Universiteit Leiden
The Netherlands

Computer Science

# Proxy Caching in Accenture's Employee Evolution Tree Project

A case study of it's effect on performance

Lucas Allison

dd/mm/yyyy

Thesis Supervisors:
**Dr. A.W. Laarman**
**Dr. A. Uta**

Internship Supervisor:
**Guillermo Martinez, Senior Manager at Accenture**

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

**Abstract**

For this thesis we explored if proxy caching could increase the performance of Accentures Employee Evolution Tree project. We did this by deploying a proxy cache and benchmarking it with a realistic workload. We studied the influence of several eviction policies on the performance of the cache, namely LRU, LFU, GDS, GDS and TinyLFU. We also distributed the cache to see of this could better handle an increasing amount of concurrent requests. From the experiments we found that the proxy cache did not increase performance. The eviction policies had little effect the functioning of the cache and distributing the cache increased the average response latency.

# Contents

# 1 Introduction

## 1.1 Context

This thesis has been written for Leiden University while following an internship project at Accenture. This is a global consultancy company with large departments specialized in technology. Every employee is assigned a level: this starts at 13 and can decrease to 1 as they progress within in the company by way of promotions. To gain promotions people are expected to deliver adequate work but also to keep evolving their skill set. To do so employees can follow online courses, conferences, presentations from colleagues and many other resources. However, it is unclear what skills should be obtained at which level. Moreover the resources are scattered and can be difficult to find. The *Employee Evolution Tree* project strives to resolve these problems by offering a simple and easy to navigate dashboard where all learning resources can be found and an employees progression can be tracked. The thesis project has been designed around this internship project and seeks to add value to it by increasing it's performance.

## 1.2 Microservices and their use case

The backend of the Employee Evolution Tree is written in Go and separated into *microservices*. In a microservice architecture an application is split into different services which are deployed in an isolated manner, often running on separate severs or their own operating system process. Ideally, these services are small and focused on one specific functionality [New15]. Consumers (e.g. the frontend or other services) access this functionality by communicating with an interface provided by the service. A common way is to expose `REST` over `HTTP`, which is the case for the Employee Evolution Tree services. This is in sharp contrast with the classic monolithic structure, where functionality is often separated through packages, modules or libraries. Which approach is used has many consequences for the development process. For the microservice architecture Chris Richardson defines the following reasons as some of the key benefits in his book "Microservices Patterns" [Ric19]:

- It enables the continuous delivery and deployment of large, complex applications.
- Services are small and easily maintained.
- Services are independently deployable.
- Services are independently scalable.
- The microservice architecture enables teams to be autonomous.
- It allows easy experimenting and adoption of new technologies.
- It has better fault isolation

This is particularly useful when writing large scale software with many contributing developers. Therefore it has become a popular architecture for many companies.

## 1.3 Problem statement

Microservices do not come without their drawbacks. The development process is more complex and debugging the application can be significantly more difficult. Microservices do not only effect developers. As opposed to a monolithic architecture there are problems specific to microservice architectures which frequently occur and affect the user perceived latency:

- Communication between services is complex. This requires remote calls which can increase latency, especially when these are synchronous calls which often appear for `GET` requests. In a monolithic architecture local in process calls are used which do not incur any network latency.
- Managing consistency between services can effect performance. Keeping other services informed and processing messages from them causes overhead, particularly when this is not managed by asynchronous message queues.
- Data might be more difficult to access. In the Employee Evolution Tree project each microservice has their own database only accessible through the interface provided by the service. When a different service needs access to a different databases it has to communicate through the separate services. It cannot directly query a single database.

If not dealt with properly these aspects can lead to long response times.

## 1.4 Research questions

Designing a backend well can be difficult and performance issues (as mentioned in Section 1.3) can always occur, especially for inexperienced microservice developers. For this research project we are seeking to decrease the overhead of microservices and increase the performance of the backend. We will try to do this by deploying a *proxy cache*. In further sections we will describe in detail what this entails and in Section 3 we will elaborate on this choice of a proxy cache as opposed to other forms of caching. The main question we seek to answer is: Can proxy caching increase the performance of the Employee Evolution Tree application? We will also consider the following sub questions to help answer our main question:

- Which eviction algorithms yield the largest increase in performance?
- How do we prevent the cache from becoming a bottleneck as the workload increases?

There is one last question which is particularly useful for Accenture, since it is a consultancy company: In what scenarios can proxy caching bring value to a client?

## 1.5 Contributions

In this thesis we will explore the effect of a proxy cache on the Employee Evolution Tree project. We will use existing literature to design the cache and through experiments we will see how they effect the performance. Caching has been extensively researched, so in this paper we will try to combine this existing knowledge into a useful extension of the Employee Evolution Tree. We will discuss which aspects are necessary to successfully deploy a proxy cache within an existing application.

## 1.6 Overview

First we will introduce all necessary terms and background information in Section 2 needed for the rest of this thesis. In Section 3 we will discuss the cache implementation that has been made for this project. We will cover the experiments to measure its performance in Section 4. Some related papers and research will be discussed in Section 5. Last, the conclusions and further research can be found in Section 6

# 2 Background

## 2.1 Technology stack of the Employee Evolution Tree Project

The project consists of two parts: the *frontend* and *backend*. The frontend is a React app hosted on an Azure App Service. The backend is a collection of microservices written in Go hosted on an Azure Kubernetes Service. Each service has their own database: depending on the requirements of the service this is either a *MySQL* or a *MongoDB* database.

## 2.2 General terminology

### 2.2.1 The HTTP protocol

There are many protocols which computers can use to interact over the internet. In the case of the Employee Evolution Tree this is by means of the `HTTP` protocol. A *request* is sent to the server, to which it replies with a *response* over a lossless `TCP` connection. The three most important aspects of a `HTTP` request are the method, headers and body. The method denotes what operation the sender wants the server the execute, the headers contain key-value paired meta data and the body contains any data that the sender wishes to supply to the server. Upon receiving a request the server can "choose" to act upon it and send back a `HTTP` response. This is a way to inform the sender of the request how the server has handled it. Besides also having headers and a body, a response returns a status code: this indicates whether the request has successfully been completed.

A `HTTP` request potentially alters the state of the server. This is the case when the request method is `POST`, `PUT` or `DELETE`[1]. We will refer to them as *invalidating requests*, since they possibly invalidate cached data (see Section 2.3.3). A `GET` request only asks to retrieve data from the server and does not alter it.

### 2.2.2 REST APIs

Uniform Resource Identifiers, or URIs, are a way to address resources on the internet. A URI has the following format:

```
URI = scheme :// host / path
```

[1]The `HTTP` protocol has many more request methods, but these are not supported by the Employee Evolution Tree backend and therefore not considered in this thesis.

The scheme denotes the protocol that is used to access the resource. The host is either an IP address or a domain name which resolves to this address. The path denotes a resource of the backend relative to the base of the server (this can be compared to the root folder '/' in Linux systems). If a servers resource is accessible through a URI with a given scheme, the host that identifies the server and a path, it is also referred to as an *endpoint*. To interact with the Employee Evolution Tree `HTTP` requests can be made to URIs which identify endpoints of the backend.

The accessible URIs of the backend follow the `REST` design pattern. This has useful benefits when interacting with the backend, but for caching it is important to know that `REST` APIs impose a hierarchy on the paths. This means that a path conveys a resource model, with each forward slash separated path segment corresponding to a unique resource within the model's hierarchy [Mas12]. Consider the following URI:

```
https://mydomain.com/paths/pid/skills
```

This path implies that there are paths (accessed by `/paths`), which can be accessed individually by their path ID (accessed by `/path/pid`) and finally the skills belonging to this specific path can be accessed by the example.

