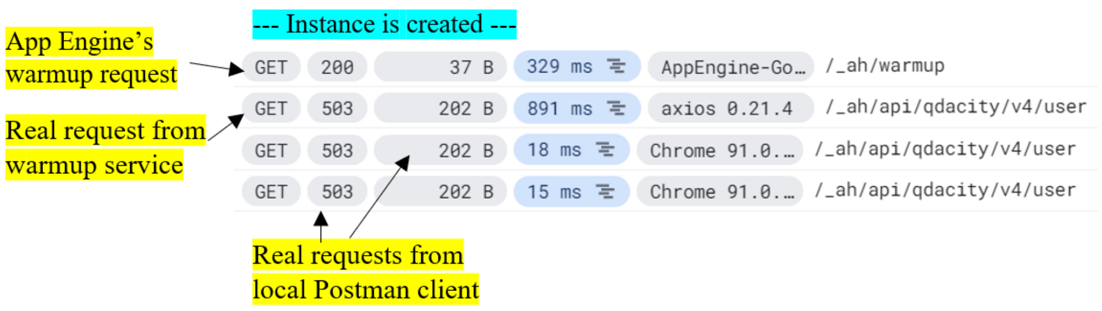


Figure 3.12: Communication of App Engine, GAE backend, and warmup service

To prove that this could work, we implemented a prototype of the warmup service as an Express[13] handler function hosted on Cloud Functions[14]. The prototype listened to GET requests at /warmup-qdacity and issued its own request to the user endpoint. The logs in Figure 3.13 show that this request was indeed a loading request with a significantly higher latency than the subsequent requests sent from a local Postman client. This prototype is not currently used in QDAcity and is not yet documented in a merge request on GitLab. We, therefore, added the servlet's code and the cloud function to Appendix B.


App Engine's warmup request
Real request from warmup service
Real requests from local Postman client

```
--- Instance is created ---
GET  200   37 B  329 ms  AppEngine-Go…  /_ah/warmup
GET  503  202 B  891 ms  axios 0.21.4   /_ah/api/qdacity/v4/user
GET  503  202 B   18 ms  Chrome 91.0.…  /_ah/api/qdacity/v4/user
GET  503  202 B   15 ms  Chrome 91.0.…  /_ah/api/qdacity/v4/user
```

Figure 3.13: Loading request issued by warmup service

[13]https://expressjs.com/
[14]https://cloud.google.com/functions

3.3.3 Cache Design

To encapsulate caching functionality and hide its complexity from the endpoint classes, the GAE backend used a facade[15] for accessing memcache. As a prerequisite for implementing GAE REQ 5 (increased memcache usage) and GAE REQ 2 (usage of memcache batching capabilities), we improved the facade's usability, increased its functionality, and enforced its consistent use across all endpoints that previously had accessed memcache directly. During code review, project management fittingly referred to these changes as *establishing an infrastructure* for future work.

Usability

We merged multiple overloaded methods that differed in the type of the id parameter into a single method that accepted a general object as id. Additionally, we updated the getter methods to return objects with generic types, thus eliminating the need for the caller to cast the retrieved objects explicitly. Code 3.10 shows an example where both of these changes come into play.

```
// Cache.java
Object get(String id, Class type) { }
Object get(Long id, Class type) { }

// Endpoint
Project p = (Project) Cache.get(1L,
                         Project.class);
```

```
// Cache.java
<T> T get(Object id, Class<T> type) { }

// Endpoint
Project p = Cache.get(1L,
                      Project.class);
```

Code 3.10: Comparison between old and new cache API

Functionality

We added four methods to the facade that provide a simple interface to the batching capabilities of memcache:

- `<T> List<T> getOrLoadAll(List<?> ids, Class<T> type)`: Tries to get the objects from Memcache. Every object that isn't found in Memcache, is loaded from the Datastore and cached afterward.
- `void cacheAll(Map<Object, ?> idToObject, Class<?> type)`
- `void invalidateAll(List<?> ids, Class<?> type)`
- `void invalidateUserLoginsAll(List<User> qdacityUsers)`: The same user object can be cached with different keys, depending on the service the current user logs in with (Google, Twitter, or Facebook). This method deletes every instance of the same user object from memcache.

[15]https://wiki.c2.com/?FacadePattern

3.4 Evaluation

GAE REQ 1: If a user sends the first request to a newly created instance, that request shall not be a loading request.

(✗) We did not satisfy this requirement.

Although App Engine's warmup request feature is a seemingly optimal solution to perform application-specific initialization tasks, we couldn't prevent the first user-facing request from being a loading request. Facing this problem became increasingly frustrating since it felt as if we were fighting against an entirely intransparent mechanism that prevented any actual initialization from being triggered from inside the warmup request. App Engine launched in April 2008, more than half a year earlier than Stack Overflow (November 2008). We believe that most companies who faced this problem back then either gave up or solved it internally without bothering to document the solution online.

Currently, QDAcity only has a handful of active users. The probability that a user experiences a loading request is therefore much higher than if hundreds of users accessed the instance every day. This rationale is not to downplay the problem but to note that loading requests will naturally occur less frequently once the userbase grows.

We presented a distributed warmup service as an alternative solution in Section 3.1.7, which would have to be implemented, maintained, and eventually paid for, making it an unsatisfactory solution.

GAE REQ 2: If an object exists in memcache and the datastore, the endpoints shall load that object from the memcache.

(✓) We satisfied this requirement.

We increased the number of datastore entities loaded from memcache in 6 hot endpoints, replacing DataNucleus queries with calls to our improved memcache facade (see Section 3.3.3). We limited our optimizations to objects that were frequently read and already existed in memcache, such as user or project entities. Using memcache instead of the datastore increased performance (see Section 3.1.6) but introduced new complexity because object states now had to be synchronized between both storage services. While implementing GAE REQ 2, we fixed multiple instances of missing or incorrect entity synchronization in different "cold" endpoints such as /removeProject.

GAE REQ 3: The endpoints shall not call `makePersistent()` explicitly after setting data to an attached object.

(✓) We satisfied this requirement.

Fulfilling this requirement was trivial. With transparent, persistence the explicit call to `makePersistent()` had no effect, and we could safely remove it without any fear of regression. We applied this optimization to three hot endpoints and removed four occurrences of the unnecessary call.

GAE REQ 4: The endpoints shall detach an attached object from the persistence manager before setting its data more than once in a row.

(✓) We satisfied this requirement.

We applied the optimization to one endpoint that was responsible for updating a code with a list of new values. In the previous implementation, the endpoint called 11 setter methods on the attached code. In our optimization, we detached the code before the set operations, which decreased request latency by 74% (562ms → 144ms), and reduced the number of datastore writes from 11 to 1.

GAE REQ 5: If the endpoints operate on independent entities, they shall use the batch APIs of memcache and the datastore.

(✓) We satisfied this requirement.

We increased the use of memcache and datastore batch APIs in three hot endpoints. Table 3.7 shows the additional uses of the batch APIs.

Storage Service	Batch API	Additional Uses
Datastore	DataNucleus → makePersistentAll	2
Datastore	DatastoreFacade.java → delete	1
Memcache	Cache.java → getOrLoadAll	2

Note: getOrLoadAll was a custom facade method described in Section 3.3.3.

Table 3.7: Additional use of memcache and datastore batch APIs

GAE REQ 6: The endpoints shall use the low-level datastore API instead of DataNucleus to query superclass tables.

(✓✗) We satisfied this requirement partially.

Three hot endpoints used DataNucleus to query a superclass table, which made them candidates for this optimization. Two of these endpoints, DocumentEndpoint → getTextDocument and CodingEndpoint → listCodingsForProject, returned domain objects such as codings and documents.

A query executed with the low-level datastore API returns an instance of the Entity[16] class, which is essentially a hash map of key-value pairs that can get and set using the instance's `getProperty()` and `setProperty()` methods. DataNucleus maps the Entity instance to the domain object behind the scenes, but with the low-level datastore API, this would have to be done manually (see Code 3.11).

```
DatastoreService ds = DatastoreServiceFactory.getDatastoreService();
Key k = KeyFactory.createKey("Project", 1L);
Entity e = datastore.get(k); // Entity is retrieved

Project p = new Project(); // Domain object from com.qdacity
p.setName(e.getProperty("name")); // Manual mapping of properties
p.setDescription(e.getProperty("description"));
// ...
```

Code 3.11: Manual mapping from entity to domain object

We decided not to optimize these two endpoints because the performance improvements did not justify the increased complexity of mapping entities to domain objects on our own.