### 2.2.3 Scaling applications

We can differentiate between two ways of scaling applications: vertical and horizontal scaling. In *vertical scaling* the computing power of the host machine that runs the application is increased (e.g. more RAM, CPU cores, etc.). In *horizontal scaling* the amount of machines that the application runs on is increased. In this case the application will be run distributively over these machines [Clo21]. It is important to note that vertical scaling is powerful but the amount of hardware that can be added to a machine is limited. Horizontal scaling can relieve this problem (you can add as many machines to the distributed system as you have to your disposal) but is only useful when the application is parallelizable. Note that for applications serving `HTTP` requests this is the case. Requests are independent of each other be handled separately. This observation will be useful when considering the scalability of the proxy cache.

## 2.3 Caching

Caching in computer science is the practice of storing data in temporary storage that is often in closer proximity to the processing/querying party to decrease latency and increase throughput. Often it is referred to in the context of hardware performance, but it can be used in many more places. For this thesis we will specifically look at caching within the context of web applications where it can also be desirable.

### 2.3.1 Where can caching be applied?

When implementing caching for web applications there are three main places it can be applied [New15]:

1. **Client-Side/browser caching**: the cache runs on a users device. This is mostly managed by a users browser but can also be controlled by the frontend.

2. **Proxy caching**: a proxy is placed in between two parties communicating over the internet. This could be in between the client and the frontend or the frontend and the backend. The proxy intercepts requests and potentially serves cached data. This is useful when data has to be served in places geographically far from the server hosting the application. It is easy to build upon existing systems and is deployed separately from the actual web application.

3. **Serverside caching**: here the servers running the application handle the caching. Specifically for a microservice architecture this could mean that an individual service chooses to implement and serve data from a cache. The implementation can be easier to reason about than proxy caches and the bottlenecks (services with high latency) can be specifically targeted.

In the context of the Employee Evolution Tree project these are depicted in figure 1. In the first case the cache is called a *private cache*. The latter two cases are referred to as *shared caches* [con22].

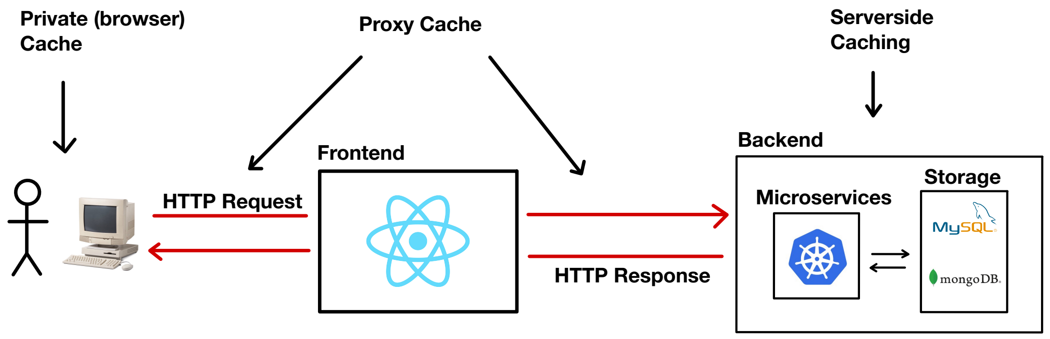

Figure 1: Possible cache locations for the Employee Evolution Tree project

### 2.3.2 What to cache?

The RFC document 7234 [Fie14] describes rules for `HTTP` caching. `HTTP` responses can be saved as key value pares in the desired cache. It states that caching responses is often limited to `GET` requests that return a status code of 200 (status OK), but can also store them when the status code is 301 (Moved Permanently), 404 (Not Found) or 206 (Partial Content). Caching can be extended to `DELETE` and `PUT` requests, since they are *idempotent* operations [Mas12]. But these are often omitted from caching, since this is not always implemented properly.

A cache should be aware of certain headers supplied in the request/response, since these can be added as directives for it. The RFC defines the following rules, which must always be followed [Fie14]:

- The request method is understood by the cache and defined as being cacheable, the response status code is understood by the cache, and the `no-store` cache directive does not appear in request or response header fields.

- If the cache is shared, the `private` response directive does not appear in the response and the Authorization header field does not appear in the request, unless the response explicitly allows it (by setting the `public` and `must-revalidate` directives).

If the cache adheres to all the impositions a response can be cached.

### 2.3.3 Invalidation

We say a cache entry is *invalid* or *stale* when the entry in the source and cache do not match for the given key. Specifically for web caches this means that if a `GET` request is made to a certain endpoint the corresponding cached response cannot be served, since a request directly to the backend would yield a different response. If an entry can safely be served by the cache it is called *fresh*. Cache *invalidation* usually refers to removing or replacing the entry, but in proxy caches the entries are *blacklisted* [Unk21]:

"*A reference to the cached content is added to a blacklist (or ban list). Client requests are then checked against this blacklist, and if a request matches, new content is fetched from the application, returned to the client, and added to the cache.*"

Cached entries can be added to the ban list once an invalidating request modifies the entry on the origin server.

How invalidation is manged depends very much on the context. The most common case for proxy caches is that the server controls the cache by supplying it with the `expires` or `max-life` header, indicating how long the cache can consider the entry to be fresh. After the entry has surpassed it's "lifetime" *etags* (see Section 3.2) can be used to revalidate the entry [con22]. Note that in this case the invalidation is managed by the server, not the cache. This is because in web caching not all entries can be treated the same by the cache. For example some entries should always be revalidated (e.g. to check credentials) or never cached while others (such as static content) can be served without revalidation for very long periods of time. There are many other aspects to consider, so the caching possibilities for an endpoint depend very much on the implementation and requires knowledge about that endpoint to determine what they are. Therefore it is useful to "control the cache" from the backend so that all the caches that the response passes through (e.g. proxy cache, client side cache, etc...) handle it in the same way. It is still possible to maintain the blacklist within in the cache, as long as the server has already informed it about the cachebility of its entries. However there are two main reasons to keep the blacklist on the server side. First, once different proxy caches are deployed it is easier to manage from the backend. Say we are running two proxy caches in geographically different locations, let's refer to them as cache A and B. If invalidation is managed separately and an invalidating request passes through A than either B would be unaware of this and could potentially serve stale content or A would have to keep B informed about the requests it receives (and vice versa). Enforcing consistency between these caches can be difficult, so managing it from a central point can be convenient. Second, in many cases the proxy cache has to confirm with the backend if a user has the proper credentials for the requested entry (in the case of the Employee Evolution Tree this holds for all endpoints). For these requests the cache has to communicate with the backend before serving a cached response, so confirming with the blacklist if an entry is fresh at this moment does not incur any overhead.

The cache control can be implemented in a microservice itself or even for specific endpoints. This can be useful if you want the cache to behave in a certain way for the responses of this particular service/endpoint. It can also be managed by more generic middleware which can be included in the microservices where caching is desired. The middleware can also be executed for every request, managing cache control for all endpoints.

### 2.3.4 Eviction

The *eviction policy* of a cache determines which entry to remove once it becomes full. A good eviction policy is a very important aspect of a well functioning cache: we do not want to remove entries that are frequently accessed. In contrast to CPU caches, which deal with uniform sized entries, proxy caches have to be able to deal with vastly different sized entries. In Section 3.3.2 we will discuss policies implemented for this thesis.

### 2.3.5 Performance measures

The Hit Ratio (HR) and Byte Hit Ratio (BHR) are common ways to measure the performance of a cache. The Hit Ratio denotes the percentage of requests to which the cache can serve a saved entry. The Byte Hit Ratio is similar, but denotes the amount of bytes served by the cache as a percentage of the total amount of bytes served to the user. Let $N$ be the total amount of requests. Let $\delta_i = 1$ if request $i \in N$ is served by the cache and $\delta_i = 0$ otherwise. Lastly, let $b_i$ denote the amount of bytes served for the *ith* request. Mathematically these measurements can be written as follows [WI11]:

$$HR = \frac{\sum \delta_i}{N} \qquad BHR = \frac{\sum b_i \delta_i}{b_i N}$$

# 3 Implementation

As mentioned in Section 2.3.1 there are three possible locations to implement the cache. To adhere to Accenture's request to keep the thesis and internship project as separated as possible we will be implementing a proxy cache. This can be deployed independently without having to write code supporting it in the internship project. For client side and server side caching this is not the case. For the former we would have to control the cache from the frontend, for the latter we would have to implement the cache in the microservices themselves. In both cases the thesis and internship code will become very coupled.