We optimized the third candidate, ProjectStatsEndpoint → `getProjectStats()`. The endpoint was better suited to use the low-level datastore API since it only required meta-information about a query, such as the number of returned entities and not the fully mapped domain objects. By replacing DataNucleus with the low-level batch API described in Section 3.3.1, we reduced the endpoint's response time by 78% (301 ms → 67 ms).

[16]https://cloud.google.com/appengine/docs/standard/java/javadoc/com/google/appengine/api/datastore/Entity

3.5 Additional Refactoring

In addition to the quality requirements for performance, some parts of the code were also refactored for better maintainability.

Project Authorization

QDAcity has four project types: Project, Revision, ValidationProject, and ExerciseProject, all of them are derivatives of the abstract BaseProject class. There are different security requirements for all of these types. For example, to access a regular project, users need to own that project, but they have to be validation coders to access a validation project and so on. Most endpoints handled this problem in a similar way. A switch statement was used to determine the project type, load the appropriate project instance, and call the proper authorization method (see Code 3.12).

```
switch(projectType){
  case PROJECT:
       /* load project from "Project" table,
          check if user is owner */
  case VALIDATION_PROJECT:
       /* load project from "ValidationProject" table,
          check if user is validation coder */
  // ...
}
```

Code 3.12: Switch statement to determine project type

Because project authorization was necessary for almost every endpoint, this structure was repeated throughout the system and was the cause for some repeated boilerplate code. We made the refactoring in two steps. First, we removed the switch statement from the endpoints and buried it in the memcache facade, where it acted as a factory that loaded the appropriate project from the datastore or memcache. Secondly, we added a *template method*[17] to the abstract BaseProject class. The template method executed the default authentication steps and delegated the operations dependent on the project type to the subclasses.

With the refactored solution, endpoints obtained the project from the memcache facade and dispatched the authorization method polymorphically through the template method of the BaseProject class.

Even though code maintainability is a quality aspect, it's not directly related to our thesis goal performance. Therefore, we could have ignored this opportunity to repay technical debt and moved on to other tasks. But we believe in the boy

[17]https://wiki.c2.com/?TemplateMethodPattern

scout rule that says to “leave the campground cleaner than you found it,” and so we paid some of QDAcity’s technical debt with this refactoring (Martin, 2008).

Code Coverage and Good Test Quality

During implementation, which mainly consisted of enhancing existing code, it became apparent that high code coverage and good test quality were a prerequisite for any non-trivial refactoring. Without enough unit tests that captured an endpoint’s previous behavior, we could not make significant changes without risking functional regression. The GAE backend had a code coverage of 60%, which is decent, but coverage does not reflect the quality of individual tests, which were sometimes difficult to adjust.

To improve test maintainability, we used the builder[18] pattern to implement a builder for constructing complex project entities populated with test data. Additionally, we introduced a new naming convention for test methods and documented it in the GitLab wiki, allowing the QDAcity team to reference it in code reviews.

Old naming convention: `test{method-under-test}`

New naming convention: `{method-under-test}_{expected-behavior}_{condition}`

[18]https://wiki.c2.com/?BuilderPattern

4 Performance in the RTCS

4.1 RTCS Infrastructure

The RTCS is one of the three major components in QDAcity and acts as a proxy to the GAE backend. It enables real-time collaborative coding of the same document for multiple project members.

Real-time collaboration requires WebSockets for full-duplex communication and a server that can keep a coding session open for an extended period. The standard environment of App Engine doesn't support these features, and so the RTCS isn't part of the GAE backend but runs on a virtual machine (VM) on Compute Engine instead. Compute Engine satisfies both functional requirements from above but, because its infrastructure needs to be self-managed, comes with significant downsides regarding the following quality aspects:

- *Maintainability*: Keeping a custom VM up to date requires the maintainers to connect via SSH and install dependencies manually with the shell. That's why the VM's operation system, critical packages, NodeJS, and the direct dependencies of the RTCS were not updated regularly.
- *Versioning*: Compute Engine did not store previously deployed versions of the RTCS, making it difficult to roll back to a previous revision or gradually roll out a new version by splitting traffic between multiple revisions, which could open up the possibility of A/B testing.
- *SSL Certificate*: For the frontend to establish connections via HTTPS, an SSL certificate had to be manually created with Let's Encrypt and installed on the NGINX reverse proxy.
- *Monitoring*: The RTCS sent logs to Cloud Logging with a third-party client called winston[1], which required all RTCS developers to create a Service Account with "Logging Admin" permissions. The account's credentials had to be stored in a local configuration file making the already tricky setup more complicated.

[1]https://github.com/winstonjs/winston

These were the main reasons why QDAcity searched for a serverless solution for the RTCS, which could satisfy the following eight requirements:

- RTCS REQ 1: A QDAcity developer shall be able to deploy the RTCS to the serverless solution from a local machine and from the CI/CD pipeline.
- RTCS REQ 2: The serverless solution shall support WebSockets.
- RTCS REQ 3: The serverless solution shall not have a maximum request timeout making long coding sessions possible.
- RTCS REQ 4: The serverless solution shall automatically scale instances horizontally to handle all incoming requests.
- RTCS REQ 5: The serverless solution shall "scale to zero" when it handles no traffic enabling usage-based pricing.
- RTCS REQ 6: The serverless solution shall automatically provide an HTTPS endpoint through which the frontend can access the RTCS.
- RTCS REQ 7: The serverless solution shall store previously deployed versions of the RTCS to enable traffic splitting and rollbacks (versioning).
- RTCS REQ 8: The serverless solution shall automatically send all logs of the RTCS to Cloud Logging without using a third-party logging client.

Google Cloud offers four serverless solutions: Kubernetes Engine[2], App Engine Flex, App Engine Standard, Cloud Functions, and Cloud Run[3]. We decided against the first four solutions due to the following deal-breakers (see Table 4.1).

Serverless Solution	**Deal-breaker (the main reason against)**
App Engine Flex	Instances do not scale to zero during low traffic, which means at least one instance is always running — a sub-optimal cost model for QDAcity's inconsistent traffic (primarily accessed during EU daytime).
App Engine Standard	No support for WebSockets and a maximum request timeout of 60 seconds.
Kubernetes Engine	High complexity of container orchestration; unnecessarily high level of configurability.
Cloud Functions	No support for WebSockets and a maximum request timeout of 9 minutes.

Table 4.1: Deal-breakers of Google Cloud serverless solutions

[2]https://cloud.google.com/kubernetes-engine
[3]https://cloud.google.com/run

The other serverless solutions were eliminated due to the above deal-breakers, making Cloud Run the most suitable solution for the RTCS given our requirements. The rest of this chapter will show the performance characteristics of the RTCS on Cloud Run and its migration from Compute Engine.

4.2 Migration to Cloud Run

Cloud Run was released in November 2019 by Google and is a serverless solution for hosting backend applications. Cloud Run allows us to deploy and run containerized code written in any language with any dependencies. By using containers, Cloud Run overcomes the limitations of other cloud services such as App Engine that only offer a limited set of programming languages and runtimes. After Google added support for WebSockets to Cloud Run in February 2021[4], the service became a suitable serverless solution for the RTCS.

Containerizing the RTCS

We containerized the RTCS with the Dockerfile shown in Code 4.1. It used alpine[5] as a base image which is a minimal Linux distribution, and defined steps for installing Node, NPM and all direct dependencies of the RTCS.

```
FROM alpine:latest
WORKDIR /usr/src/app
COPY package*.json ./
RUN apk add --update nodejs npm
RUN npm ci --only=production
COPY . ./
CMD [ "node", "index.js" ]
```

Code 4.1: Dockerfile of the RTCS

The subsequent deployment of the RTCS was simple and only consisted of two steps. First, we built an image from the Dockerfile and published it to Google's Container Registry[6]. Then, we deployed the RTCS to Cloud Run by specifying the target project and the previously created image via its URL (see Code 4.2).

```
gcloud builds submit --tag gcr.io/$GCLOUD_PROJECT_ID/rtcs

gcloud run deploy rtcs --image=gcr.io/$GCLOUD_PROJECT_ID/rtcs
```

Code 4.2: Commands for building and deploying the RTCS

[4]https://cloud.google.com/blog/products/serverless/cloud-run-gets-websockets-http-2-and-grpc-bidirectional-streams

[5]https://hub.docker.com/_/alpine

[6]https://cloud.google.com/container-registry

4.2.1 Profiling Resource Usage

Cloud Run allows the limitation of a container's resource usage by setting limits for the number of CPUs and amount of memory a container shall use (see Table 4.2).