To successfully cache responses two parts have to implemented: the proxy cache and middleware controlling it. They will be discussed separately.

## 3.1 Proxy cache implementation

The cache abstractly consists of three layers:

1. Request handling and tagging
2. Concurrent access and eviction management
3. Cache implementation

This is the case because the proxy cache is an extension of Google's Groupcache. The first layer is built on top of the existing cache in order to proxy and tag requests. The third layer has been extended to support cache implementations with different eviction algorithms; Groupcache only supports LRU eviction.

The first layer receives requests and determines what to do with them. If an invalidating request is received it is immediately proxied to the backend (and received by the cache control). Responses to these requests are never cached. In the case of a `GET` request there is more work to be done. If the cache holds a response to this request it tags the request (with the etag saved for the cached response, see 3.2) and sends it to the backend. The backend either responds with a status code 304 or with any other status code. In the former case this means the cached entry is valid and a request to the second layer can be made to retrieve it. In the latter case the cached entry is invalid (or not yet stored) and a request to the second layer can be made to store the response.

In order to store a response the first layer calculates and stores it's etag. Afterwards it deconstructs and encodes the response as a byte array which is sent as a key value pair to the second layer. The key is set to be the path of the URI that the request is sent to. If the cache wants to serve a fresh entry, the first layer retrieves the stored byte array which it decodes and uses to reconstruct a response that can be served.

The second layer manages requests to store and retrieve data from the cache (the third layer). This is done by using a mutex lock. The lock can be held by an arbitrary number of readers or a single writer, which ensures data consistency when concurrent writes appear. When the first layer wants to retrieve an entry, the second layer obtains the lock as a reader and returns whether the cache holds data for the supplied key and if so it also returns the data. In case the first layer wants to store data the second layer obtains the lock as a writer and stores the key-value pair in the cache. If the entry will cause the number of bytes stored by the cache to exceed the maximum number of available bytes (set by the programmer), a request is made to evict entries until there is enough space to store the new key-value pair.

The third layer is the implementation of the actual cache. It stores key-value pairs which can be removed based on an eviction policy. This is discussed further in Section 3.3.2.

## 3.2 Cache control middleware

As mentioned in Section 2.3.3 the cache is controlled by the backend. In this project the cache control has been implemented as middleware running on the backend cluster. It is essential in making the cache useful. Invalidation is completely managed by the middleware and it informs the cache about the freshness of its entries. It intercepts requests before they go on to the designated microservice. Even though it is a part of the backend the middleware can be though of as "sitting in between" the proxy cache and the backend. Therefore they will be discussed as if they are separate entities.

Fundamentally the middleware stores *etags* (or simply *tags*) for each endpoint accessed by `GET` requests. An etag is a hash of a `HTTP` response. When an endpoint is accessed by a `GET` request for the first time the etag of the response is saved by both the middleware and the cache. The cache can now confirm with the middleware if it's entry is fresh by sending the etag to it. If the etag's match the status code 304 is sent back to the cache. If the etag is empty or not present this means that the cached entry might not be fresh anymore. The middleware forwards the request to the backend and determines and saves the etag for the given response. Before sending the response to the cache it compares the etag supplied by the cache with the newly calculated etag once again: it is possible that the cache still holds a valid entry. If they match the status code 304 is sent, otherwise the response from the backend. Notice that in the former case the latency is not necessarily reduced, but the amount of bytes sent over the network is decreased.

Invalidation is a bit more difficult. Blacklist too many endpoints and the cache will become obsolete. Targeting very specific endpoints to blacklist is the most effective but can be computationally intensive or difficult to implement. In this middleware the following heuristic is used: if an invalidating request is made to a microservice all endpoints of this microservice are blacklisted. This heuristic is useful since the microservices should have tight coupling and loose cohesion [New15]. Therefor it is likely that an invalidating request to a microservice has an effect on many of its endpoints, but not on those of other services. In the backend a request is routed to a microservice based on first prefix of the path of a URI. For example the path

```
/users/5/skills
```

is routed to the user service, since `/users` indicates that all paths containing this prefix are related to users due to the hierarchy imposed by `REST` API's 2.2.2. We use this to our advantage and let the first prefix of a path create a natural categorization for saving tags. Each tag is saved in a *tag pool*. A tag pool contains all tags for endpoints accessed with the same prefix (in the backend they would be routed to the same microservice).

When an invalidating request comes in all tags within the corresponding tag pool are blacklisted. Blacklisting in this case simply refers to setting the etag to be empty: empty tags are never used for comparison, so the tag pool controller will always forward requests for these entries to the backend. Notice that we now probably over invalidate endpoints. It can be the case that an endpoint is not effected by the invalidating request. So an empty etag in the middleware does not imply that a cached entry is stale, it just cannot guarantee that it is fresh.

Note that it is possible that a request to a specific microservice can also invalidate endpoints of different microservices (this occurs when the services are coupled). In this case the programmer has to explicitly set instructions for to the tag pool controller. Once the request comes in all tags in the coupled tag pools are concurrently blacklisted.

Finally, there is one more important thing to note: storage is never infinite, so if the tag pool becomes full (a maximum amount of tags/bytes are stored in the pool) tags are removed based on the LRU policy.

## 3.3 Expected effect of the implementation on performance

### 3.3.1 How can the proxy cache increase performance?

When the proxy cache is deployed it has two advantages that can decrease latency:

1. Each time a `GET` request is made for which the cache holds a fresh entry this can directly be served as a response. There is no need for the backend to retrieve data from a database or recompute the same response.

2. The proxy cache can be deployed geographically closer to originating requests, incurring less networking latency.

Managing the cache also incurs overhead, especially when an entry is stale or not present. However the expectation is that this effect is small compared to serving requests directly from the backend.

### 3.3.2 Managing eviction

The eviction policy that is used can have a big effect on the performance of the cache. On initialization a specific eviction policy is chosen to manage the storage. The following policies are supported:

- *LRU*: evict the least recently used entry.
- *LFU*: evict the least frequently used entry.
- *GDS*: function based eviction.
- *GDSF*: an extension of the GDS algorithm also taking frequency into account.
- *TinyLFU*: manages cache admission on top of eviction.

The advantages of LRU and LFU are that they are fast and easy to implement. They also retain entries based on intuitively valuable aspects: recently and frequently accessed items are more likely to be accessed again. The GDS algorithm is designed to consider specific properties of internet resources. It is a function based eviction policy and takes the recency, fetching cost and size of entry into account. The GDSF algorithm extends this by also considering the frequency of the entry. The TinyLFU algorithm seeks to combine the desirable aspects of considering recency and frequency by evicting based on the LRU policy and admitting based on LFU.

The downside of the LRU and LFU policies is that both of them are limited since they only take a single aspect of the access pattern into account. The expectation is that using a more sophisticated policy, such as the latter three, will yield better performance results. They are discussed in more detail in Section 5

### 3.3.3 Preventing a bottleneck

As the application scales and the amount of requests increase it is important that the proxy cache and the middleware can keep up with the demand. If they cannot this would make the cache irrelevant.

We can scale the cache vertically, but as mentioned in Section 2.2.3 there is a limit to this approach. We can use the fact that `HTTP` request can be handled independently to distribute the cache over multiple machines which can serve them separately. This is achieved by defining a *cache pool* on initialization. This is a constant sized array containing the HOST addresses of each cache instance. If our array has 50 entries and we have 3 caches the HOST address of each cache will appear roughly $\frac{50}{3}$ times. They are inserted in a cyclical manner: in our example the HOST address of the first cache will appear at the index 0,3,6, etc. When a request is sent to the cache the path of the URI is hashed and used to determine an index within the array boundaries. Retrieving the entry from the cache pool with this index yields one of our cache addresses to which the request can be forwarded. This is called *peer picking*.

For the middleware there are two main concerns to keep in mind:

1. Checking if an entry is fresh and blacklisting endpoints should be fast operations: if this is not the case the backend could have directly served the response.
2. It should scale well horizontally: the same argument holds here as it does for the cache. If the workload increases the middleware could become a bottleneck, so having the possibility to scale it easily is important.

Figure 2 shows the components of the middleware. With the help of this illustration we can easily explain why the middleware satisfies our requirements. Checking if an entry is fresh can be done very quickly. A tag can be retrieved in constant time $\mathcal{O}(1)$. This is because the tag pool controller accesses tag pools through a map, using the prefix of the URI that the request is made to as a key. Within the tag pool a tag is accesses through another map with the entire path as the key. Blacklisting can be done in polynomial time $\mathcal{O}(n)$, where $n$ denotes the amount of tags of the largest tag pool. This is because for an invalidating request all tags within a tag pool are set to be empty. The figure also shows that it can easily be scaled horizontally: tag pools operate independently of each other and can be managed on different machines. The only difference now being that the tag pool controller accesses a pool over the remotely.

## 3.4 Example trace

To clarify how requests are served and endpoints are invalidated we will show an example of a request trace. Consider that the cache holds data for the endpoint `/path/3` and when the response for this endpoint was cached the etag was calculated and set to "3a4dk". Now consider three consecutive requests depicted in figures 3, 4 and 5 respectively.

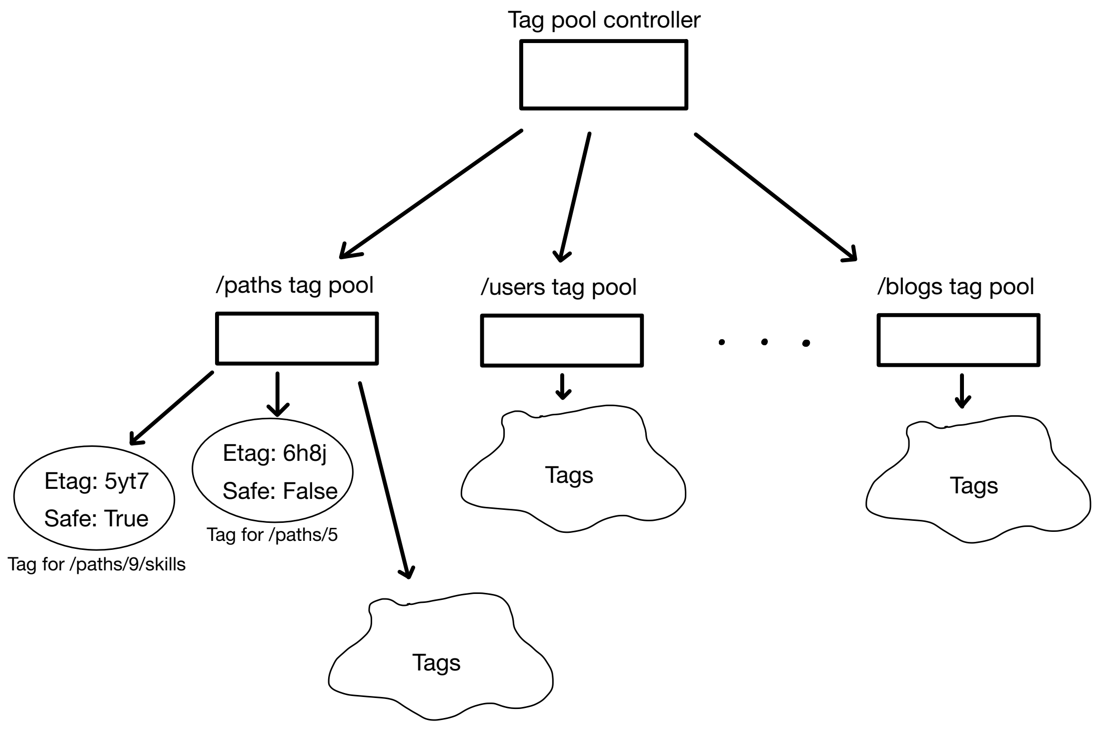

Figure 2: Cache control middleware

The first request is a `GET` request to the cached endpoint `/path/3`. The request is first tagged by the cache and sent to the backend. The middleware validates this tag and sees that it is safe and equals the tag for this endpoint, which means the cached data is valid. There is no need to retrieve the data from the backend so the middleware responds with the status code 304. The cache is informed that it's data is fresh and responds with the cached entry.

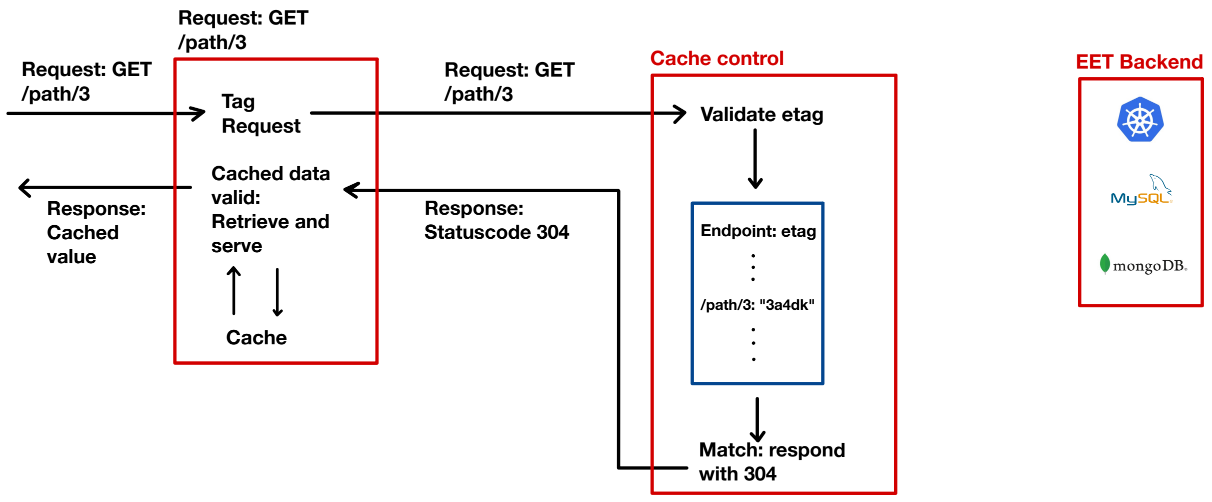

Figure 3: Cached data is fresh

The second request is a `POST` request. This potentially invalidates all data for endpoints with the prefix `/path`, so all tags within the tag pool `/path` are set to empty by the middleware.