Resource	Description	Value Range
CPU	Number of vCPUs given to the container instance	[1 - 4]
Memory	Amount of RAM given to the container instance	[128 MiB - 8 GiB]

Table 4.2: Cloud Run resource limits

Finding the optimal limit for CPU and memory is challenging given the wide range of options. To better understand the RTCS' hardware usage, we conducted an experiment that we present in this section.

In the experiment, we incrementally increased the number of concurrent web-socket connections from 0 to 250 to determine the thresholds at which the instance needed additional hardware to handle the load. We picked 250 as an upper boundary because it's the maximum number of connections a container can maintain, limited by the *concurrent requests*[7] setting.

Based on these thresholds, we can determine the resource limits necessary to handle the expected number of connections and configure the container accordingly thus saving money for unnecessary container resources.

Simulating Traffic

To simulate the actual traffic that an active coder would create, we had each WebSocket connection emit a message to the RTCS in a 30-second interval. A connection would therefore emit ~20 messages throughout a 10-minute experiment. The messages consisted of an identifier for a common coding activity, the payload, and a callback function for the RTCS to call after successfully processing the message. The connection emitted the messages alternately in a round-robin fashion.

We used the cluster mode of PM2[8] to distribute the Node processes across five CPUs of our testing machine, each process handling a fifth of the total amount of connections. Because a single instance of NodeJS runs in a single thread, this was the only option to emit messages in parallel. Figure 4.2 illustrates the simulation.

[7]https://cloud.google.com/run/docs/about-concurrency
[8]https://pm2.keymetrics.io/docs/usage/cluster-mode/

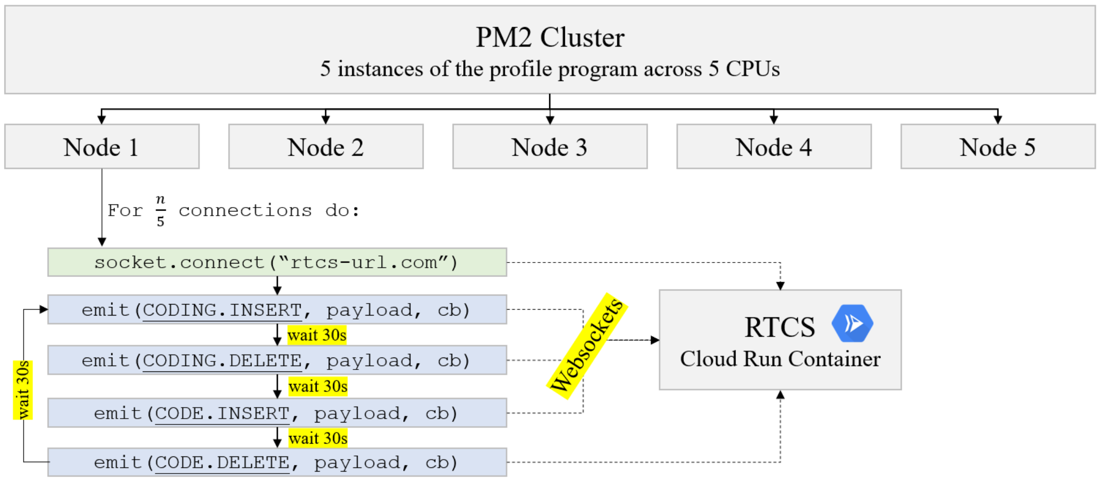

PM2 Cluster
5 instances of the profile program across 5 CPUs
Node 1
Node 2
Node 3
Node 4
Node 5
For $\frac{n}{5}$ connections do:
socket.connect("rtcs-url.com")
emit(CODING.INSERT, payload, cb)
wait 30s
emit(CODING.DELETE, payload, cb)
wait 30s
emit(CODE.INSERT, payload, cb)
wait 30s
emit(CODE.DELETE, payload, cb)
wait 30s
Websockets
RTCS
Cloud Run Container

Figure 4.1: Profiling the RTCS with multiple node instances

For each message, we measured the time from the initial emit until the RTCS used the callback. In the callback, passed to the emit function as a third argument, we used a NodeJS PerformanceObserver[9] to calculate and save the message's response time.

Confounding Parameter: Cloud Run auto-scaling

With this experiment, we aimed to analyze the resource consumption of *one* container. We, therefore, disabled Cloud Run's auto scaling mechanism by setting *max-instances* to 1. Without this settings, the load balancer would distribute the connections across multiple instances, making it impossible to trace which connections belong to which container.

Confounding Parameter: GAE Backend Response Times

So that the GAE backend would not become a bottleneck, we started three of the fastest instances (F4_1G) and warmed them up before the experiment. We removed any business logic from the endpoints to not fill the datastore with test data or accidentally delete existing entities. To simulate realistic response times, we let each endpoint sleep for 50 milliseconds before sending the response.

Results

We started the experiment by deploying a container with the least possible hardware resources, 128 MiB and 1vCPU. We increased the number of active connections while monitoring the container's resource consumption with the Cloud Run Dashboard. 128 MiB were enough to handle 0, 25, 50, and 75 connections, but after we added another 25 connections, totaling 100, the instance exceeded its

[9]https://bit.ly/3ogg17R

memory limit and was terminated by Cloud Run. We doubled the memory limit to 256 MiB and re-deployed the instance, starting with 100 active connections. No other re-deploy was necessary since the new instance could handle subsequent increases up to 250 concurrent connections, as Table 4.3 shows.

Active WebSocket Connections	**Memory Allocation**	**Utilization of 1 vCPU**	**Resource Limits (Memory, vCPUs)**
0	106 MiB (83%)	0.95%	128 MiB, 1 vCPU
25	115 MiB (90%)	1.95%	128 MiB, 1 vCPU
50	118 MiB (92%)	2.95%	128 MiB, 1 vCPU
75	122 MiB (95%)	3.95%	128 MiB, 1 vCPU
100	136 MiB (53%)	5.95%	256 MiB, 1 vCPU
150	148 MiB (58%)	10.95%	256 MiB, 1 vCPU
200	161 MiB (63%)	13.95%	256 MiB, 1 vCPU
250	179 MiB (70%)	17.95%	256 MiB, 1 vCPU

Note 1: With col 1 and 2, we can calculate the average memory size of a single connection: ~500 KiB

Note 2: Row 1 shows that the container uses 83% of memory for the RTCS alone, without connections

Note 3: 250 is the limit for concurrent connections, so resource limits above 256 MiB, 1 vCPU are pointless

Table 4.3: RTCS resource usage

For 50, 150, and 250 connections, median response times ranged from 224 ms to 234 ms and are illustrated for all four message types in Figure 4.2. We attribute the "drop" in the middle due to the fact that we switched to a more powerful instance after 75 connections (see above).

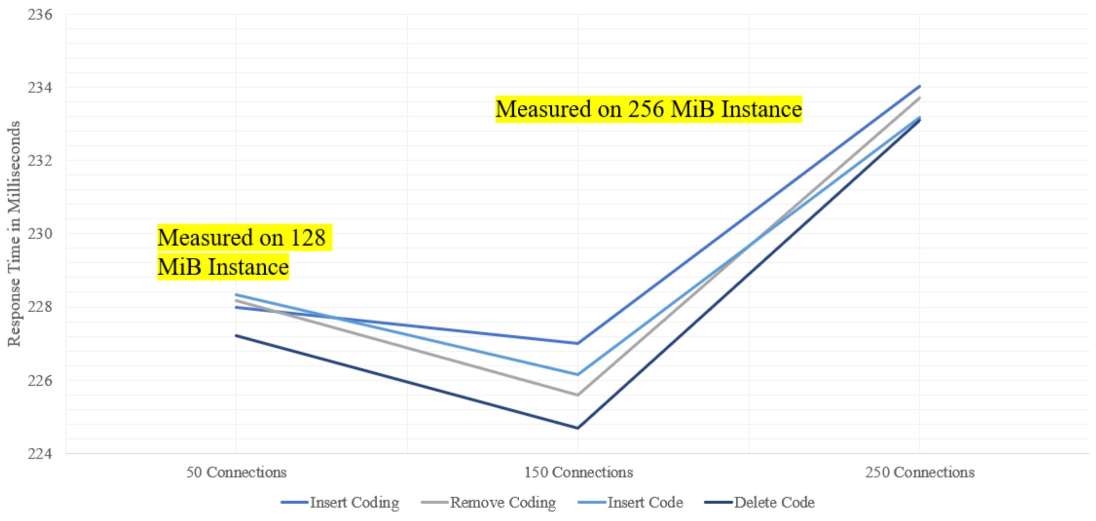


Figure 4.2: RTCS response times for 50, 150, and 250 connections

4.3 Evaluation of Migration

In this section, we compare the RTCS infrastructure requirements from section 4.1 with the result of the Cloud Run migration.

RTCS REQ 1: A QDAcity developer shall be able to deploy the RTCS to the serverless solution from a local machine and from the CI/CD pipeline.

(✓) We satisfied this requirement.

The configuration for Compute Engine consisted of 452 lines within ten files, ranging from deployment scripts that uploaded the latest RTCS version to the VM, over custom configuration for the NGINX[10] reverse proxy, to scripts for generating an HTTPS certificate with Let's Encrypt[11]. We replaced the existing configuration with 57 lines in three files, reducing the total RTCS configuration by 87%. The updated deployment configuration consisted of two files for local deployment from Windows or Linux and an additional file for deploying via the CI/CD pipeline on GitLab.

RTCS REQ 2: The serverless solution shall support WebSockets.

(✓) We satisfied this requirement.

Cloud Run has supported WebSockets since February 2021.

RTCS REQ 3: The serverless solution shall not have a maximum requests timeout making long coding sessions possible.

(✓✗) We satisfied this requirement partially.

In Cloud Run, WebSockets are treated as long-running HTTP requests, making them subject to a request-timeout policy, which automatically disconnects the WebSocket after 60 minutes. We could mitigate this problem by taking advantage of the auto-reconnect feature of Socket.IO v2.x[12], which causes the client socket to attempt a reconnect immediately after Cloud Run disconnected the socket. However, the socket will not reconnect automatically in later versions of Socket.IO (e.g., v3.x), which the developers need to be aware of when upgrading the library.

[10]https://www.nginx.com/
[11]https://letsencrypt.org/
[12]https://socket.io/docs/v2/client-api

RTCS REQ 4: The serverless solution shall automatically scale instances horizontally to handle all incoming requests.

(✓) We satisfied this requirement.

Cloud Run scales each revision automatically to the number of instances required to handle the incoming requests.

Using the method from Section 4.2.1, we profiled Cloud Run's auto-scaling mechanism by establishing 25 WebSocket connections with the RTCS while monitoring the amount of newly created instances. We found that the number of newly created instances depended heavily on whether an already active instance existed prior to the experiment.

For a "cold start," without an active instance, Cloud Run initially started 13 instances, which were then scaled back to 5 active instances after about 5 minutes (see Figure 4.3).

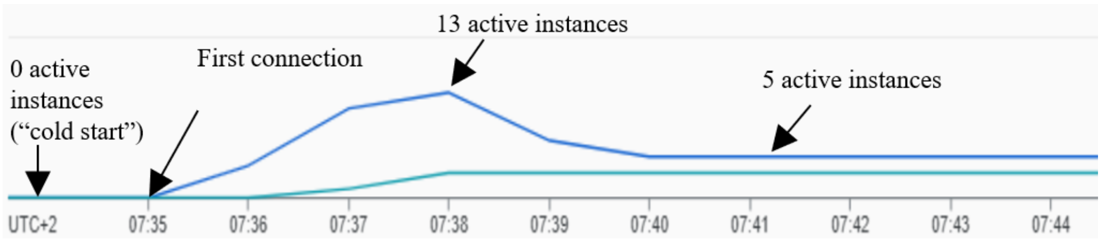


Figure 4.3: Cloud Run auto-scaling for 25 connections (cold start)

For a "warm start," with an existing active instance, Cloud Run did not create any new instances and let the existing one handle all 25 connections.

In App Engine, the paste in which instances are created can be controlled via the max-pending latency parameter, which defines the maximum time that App Engine allows a request to wait in the request queue before starting a new instance to handle it. In Cloud Run, this parameter does not exist, which gives us less control over auto-scaling.

RTCS REQ 5: The serverless solution shall "scale to zero" when it handles no traffic.

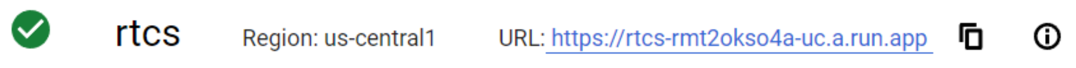
(✓) We satisfied this requirement.

This is the default behavior of Cloud Run - instances are shut down approximately 15 minutes after handling the last request. We can change this default to keep one or many instances idle with the minimum-instances setting.

RTCS REQ 6: The serverless solution shall automatically provide an HTTPS endpoint through which the frontend can access the RTCS.

(✓) We satisfied this requirement.

By adding the allow unauthenticated access flag in the deploy command, we told Cloud Run to create a unique HTTPS endpoint through which the RTCS can be accessed from the public internet (see Figure 4.4).



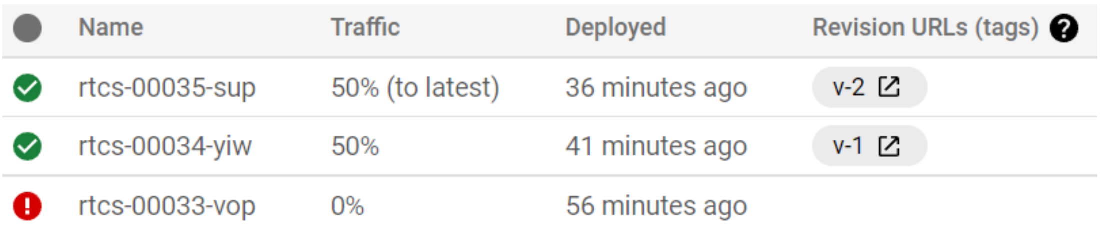
✓ rtcs Region: us-central1 URL: https://rtcs-rmt2okso4a-uc.a.run.app

Figure 4.4: RTCS HTTPS endpoint on Cloud Run

RTCS REQ 7: The serverless solution shall store previously deployed versions of the RTCS to enable traffic splitting and rollbacks (versioning).

(✓) We satisfied this requirement.

Cloud Run stores the last 50 RTCS deployments, including the RTCS image and the container configuration in the revision history. We can use this history to define traffic percentages received by a revision, making it easy to roll back to a previous revision or split traffic between multiple revisions. As shown in Figure 4.5, where we split traffic evenly between the last two RTCS deployments.



●	Name	Traffic	Deployed	Revision URLs (tags)
✓	rtcs-00035-sup	50% (to latest)	36 minutes ago	v-2
✓	rtcs-00034-yiw	50%	41 minutes ago	v-1
!	rtcs-00033-vop	0%	56 minutes ago	

Figure 4.5: Cloud Run versioning

RTCS REQ 8: The serverless solution shall automatically send all logs of the RTCS to Cloud Logging without using a third-party logging client.

(✓) We satisfied this requirement.

With Compute Engine, collecting the VM's logs required a third-party client library called winston. With Cloud Run or App Engine, everything written into the standard output gets picked up automatically by Cloud Logging, so we removed winston from the RTCS and replaced it with calls to `console.log()`.

5 Performance in the Frontend

This chapter describes the effects of six optimizations on the response time and resource utilization of the frontend. We measured their impact against a performance baseline which we describe in the following section.