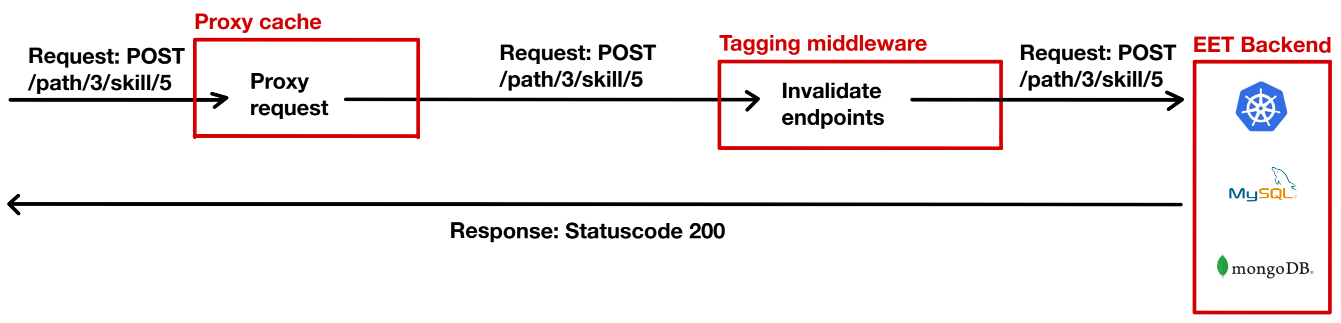

Figure 4: POST request invalidating endpoints

The third request is another `GET` request to the endpoint `/path/3`. As with the first request it is tagged but the middleware sees that the tag is empty. The request is sent to the backend and its etag is calculated. The middleware compares it too the supplied etag and sees that they do not match: the `POST` request changed the data for this endpoint. The middleware cannot respond with the status code 304 so it sends the respond from the backend to the cache, which updates the entry.

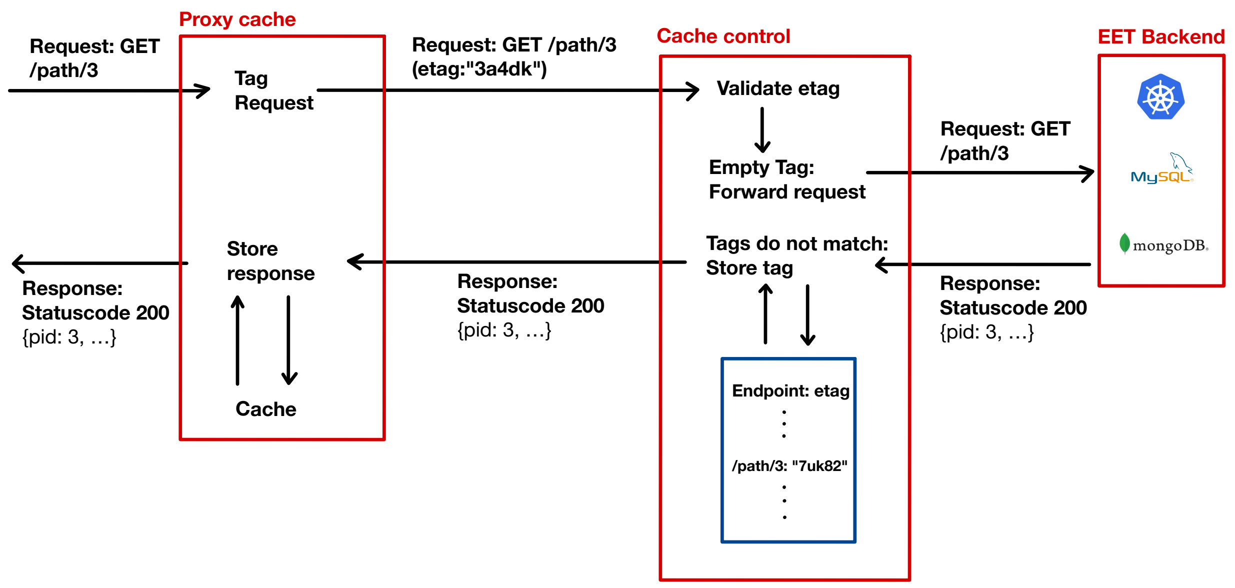
Proxy cache
Request: GET /path/3
Tag Request
Request: GET /path/3 (etag:"3a4dk")
Cache control
Validate etag
Empty Tag: Forward request
Request: GET /path/3
EET Backend
Store response
Response: Statuscode 200 {pid: 3, ...}
Cache
Response: Statuscode 200 {pid: 3, ...}
Tags do not match: Store tag
Response: Statuscode 200 {pid: 3, ...}
Endpoint: etag
/path/3: "7uk82"

Figure 5: Cached data is stale

## 4 Experiments

### 4.1 Validation

Before running the experiments it is important to validate that the cache behaves as expected. A big pitfall of caching is serving stale data. This undesirable for two reasons:

1. Stale data might never be evicted, potentially being served forever.

2. If a lot of stale data is served the benchmarks will not be realistic, since this is computationally less expensive than fetching fresh data for an endpoint.

It is difficult to guarantee that a cache will always serve fresh data, but in order to combat this problem a *validator* has been written, which can optionally be enabled. For every entry that the cache serves the validator confirms with the backend whether this data is actually fresh. If this is not the case it will write to the logs for which entry this appears. Requests can be generated and sent to the proxy cache, as described in section 4.2. The logs are now easily examined to determine the correctness of the cache.

## 4.2 Experiment setup

In order to yield the most realistic results we will use the existing server infrastructure to deploy the cache. We have two servers to our availability: the Kubernetes cluster from the backend and an Azure App Service from the frontend. It is a possibility to deploy the proxy cache on the Kubernetes cluster, however if it would be used in a production environment it would be deployed on a separate server, preferably as close as possible to the originating requests. To simulate this we will deploy the proxy cache on the Azure App Service. Both servers are located in West-Germany, so the proxy will not have a geographical advantage, however we do not have a server at our disposal for which this is the case. The proxy cache can be deployed within a *slot*. Here a web app can run in an isolated manner accessible through a public endpoint. Each slot has 3.5 GB of RAM available. We will use 2.5 GB of this for the storage of cached entries. The rest can be used by the to Go program running the cache. In Section 4.3.4 we will look at a smaller cache size, but in the rest of the experiments we will not vary this value. In a production environment the cache size would also be set to be as large as possible, so there is no need to restrict it with less storage.

To benchmark the deployed cache a couple of steps are execute. First a custom Go program deletes all data from the database and floods it with random entries. The entries that are generated are based on what would be representative of the production database. For example the app will have about 100 users, so this is the amount of users that will be generated as mock data. After having generated the mock data the program generates *trace files*. These are simply files denoting which endpoints are accessible. For example if the program would generate a path with ID 3 it would add `/path/3`, `/path/3/skill`, `/path/3/info` and many others to the trace files. Each supported `HTTP` method has a separate trace file. This is necessary because from these files a program called YCSB generates a workload and sends requests to the cache. The program uses a Zipfian distribution of the endpoints to generate an realistic workload. The amount of times a certain request method occurs can be set as a percentage of the total amount of requests. For each experiment the total workload consists of 80% `GET` requests, the other 20% are `POST`, `PUT` or `DELETE` requests. This division is chosen since the application will be mostly used to retrieve information. Altering data will occurs much less frequently. For each request YCSB can only set the method. However, some requests require a body or specific headers. This is mostly the case for `POST` and `PUT` requests. To solve this, the requests pass through a Go program which will make sure to turn them into valid requests by adding a body and/or headers where necessary.

Each request in the experiments is generated and sent from a Dell XPS 15 (Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz) located in Leiden. They are sent to the proxy cache on the Azure App Service in West-Germany. The cache then communicates with the Kubernetes cluster, also located in West-Germany. In case the performance of the backend is measured requests are sent directly to the cluster, skipping the proxy cache.

## 4.3 Experiment Results

### 4.3.1 Increasing the request count

For the first experiment we will look at performance of the proxy cache as the request count increases. A request is sent once the previous has received a response. In Figure 6 we can see the average response latency to a request as the total request count increases. Here we can see that the latency does not drop as the amount of requests increase. This is interesting to note since we can see in figures 7 and 8 that the amount of responses served from the cache increases overtime. This implies that serving an entry from the cache is not faster than serving it directly from the backend. In fact, from Figure 6 is it is clear that the overhead that serving entries from the cache incurs even slows down response times.