5.1 Creating a Baseline

We established a baseline to measure the effects of subsequent changes; a snapshot of the frontend's current response time and bundle size. To create the response time baseline, we used Lighthouse, an open-source tool for auditing websites[1]. Typically, Lighthouse is run manually from the developer tools of chromium-based browsers, but we used it as a Node module which enabled us to:

- Run audits automatically, thus making it easy to increase sample size and decrease fluctuation between test runs,
- throttle device settings such as bandwidth and device type,
- get access to metrics not displayed in the UI, such as the execution and download times of specific JavaScript bundles.

We used a network profile with 10240 Kbps bandwidth which corresponds to the download speed of a slow 4G/LTE mobile connection[2]. Device CPU and memory were throttled to a lighthouse benchmark index of 2000 which resembles the hardware specification of a 2019 16" Macbook Pro (6-Core Intel Core i7, AMD Radeon Pro 5300M, 16 GB RAM)[3].

After each run, we reset the headless chrome browser to not influence subsequent runs with previously cached resources.

[1]https://github.com/GoogleChrome/lighthouse
[2]https://www.opensignal.com/reports/2018/02/state-of-lte
[3]https://github.com/GoogleChrome/lighthouse/blob/master/docs/throttling.md

Lighthouse Performance Score

For each run, lighthouse calculates a performance score between 0 - 100, which is the weighted average of the following six performance metrics:

Metric	Weight	Description
TBT (Total Blocking Time)	30%	The total time the main thread was blocked from handling input events.
LCP (Largest Contentful Paint)	25%	Time until the page rendered its main content (e.g., the landing page image).
CLS (Cumulative Layout Shift)	15%	Time that the page's elements moved around during load.
FCP (First Contentful Paint)	10%	Time until the page rendered the first visible element.
TTI (Time To Interactive)	10%	Time until the page became fully interactive.
Speed Index	10%	Average time at which visible parts of the page are displayed.

Table 5.1: Lighthouse performance metrics

TBT, LCP, and CLS are weighted higher by lighthouse because they are *core web vitals* - a set of performance metrics that, according to Google, are best suited for quantifying user experience[4]. We used TBT as a substitute for the official core web vital FID (First Input Delay), to which it has a positive correlation[5]. In the following sections, we focus on these three metrics (TBT, LCP, and CLS).

Bundle Analysis

We used Source Map Explorer[6] to visualize the frontend's minified JavaScript bundle as a treemap (see Figure 5.1). The treemap was an essential tool to understand the bundle's internal structure and measure the size of specific React components or third-party libraries.

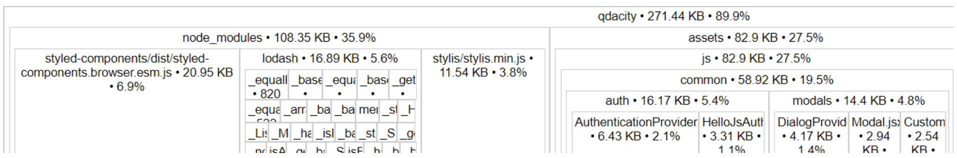

qdacity • 271.44 KB • 89.9%
node_modules • 108.35 KB • 35.9%
styled-components/dist/styled-components.browser.esm.js • 20.95 KB • 6.9%
lodash • 16.89 KB • 5.6%
stylis/stylis.min.js • 11.54 KB • 3.8%
assets • 82.9 KB • 27.5%
js • 82.9 KB • 27.5%
common • 58.92 KB • 19.5%
auth • 16.17 KB • 5.4%
modals • 14.4 KB • 4.8%
AuthenticationProvider • 6.43 KB • 2.1%
HelloJsAuth • 3.31 KB • 1.1%
DialogProvid • 4.17 KB • 1.4%
Modal.js • 2.94 KB •
Custom • 2.54 KB •

Figure 5.1: Treemap of frontend bundle; generated with Source Map Explorer

[4]https://web.dev/vitals
[5]https://web.dev/fid
[6]https://github.com/danvk/source-map-explorer

5.1.1 Results

Frontend Performance Baseline

Table 5.2 shows the median values of 200 Lighthouse runs. The performance score was color-coded based on the following thresholds by Google[7]:

- 0 to 49 (red): Poor
- 50 to 89 (orange): Needs Improvement
- 90 to 100 (green): Good

Row four shows the time it took for the browser's JavaScript engine (V8[8]) to parse, compile and execute the frontend's main bundle *index.dist.js*. Details such as the execution time of a specific bundle are only accessible with Lighthouse's Node module.

Metric	**Value**
TBT (Total Blocking Time)	401 ms
LCP (Largest Contentful Paint)	3491 ms
CLS (Cumulative Layout Shift)	0 ms
Parse, Compile, Execute *index.dist.js*	1118 ms
Performance Score	🟥 47 (poor)

Table 5.2: Frontend performance baseline

Frontend Bundle Size Baseline

The frontend consisted of the main bundle, a CSS stylesheet, code for two service workers, and other web resources such as images and fonts (see Table 5.3).

File	**Size**
index.dist.js (main bundle)	4538 KB
styles.css	142 KB
pdf.worker.dist.js	619 KB
service-worker/sw.dist.js	181 KB
Others (Images, Fonts)	213 KB

Table 5.3: Frontend bundle size baseline

[7]https://web.dev/performance-scoring
[8]https://v8.dev

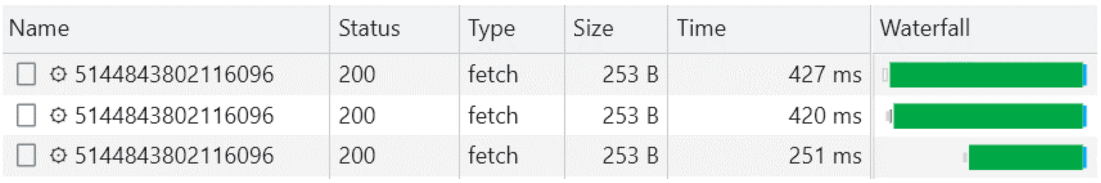
5.2 Quality Requirements

We defined the following quality requirements to improve frontend performance:

FE REQ 1: The frontend shall receive a lighthouse performance score of at least 90 (“good”) when measured using the method described in Section 5.1.

FE REQ 2: The frontend’s main pages (Personal Dashboard, Project Dashboard, Coding Editor) shall make no redundant network requests on load.

The frontend sent several requests during page load, although they did not differ in their payload nor in the response which they received from the backend. Figure 5.2 shows how the coding editor loads the same code system three times in a row during page load.



Name	Status	Type	Size	Time	Waterfall
☐ ⚙ 5144843802116096	200	fetch	253 B	427 ms	🟩
☐ ⚙ 5144843802116096	200	fetch	253 B	420 ms	🟩
☐ ⚙ 5144843802116096	200	fetch	253 B	251 ms	🟩

Figure 5.2: Three redundant network requests during page load

To improve resource utilization, bandwidth being the resource, the frontend shall omit these unnecessary requests.

FE REQ 3: The frontend shall be bundled with version 5 of webpack.

This requirement was based on a note in the webpack 5 documentation that an upgrade may reduce bundle size due to improved tree shaking, a dead-code elimination technique[9].

[9]https://webpack.js.org/guides/tree-shaking

5.3 Implementation

5.3.1 Webpack 5 Upgrade

We upgraded webpack v4.28.4 to v5.48.0. The upgrade required a babel version of at least 7; QDAcity was using v6.23.3. We used the open-source tool babel-upgrade[10] to automatically upgrade seven babel dependencies and the .babelrc configuration file. Overall, we deleted 14 node modules, added nine, and bumped versions of 6 already existing packages.

The upgrade decreased the size of the main bundle by 172 KB, a 4% reduction. A positive side effect was that the amount of unmapped code in the bundle was reduced by 329 KB and could therefore be visualized with source map explorer. The upgrade negatively impacted the frontend's total build time which increased by ~146% (28s $\rightarrow$ 69s).

5.3.2 Code-splitting

Code-splitting is a technique for splitting an application's main JavaScript bundle into various smaller files, called chunks. The app can then lazy-load these chunks on-demand, commonly based on user activity, e.g., page navigation. Code-splitting doesn't reduce the overall bundle size but reduces the amount of code needed during the initial load and, therefore, can lead to faster initial load times for first-time visitors. The feature is supported by bundlers like browserify[11], rollup[12], or webpack.

This section presents two approaches for splitting the frontend's main bundle with webpack.

First Approach: Multiple Small Chunks

We set up route-based code-splitting for the components PersonalDashboard, ProjectDashboard, and CodingEditor, which contain the frontend's primary functionality. Route-based code-splitting means that the component will be lazy-loaded when the user navigates to a specific page. This resulted in four files, the main bundle, and three chunks, one for each component.

While loading the currently requested component, the frontend displayed a blank page or a loading animation. We concluded that these repeated loading breaks interrupted the expected workflow from a UI perspective. We, therefore, discarded this initial prototype.

[10]https://github.com/babel/babel-upgrade
[11]https://github.com/browserify/factor-bundle
[12]https://rollupjs.org/guide/en/

Second Approach: Single Chunk for Core Functionality

In a second approach, we extracted core components, only accessible to the user after successful sign-in, from App.jsx, which is the root of the component tree. We moved such components into a new component, called Core.jsx, which we then split from the main bundle based on a route. The frontend loaded Core.jsx as a single chunk after the user accessed the core functionality, e.g., navigated away from the frontend's landing page.

Figure 5.3 illustrates the initial composition of App.jsx and the addition of the lazy-loaded Core.jsx component. We attenuated the number of components in Core.jsx to make the point (it contained more than 20 components).

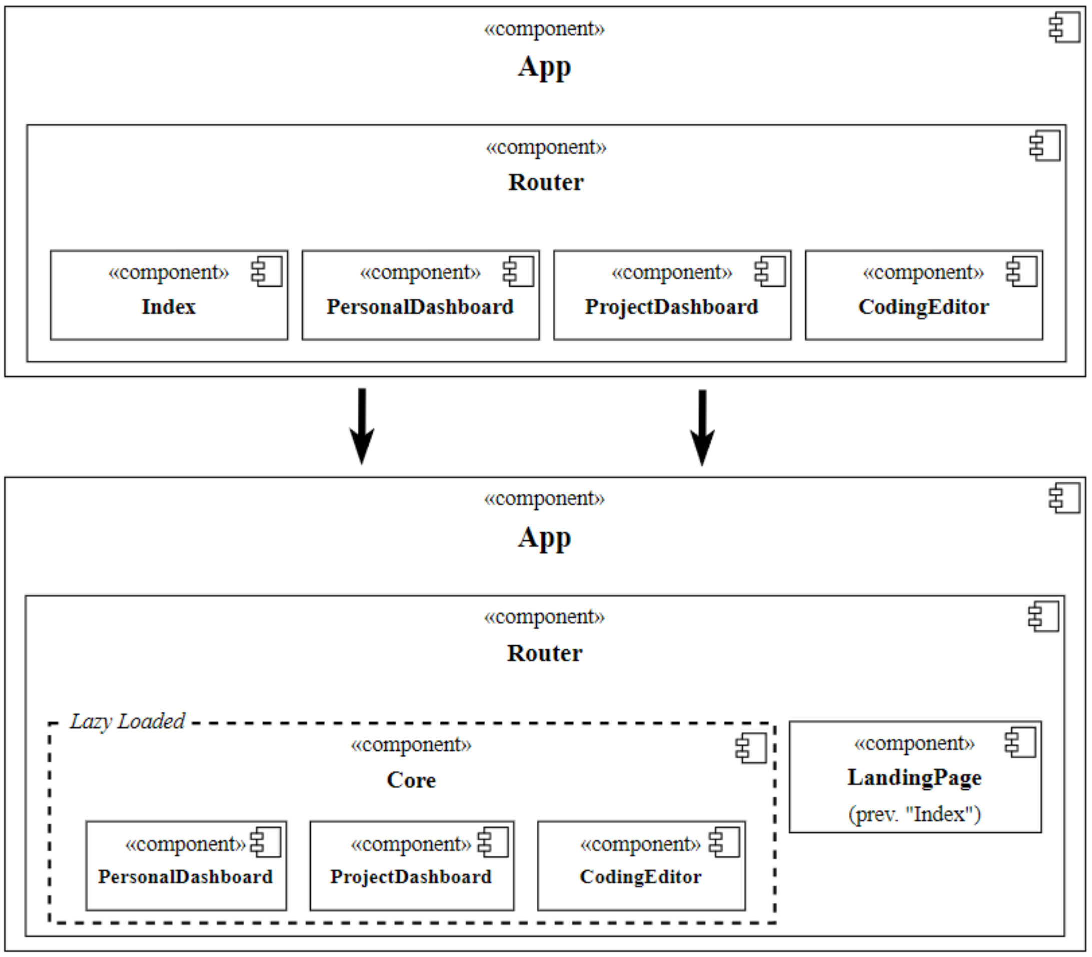


Figure 5.3: Extraction of core components from App.jsx

Effects on Bundle Size:

The newly created chunk of Core.jsx was 3902 KB, shrinking the main bundle to 682 KB. Together, the two files were 46 KB larger than the previous main chunk because webpack required some overhead to manage the extracted chunk.

Effects on Performance:

With code-splitting, the frontend no longer had to load the app's entire JavaScript but only 682 KB necessary to display the loading page. This increased the overall performance score by 82% (see Table 5.4).

Metric	**Baseline**	**Optimized**	**Percentage Change**
TBT (Total Blocking Time)	401 ms	67 ms	83% DECREASE
LCP (Largest Contentful Paint)	3491 ms	1247 ms	64% DECREASE
CLS (Cumulative Layout Shift)	0 ms	0 ms	-
Parse, Compile, Execute *index.js*	1118 ms	342 ms	69% DECREASE
Performance Score	47	86	82% INCREASE

Table 5.4: Frontend performance after code-splitting

5.3.3 Bundle-splitting

The previous optimizations focused on improving the performance of the initial page load, which primarily benefits first-time visitors. In this section, we optimize for long-term engagement by improving performance for QDAcity users who regularly visit the site.

The code in the frontend changes at different paces. While application code (e.g., code in our own React components) changes with almost every release, vendor code in our node modules only changes rarely. This dynamic resembles the water speed in a river where the friction with the riverbed decreases the velocity causing the water to flow faster near the surface and slower near the ground. However, the river seems to be flowing at the same pace from the outside (see Figure 5.4).

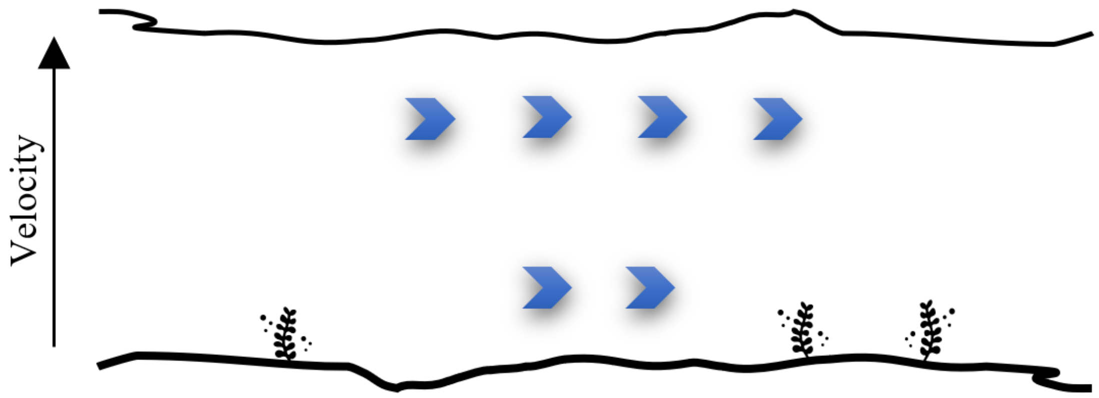


Figure 5.4: Water flows faster near the surface and slower near the riverbed

When QDAcity releases a new version, the version number is added to the name of the new bundle (e.g., index-89.dist.cache.js). This is a common cache-busting strategy that causes the browser to discard its current bundle and load the new one. This means that the user must download a new bundle for each release, which can be very inefficient if only a few lines are changed while everything else remains the same.