When looking at figures 7 and 8 we can see that the Hit Ratio and Byte Hit Ratio are nearly identical for all eviction policies. At first this seems strange, as it would imply that they all evict entries similarly. On closer inspection this is not the case, but rather that entries are not being evicted at all. We ran the experiments with a cache size of 2.5 GB. If we look at the responses the backend serves these are all very small, many under 1 KB. Some endpoints serve “larger” data, for instance when requesting all paths, but almost all are under 20 KB. If we assume that each response we cache is 20 KB (which is quite large overestimation), we can cache response for over $10^5$ unique endpoints. This is well over the amount supported by the Employee Evolution Tree. Since the workload is generated by the YCSB program and overtime will roughly send requests with a similar access pattern, it causes the cache to contain roughly the same amount and variety of entries for each experiment. This explains the similarity between the eviction policies seen in the figures.

We can also see that the TinyLFU algorithm sometimes performs much worse than the rest of the eviction policies. This is probably due to the overhead that maintaining the counting Bloom filter incurs. For each `GET` request the filter has to be updated. This is done by acquiring a mutex lock. Waiting for other requests to finish updating the filter can slow down serving requests. This likely caused the increase in response latency.

### 4.3.2 Increasing amount of concurrent requests

As applications become more popular the amount of concurrent requests will increase. In Figure 9 we can see the effect of sending requests on an increasing amount of threads to the cache and server. For this experiment the request count was not varied and set to 15000. We can see that the latency increases similarly for each eviction algorithm. This is likely due to the fact that entries are not being evicted, as mentioned in Section 4.3.1. We can also see that the response latency of the server increases as the amount of concurrent request increase. We can also see that the response latency of the backend increases similarly, but for each amount of threads performs better than the cache.

### 4.3.3 Distributing the cache

Since the proxy cache can be distributed we will look at the effect of this as the amount of concurrent requests increase. The experiment setup was the similar as in Section 4.3.2, however this time we have varied with the amount of cache instances and set the eviction policy to be LRU for all caches. In Figure 10 we can see that distributing the cache actually decreases performance. This is caused by the overhead of forwarding a request. Since the single proxy cache can store all the responses, having a different cache serve the response only increases the latency. It does seem the case that if the cache is distributed (and the overhead of forwarding requests appears) using three distributed caches over time will serve responses faster than two distributed caches.

### 4.3.4 Decreasing the cache size

For the last experiment we will try to find the smallest possible cache size for which it offers 90% of the performance. This way we can make a recommendation for a cheaper cache. The total request count will be set to 15000 and sent using 4 threads. We will measure it's performance based on the hit ratio. We will use hit ratio over average response latency since this will give an indication for which amount of memory the cache can serve a similar amount of requests from it. The latency will not be a useful metric since we've seen in the previous experiments that serving requests from the backend is faster than serving them from the cache. For such a workload the cache used in the previous experiments had a hit ratio of roughly 78% for all eviction policies. In Figure 11 we can see that for all eviction policies a cache size of $7 \times 10^6$ bytes (or almost 7 MB) is sufficient to obtain 90% of the performance (in terms of hit ratio) of the full proxy cache. If the cache would be smaller than this, the the LFU policy performs considerably worse and the TinyLFU policy slight better than the other eviction algorithms.

## 5 Related Work

Since invalidation is very dependent on the context, most research is related to eviction policies. Managing eviction for proxy caches is different than for example hardware caches. It deals with vastly different fetching latencies and entry sizes. Coa and Irani developed the Greedy Dual Size (GDS) algorithm specifically for this case [CI97]. Each entry is given a value based on a function. A priority queue is maintained where entries with the smallest values are removed when they are "popped" from the queue (i.e. evicted from the cache). Inserting entries can be done in $\mathcal{O}(\log n)$ time. By keeping track of all the existing entries in a map cache hit's can be handled in constant time, $\mathcal{O}(1)$. For an entry $p$ it is assigned a value based on the following function:

$$H(p) = L + \frac{C(p)}{S(p)}$$

Here $L$ denotes an aging factor: it is updated as an entry is accessed which causes "old" entries to be moved to the front of the queue more quickly. $S(p)$ denotes the size of the entry and $C(p)$ the cost. How the latter value is determined depends on the goal of the cache. As mentioned in the paper:

"*Cost is set to 1 if the goal is to maximize hit ratio, it is set to the downloading latency if the goal is to minimize average latency, and it is set to the network cost if the goal is to minimize the total cost.*"

The GDS algorithm does not take the access frequency of the entries into account. However it can be useful to incorporate this. Cherkasova extended the algorithm by including the frequency while determining an entries value, know as the Greedy Dual Size Frequency (GDSF) algorithm [Che98]. The function now becomes as follows:

$$H(p) = L + F(p) \cdot \frac{C(p)}{S(p)}$$

Where $F(p)$ is initially set to 1 and incremented on every access.

Conventional eviction policies can be adequate, but are often limited. For example eviction based on LFU is intuitively desirable; we want to keep frequently accessed items in the cache. But often access patterns change rapidly overtime. A popular video on one day might not be viewed much the next. In this case it is desirable to take the recency of the access patterns into account. For this Einziger, Friedman and Manes developed the TinyLFU cache admission policy [GM17]. Instead of immediately evicting an entry the algorithm decides whether it is desirable to cache the replacement entry.

TinyLFU uses the access frequency of entries to determine whether it is worth replacing an entry. For this project the least recently used entry will be replaced. Keeping track of the full request history will be too memory intensive. Instead it uses a *counting Bloom filter* to estimate the access frequency of an entry. Normally a Bloom filter is used to predict whether an entry is part of a set. It relies on the fact that false positive matches are possible, but false negatives are not. In the case of TinyLFU this means that if the Bloom filter indicates that an item is accessed infrequently we can guarantee that it is. However if it indicates that an item is accessed frequently we only know that this *might* be the case. With this estimate we can now choose if we want to replace the least recently used item with the newly accessed entry. This way infrequently visited endpoints are not cached, and frequently visited ones replace the items that have not been visited recently.

## 6 Conclusions and Further Research

With the experiment results we can answer our research questions. It is clear that for a representative proxy cache the choice of eviction policy matters very little. This is due to the fact that entries do not have to be evicted, as mentioned in Section 4.3.1. The conventional eviction policies, LRU and LFU, perform slightly better than others. This is however not because the eviction management is better, but rather the "book keeping" of these algorithms is computationally less expensive. Distributing the cache does not help in combating an increase in concurrent requests and even slows down response times. Forwarding requests to another cache instance incurs unnecessary overhead. We can surely say that proxy caching does not increase the performance of the Employee Evolution Tree.

From this project is has become clear that for an application that serves small amounts of data and does not suffer any performance issues that proxy caching brings no value to it. However, in the following scenarios it might be of use to one of Accentures clients:

- Large amounts of data have to be fetched from a server geographically distant from the originating requests. A strength of a proxy cache is that it can serve content closer to the source of requests, decreasing request latency. For companies this can be valuable since a single server can be maintained but content can be served from a proxy closer to the clients.
- Generating a response is computationally expensive. If response latency of the server is high due to expensive operations that have to be performed in the backend a proxy cache can reduce response latency since the produced values do not have to be recomputed again.
- A large or difficult code base suffering from performance issues. Increasing the performance by optimizing code can be difficult in these cases. A big advantage of a proxy cache is that it can be deployed independently without having to have much knowledge of the application. This could be an easier way to solve the performance problems than modifying the code base.

From the last experiment, discussed in Section 4.3.4, we can conclude that we if we are deploying the cache this can be done on a much cheaper server. For example we could deploy it on Azure B-series virtual machine. Here we have 0.5 GB of memory to our availability, which is more than enough to run the cache with similar performance as the one deployed on the Azure App service. The B-series virtual machine costs only 0.0052 dollars per hour, which is roughly 3,7 dollars a month. This is considerably cheaper than the price of the Azure App Service, which costs 83.95 dollars per month. However, it is worth noting that scaling down the server for the proxy cache is only useful if the server is dedicated to it. Currently deploying the cache does not incur any extra costs, since the Azure App Service is needed by the frontend.