We reduced the amount of code a user has to download by moving large and rarely changing vendor code into separate chunks. This technique is a lesser-known feature of webpack and is called bundle splitting. For each vendor chunk, we added a hash of its content to bust the cache if the code inside ever happened to change. We extracted 328 KB of vendor code from the frontend's main bundle, including libraries like react or react-dom. Additionally, we extracted a vendor chunk with 2862 KB from the lazy-loaded core components chunk, which we added in Section 5.3.2.

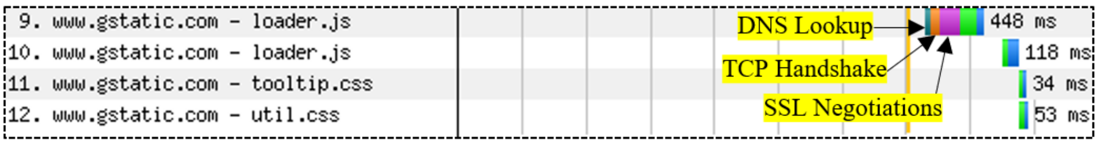
Table 5.5 shows the long-term effects of bundle splitting in a hypothetical example in which QDAcity releases every week and only changes code within the main bundle or the core chunk. With bundle splitting, the user has to download 11800 KB less over five weeks.

	File	Week 1	Week 2	Week 3	Week 4	Week 5
Baseline	index-v.dist.js	682 KB	682 KB	682 KB	682 KB	682 KB
	core-v.dist.js	3902 KB	3902 KB	3902 KB	3902 KB	3902 KB
Bundle-splitting	index-v.dist.js	354 KB	354 KB	354 KB	354 KB	354 KB
	index.vendor.[hash].js	328 KB	0 KB	0 KB	0 KB	0 KB
	core-v.dist.js	1232 KB	1232 KB	1232 KB	1232 KB	1232 KB
	core.vendor.[hash].js	2862 KB	0 KB	0 KB	0 KB	0 KB

Table 5.5: Long-term effects of bundle-splitting

An alternative we considered was to load vendor code from a Content Delivery Network (CDN), which could decrease load times if popular libraries like React, jQuery, or Bootstrap[13] were already stored in the visitors' browser cache. We found the probability that visitors have a library with the same version (e.g., Bootstrap v3.4.2) and from the same CDN in their cache too small to justify the downsides of using a CDN such as:

1. Each CDN becomes an additional point of failure.
2. Difficult offline development unless resources already exist in the browser cache.
3. For the first request to a new CDN, the browser has to establish a connection which causes overhead (see Figure 5.5). If many different CDNs are used, this could become a performance bottleneck.



```
 9. www.gstatic.com - loader.js      448 ms
10. www.gstatic.com - loader.js      118 ms
11. www.gstatic.com - tooltip.css     34 ms
12. www.gstatic.com - util.css        53 ms
```

Figure 5.5: Overhead of the first request to a new CDN

4. No deprecation warnings for outdated and broken libraries.

[13]https://github.com/twbs/bootstrap

5.3.4 Page Refresh During ServiceWorker Installation

By default, an activated service worker only starts controlling a page after a second refresh and doesn't handle messages or intercept requests until then. We avoided the refresh by allowing the worker to control the page with `clients.claim()` and by sending messages asynchronously (see Code 5.1).

```
// Old: Service worker is called directly.
navigator.serviceWorker.controller.postMessage(message);

// New: Service worker is called asynchronously via callback.
navigator.serviceWorker.ready.then(({active}) => active.postMessage(message));
```

Code 5.1: Sending a message to a service worker via callback

Without the refresh, the performance score increased by 38% (see Table 5.6).

Metric	Baseline	Optimized	Percentage Change
TBT (Total Blocking Time)	401 ms	102 ms	75% DECREASE
LCP (Largest Contentful Paint)	3491 ms	2981 ms	15% DECREASE
CLS (Cumulative Layout Shift)	0 ms	0 ms	-
Parse, Compile, Execute *index.js*	1118 ms	501 ms	55% DECREASE
Performance Score	47	65	38% INCREASE

Table 5.6: Frontend performance after optimizing service worker installation

5.3.5 Removing jQuery

The frontend was initially built with jQuery and has been gradually refactored into a React app since then. The library was still used in five components and therefore remained in the bundle with 83 KB. We removed jQuery from these components primarily to decrease bundle size but also to improve maintainability.

In React, data "flows down," meaning that parent components can pass data or functions via props to their children but not vice versa. Child components can emit events by calling functions on their props but can't access the components above them directly. With jQuery, we can bypass this one-way information flow by directly selecting UI elements and applying modifications to them even though they belong to the output of a parent component. This increases coupling and makes it challenging to track UI changes because any component in the entire hierarchy could have triggered them.

Because jQuery bypassed the regular communication between components, we had to build the communication from the ground up, making refactoring difficult.

We removed jQuery from five components, making the library disappear from the entire application and decrease bundle size by 83 KB.

5.3.6 Omit Redundant Requests

The following components issued redundant requests to the GAE backend:

- CodeSystem.jsx (Coding Editor) requested the project's code system 10 - 12 times during page load.
- AgreementModal.jsx (Project Dashboard) initiated an infinite loop of requests, which could only be broken by navigating to another page.
- Navbar.jsx (Global Component) issued a request to the user endpoint each time a modal was opened or closed anywhere in the frontend.
- Navbar.jsx (Global Component) requested the current user five times during page load.

The root cause was the same in all four cases. React re-renders a component if its props or state change which is very often the case. This becomes a performance problem if the component performs CPU-heavy tasks during the re-rendering or issues network requests to initialize state in the case of the components above.

We optimized the re-render behavior of Navbar.jsx by moving the user state up to a parent component (App.jsx at the top of the hierarchy) which meant that it no longer had to be initialized by the re-rendering Navbar.jsx through a network request.

In CodeSystem.jsx and AgreementModal.jsx, we moved any initialization tasks from the render method into a life cycle hook guarded by appropriate conditionals to prohibit multiple executions. This optimization caused CodeSystem.jsx to load the project's code system only once instead of 10 - 12 times and prevented AgreementModal.jsx from initializing an infinite loop of requests.

5.4 Evaluation

FE REQ 1: The frontend shall receive a lighthouse performance score of at least 90 ("good") when measured using the method described in Section 5.1.

(✓) We satisfied this requirement.

Table 5.7 shows the final measurement after all applied optimizations.

Metric	**Baseline**	**Optimized**	**Percentage Change**
TBT (Total Blocking Time)	401 ms	51 ms	87% DECREASE
LCP (Largest Contentful Paint)	3491 ms	1127 ms	68% DECREASE
CLS (Cumulative Layout Shift)	0 ms	0 ms	-
Parse, Compile, Execute *index.js*	1118 ms	97 ms	91% DECREASE
Performance Score	47	95	102% INCREASE

Table 5.7: Frontend performance after all optimizations

FE REQ 2: The frontend's main pages (Personal Dashboard, Project Dashboard, Coding Editor) shall make no redundant network requests on load.

(✓✗) We satisfied this requirement partially.

Component	**Previous Behavior**	**Optimized Behavior**
CodeSystem.jsx	Requested the code system 10 - 12 times during load	Was reduced to a single request
AgreementModal.jsx	Started an infinite loop of requests when opened	Does not initiate a loop and initializes state with a single request
Navbar.jsx	Loaded the current user five times during page load, and whenever a user opened or closed a modal anywhere in the frontend	Does not manage the user state anymore (we moved its state up to App.jsx), so it is unnecessary to initialize it.

Table 5.8: Frontend optimizations to reduce redundant requests

App.jsx still loads the current user, previously managed by Navbar.jsx, five times during page load, which is why this requirement was only partially satisfied.

FE REQ 3: The frontend shall be bundled with version 5 of webpack.

(✓) **We satisfied this requirement.**

Webpack was successfully upgraded from v4.28.4 to v5.48.0, which caused the bundle to shrink by 4%, improved visualization but increased build times by 169%. As a prerequisite for the webpack upgrade, we also bumped babel from v6.26.3 to v7.14.8.

6 Conclusions

In "Performance in the GAE Backend," we have shown that

- entities can be retrieved faster from memcache than from the datastore,
- ignoring JDO's transparent persistence mechanism leads to higher request latency and unnecessary RPCs,
- a DataNucleus query on a superclass tables issues (1 + number of subclasses) RPCs, which can be reduced to a single one with the low-level datastore API,
- using the batch APIs of datastore and memcache decreases latency and the number of RPCs,
- the first user request issued to a new App Engine instance faces high latency due to initialization tasks performed by the instance.

We drew on these findings to optimize performance in the most frequently used ("hot") endpoints and achieved significant performance improvements in the case of batchProcess (51 $\rightarrow$ 4 RPCs) and getProjectStats (78% latency decrease). In addition, we presented a prototype for a distributed warmup service that can prevent loading requests.

In "Performance in the RTCS," we have described the successful migration of the RTCS from Compute Engine to Cloud Run. We evaluated the migration result by comparing it against QDAcity's requirements for a serverless infrastructure. We backed the migration by our profiling results showing that a Cloud Run instance with the lowest resource limits (1vCPU, 128 MiB Memory) could already handle up to 75 concurrent WebSocket connections.

In "Performance in the Frontend," we have described six optimizations through which the frontend's performance score increased by 102% (47 $\rightarrow$ 95). We achieved the most-significant reduction of initial load time by splitting core functionality into separate JavaScript chunks, which the frontend then loaded on-demand. We measured our optimizations' impact against a baseline that we created with the lighthouse node module and a bundle analyzer.

Appendices

A *Formula*: Percentage Increase or Decrease from One Value to Another

In this thesis, we used the following formula to calculate the increase or decrease from an original value to a final value in a percent:

$$\left|\frac{\textit{original value} - \textit{final value}}{\textit{original value}}\right| \times 100 = \text{percent change}$$

The percent change of the formula will be

- an <u>increase</u> if the original value is less than the final value,
- a <u>decrease</u> if the original value is greater than the final value.

Example 1: original value $= 20$, *final value* $= 30$

$$\left|\frac{20-30}{20}\right| \times 100 = 50\% \text{ INCREASE}$$

Example 2: original value $= 45$, *final value* $= 15$

$$\left|\frac{45-15}{45}\right| \times 100 = \sim 66,6\% \text{ DECREASE}$$

B Warmup Service Prototype

Java servlet at /src/com/qdacity/servlet/WarmupServlet.java

```java
import com.google.appengine.api.urlfetch.*;

public class WarmupServlet extends HttpServlet {

  @Override
    void doGet(HttpServletRequest req, HttpServletResponse resp) {
          if (!req.getRequestURI().startsWith("/_ah/warmup")) {
            resp.sendError(404);
                return;
          }

          HTTPRequest doWarmupRequest = new HTTPRequest(
            new URL("https://<warmup-service>.net/warmup-qdacity"));

          URLFetchService fetcher = URLFetchServiceFactory.getURLFetchService();
          fetcher.fetch(doWarmupRequest);
        }
}
```

Code B.1: Custom servlet responding to a warmup request

Warmup service as a Express handler function on Cloud Functions

```javascript
const axios = require('axios');
const url = require('url');

exports.warmupQdacity = (req, res) => {
  /* Send the response immediately, issue the requests afterwards */
  res.end();

  const userEndpointUrl = `${getFullUrl(req)}user`;

  axios.get(userEndpointUrl, {
    // Trigger token validation in QdacityAuthenticator.
    headers: { Authorization: 'Bearer ABCD' },
  });
};

function getFullUrl(req) {
  return url.format({ protocol: req.protocol,
                      host: req.get('host'),
                      pathname: req.originalUrl});
}
```

Code B.2: Warmup Service on Cloud Functions

List of Figures

List of Tables

List of Codes

References

Abels, E. G., Domas White, M. & Hahn, K. (1997). Identifying user-based criteria for web pages. *Internet Research*, *7*(4), 252–262. https://doi.org/10.1108/10662249710187141

Bazeley, P. (2013). *Qualitative data analysis: Practical strategies*. SAGE Publications Ltd.

Boehm, B. (1987). Industrial software metrics top 10 list. *IEEE Software*, *4*(5), 84–85. https://doi.org/https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&amp;arnumber=1695831

Buyya, R., Broberg, J. & Andrzej, G. (2011). *Cloud computing: Principles and paradigms*. Wiley.

Dutta, P. & Dutta, P. (2019). Comparative study of cloud services offered by amazon, microsoft and google. *International Journal of Trend in Scientific Research and Development*, *Volume-3*(Issue-3), 981–985. https://doi.org/10.31142/ijtsrd23170

Egger, S., Reichl, P., Hossfeld, T. & Schatz, R. (2012). “time is bandwidth”? narrowing the gap between subjective time perception and quality of experience. *2012 IEEE International Conference on Communications (ICC)*, 1325–1330. https://doi.org/10.1109/ICC.2012.6363769

Grbich, C. (2013). *Qualitative data analysis: An introduction*. SAGE Publications Ltd.

Kaufmann, A. (2021). *Domain modeling using qualitative data analysis* (dissertation). Friedrich-Alexander-Universität Erlangen-Nürnberg. https://opus4.kobv.de/opus4-fau/files/16736/AndreasKaufmannDissertation.pdf

Kaufmann, A. & Riehle, D. (2015). *Improving traceability of requirements through qualitative data analysis*. Gesellschaft für Informatik eV.

Kavis, M. J. (2014). *Architecting the cloud: Design decisions for cloud computing service models*. John Wiley; Sons.

Knuth, D. E. (1971). An empirical study of fortran programs. *Software: Practice and Experience*, *1*.

Kuan, H.-H., Bock, G.-W. & Vathanophas, V. (2005). Comparing the effects of usability on customer conversion and retention at e-commerce websites.

Proceedings of the 38th Annual Hawaii International Conference on System Sciences. https://doi.org/10.1109/hicss.2005.155

Liu, C. & Arnett, K. P. (2000). Exploring the factors associated with web site success in the context of electronic commerce. *Information and Management*, *38*(1), 23–33. https://doi.org/https://doi.org/10.1016/S0378-7206(00)00049-5

MacQueen, K. M., McLellan, E., Kay, K. & Milstein, B. (1998). Codebook development for team-based qualitative analysis. *CAM Journal*, *10*(2), 31–36. https://doi.org/10.1177/1525822x980100020301

Majchrzak, T. A., Biørn-Hansen, A. & Grønli, T.-M. (2018). Progressive web apps: The definite approach to cross-platform development? *Proceedings of the 51st Hawaii International Conference on System Sciences*. https://doi.org/10.24251/hicss.2018.718

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

Mell, P. M. & Grance, T. (2011). The nist definition of cloud computing. https://doi.org/10.6028/nist.sp.800-145

Miles, M. B. & Huberman, A. M. (1994). *Qualitative data analysis: An expanded sourcebook*. Sage.

Novak, T., Hoffman, D. & Yung, Y.-F. (2000). Measuring the customer experience in online environments: A structural modeling approach. *Marketing Science*, *19*, 22–42. https://doi.org/10.1287/mksc.19.1.22.15184

Palmer, J. W. (2002). Web site usability, design, and performance metrics. *Information Systems Research*, *13*(2), 151–167. https://doi.org/10.1287/isre.13.2.151.88

Pareto, V. (1897). Cours d'économie politique. *The ANNALS of the American Academy of Political and Social Science*, *9*(3), 128–131.

Rappaport, J. (1987). Terms of empowerment/exemplars of prevention: Toward a theory for community psychology. *American Journal of Community Psychology*, *15*(2), 121–148. https://doi.org/10.1007/bf00919275

Saldaña, J. (2013). *The coding manual for qualitative researchers*. SAGE.

Standish Group. (2010). Modernization: Clearing a pathway to success [https://www.standishgroup.com/sample_research_files/Modernization.pdf last visited 28th of September, 2021].

Steiner, T. (2018). What is in a web view? an analysis of progressive web app features when the means of web access is not a web browser.

Wickham, M. & Woods, M. (2005). Reflecting on the strategic use of caqdas to manage and report on the qualitative research process. *The Qualitative Report*, *10*. https://doi.org/10.46743/2160-3715/2005.1827

Zhang, Q. & Yang, Y. (2009). A study of positive effects on user experience in navigation. *2009 IEEE 10th International Conference on Computer-Aided Industrial Design Conceptual Design*, 444–447. https://doi.org/10.1109/CAIDCD.2009.5375365