Further research can be done into the effects a proxy cache has on an application that serves larger data and/or has performance issues. It can also study the effects of client side and server side caching. This might yield better results than a proxy cache since requests do not have to pass through the proxy server. For this thesis relatively simple eviction policies have been studied. Future work can look into the effects of eviction policies using machine learning on performance. These policies can for example use support vector machines or decision trees [NN18]. Another interesting topic to look into is prefetching. Here the cache predicts the access patterns of users to fetch entries before they are requested. This can be done for example by using Markov Chains [GF15].

## 7 Acknowledgment

The main contribution to this thesis is from Dr. A.W. Laarman. He set up the internship project and allowed for the freedom of working on the thesis project besides it. Most of this project was spent at Accenture, but it was always possible to ask for advise. Dr. A. Uta provided useful insights on technical aspects of the thesis. Another important contribution is from Guillermo Martinez. He allowed for us to experiment with many new technologies and provided the opportunity to work on this research project alongside the internship project. Lastly, Joey van der Wijk helped develop the backend of the Employee Evolution Tree, which the research project revolves around.

## References

[CI97] Pei Cao and Sandy Irani. "Cost-Aware WWW Proxy Caching Algorithms". In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems Monterey, California* (1997).

[Che98] Ludmila Cherkasova. "Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy". In: *Hewlett-Packard Laboratories. 1501 Page Mill Road Palo Alto, CA 94303* (1998).

[WI11] Siti Mariyam Shamsuddin Waleed Ali and Abdul Samad Ismail. "A Survey of Web Caching and Prefetching". In: *Soft Computing Research Group, Faculty of Computer Science and Information System, Universiti Teknologi Malaysia, 81310 Skudai, Johor* (2011).

[Mas12] Mark Masse. *REST API Design Rulebook*. 1005 Gravenstein Highway North, Sebstopol, United States: O'Reilly Media, Inc., 2012.

[Fie14] Ed. Fielding. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. 2014. URL: https://datatracker.ietf.org/doc/html/rfc7234.

[GF15] Arpad Gellert and Adrian Florea. "Web prefetching through efficient prediction by partial matching". In: *Appeared in World Wide Web, Vol. 19, Issue 5, pp. 921-932, USA.* (2015).

[New15] Sam Newman. *Building Microservices*. Sebastopol, United States: O'Reilly Media, Inc., 2015.

[GM17] Roy Friedman Gil Einziger and Ben Manes. "TinyLFU: A Highly Efficient Cache Admission Policy". In: *ACM Trans. Storage 13, 4, Article 35* (2017).

[NN18] Sivaraj Nimishan and Sivaraj Nimishan. "An Approach to Improve the Performance of Web Proxy Cache Replacement Using Machine Learning Techniques". In: *IEEE International Conference on Information and Automation for Sustainability (ICIAfS)* (2018).

[Ric19] Chris Richardson. *Microservices Patterns*. Shelter Island, NY, United States: Manning Publications Co., 2019.

[Clo21] CloudZero. *Horizontal Vs. Vertical Scaling: How Do They Compare?* 2021. URL: https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling#:~:text=While%5C%20horizontal%5C%20scaling%5C%20refers%5C%20to,%5C%2C%5C%20storage%5C%2C%5C%20or%5C%20network%5C%20speed..

[Unk21] Unknown. *Cache invalidation*. 2021. URL: https://en.wikipedia.org/wiki/Cache_invalidation#cite_note-:0-1.

[con22] MDN contributers. *HTTP caching*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching.

## 8 Appendix

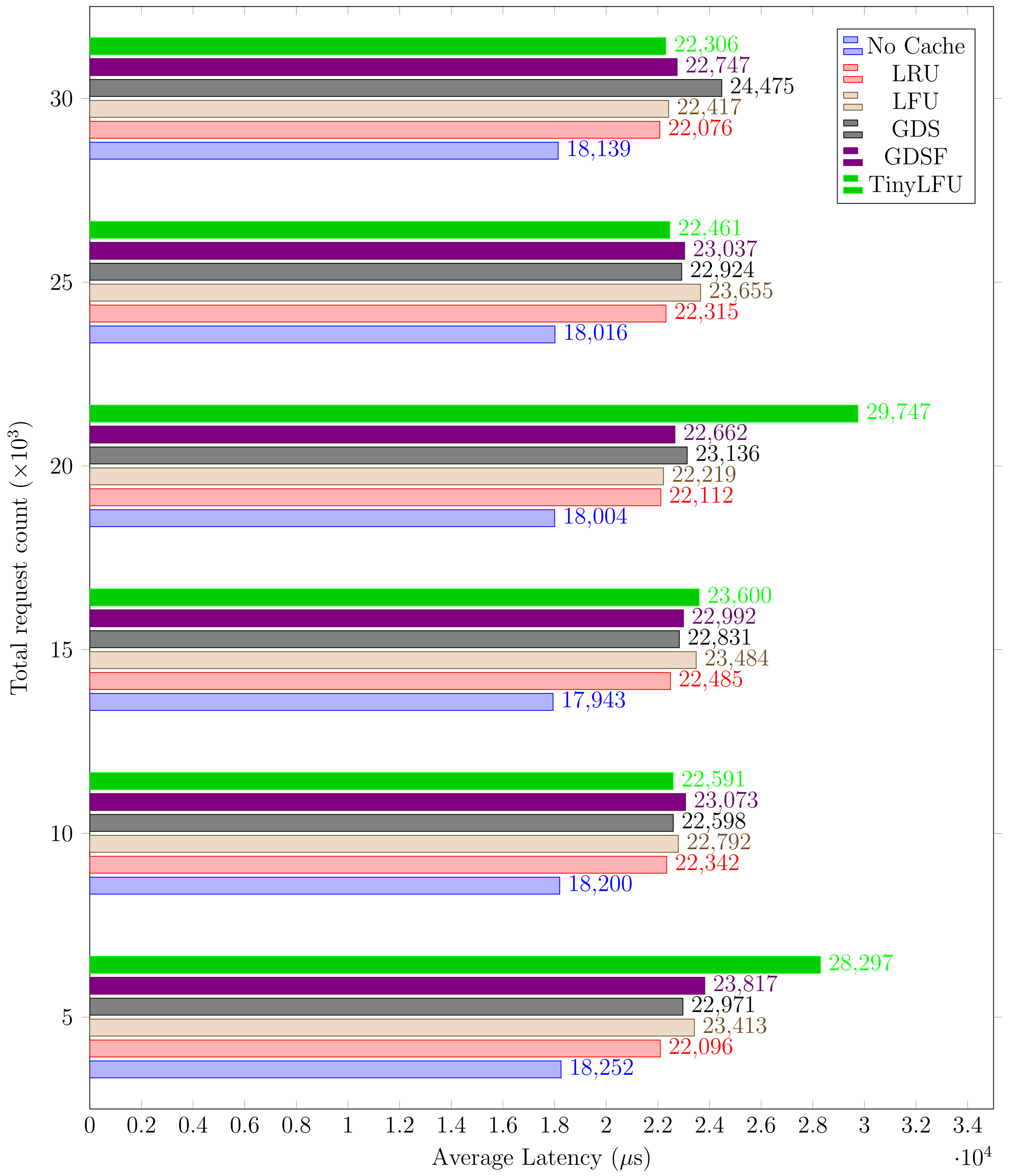

Figure 6: Effect of an increasing request count on average latency

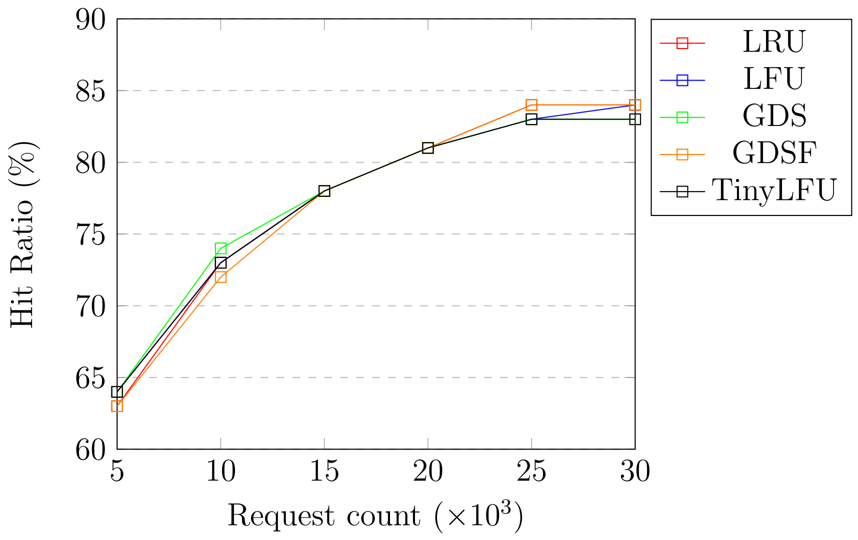

Figure 7: Effect of an increasing request count on hit ratio

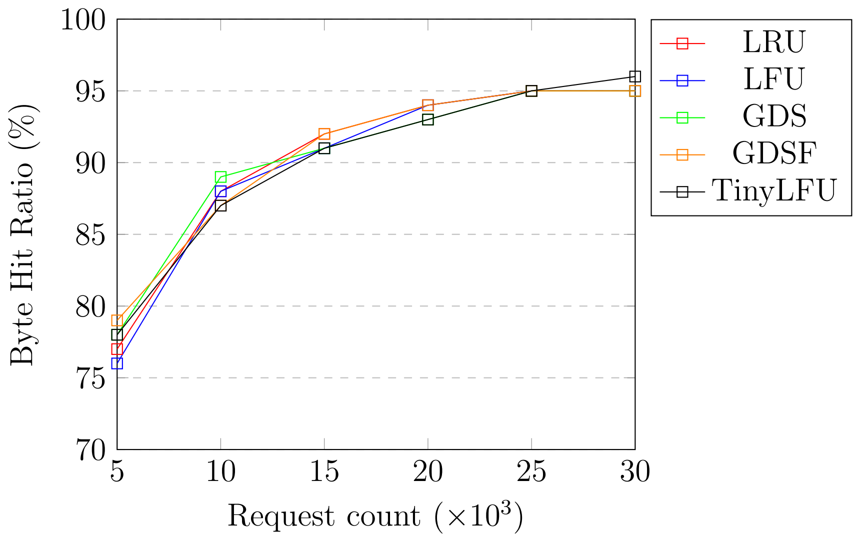

Figure 8: Effect of an increasing request count on byte hit ratio

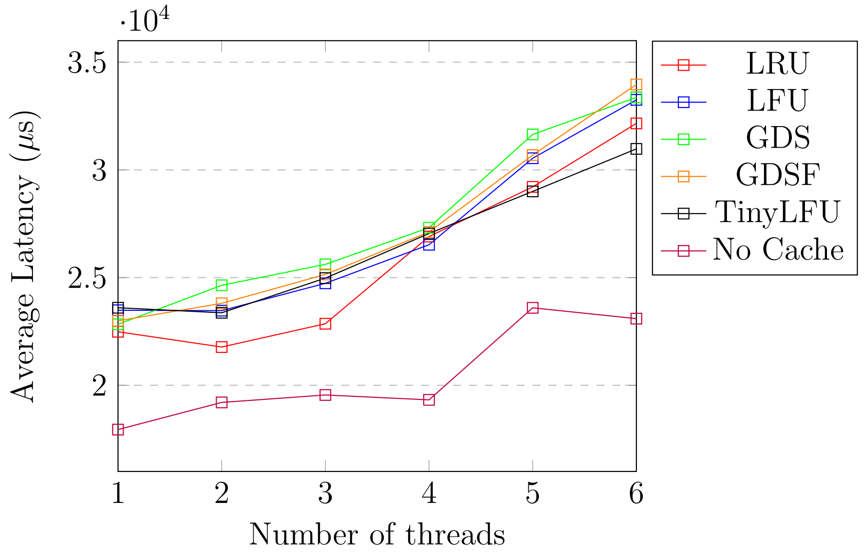

Figure 9: Effect of an increasing amount of concurrent requests on average latency

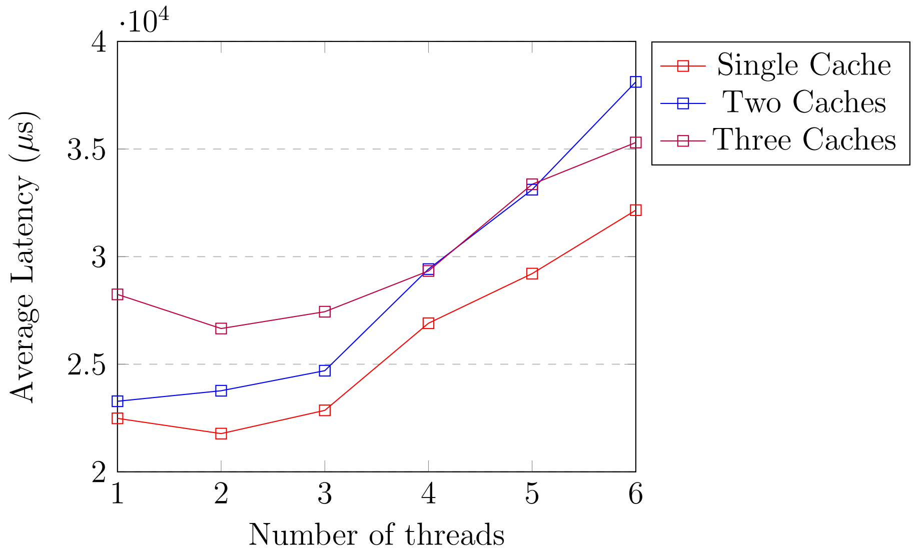

Figure 10: Effect of distributing the cache over multiple instances

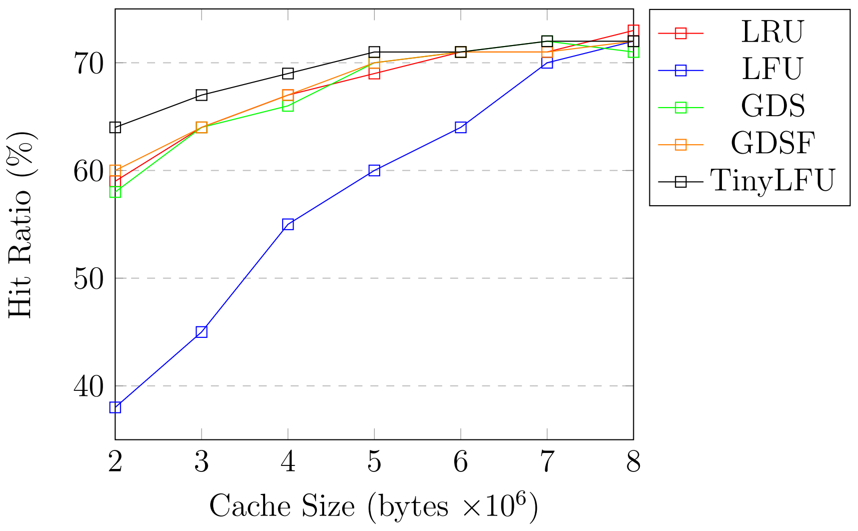

Figure 11: Effect of restricting cache size of hit ratio