Bachelor's thesis

Information and Communications Technology

2025

Eelis Lynne

Designing Secure and Scalable Systems on Public Cloud Platforms

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2025 | 34 pages

Eelis Lynne

Designing Secure and Scalable Systems on Public Cloud Platforms

Public cloud platforms such as Amazon Web Services (AWS) and Microsoft Azure offer on-demand access to cloud computing resources, including servers, storage and databases, for building online applications and services. These platforms operate on a pay-as-you-go pricing model, allowing systems to scale seamlessly from zero to virtually unlimited demand. This pricing model has numerous advantages but also introduces risks of uncontrolled costs during abnormal situations, such as traffic spikes or cyber-attacks.

The objective of this thesis was to demonstrate different methods for securing and protecting web platforms and APIs from common attack vectors such as Distributed Denial of Service (DDoS) and Denial of Wallet (DoW). Another objective was to explore scalable hosting architectures (serverless functions and auto-scaling groups), in-application optimisations (including caching, database indexing and query optimisation), and additional application security measures using rate limiting and CAPTCHA verification.

The combined approach of proactive optimisation with layered security defences has reduced unnecessary cloud expenditure and improved system resilience. It has also mitigated business risks, increased long-term customer satisfaction and enabled continued growth through elastic scaling.

Keywords: public cloud, aws, waf, cloudflare, ddos, cyber-attack

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2025 | 34 sivua

Eelis Lynne

Turvallisten ja skaalautuvien järjestelmien suunnittelu julkisilla pilvialustoilla

Julkiset pilvialustat, kuten Amazon Web Services (AWS) ja Microsoft Azure, tarjoavat tarpeen mukaan käytettäviä pilviresursseja, kuten palvelimia, tallennustilaa ja tietokantoja verkkopalveluiden ja -sovellusten toteuttamiseksi. Nämä alustat toimivat *pay-as-you-go* -mallilla, joka mahdollistaa järjestelmien skaalautumisen nollasta lähes rajattomaan kapasiteettiin. Vaikka malli tuo mukanaan selvät hyödyt, se altistaa myös hallitsemattomille kustannuksille poikkeustilanteissa, kuten liikenteen äkillisissä piikeissä tai kyberhyökkäystilanteissa.

Tämän opinnäytetyön tavoitteena oli demonstroida erilaisia menetelmiä verkkoalustojen ja API-rajapintojen suojaamiseksi ja turvaamiseksi yleisimmiltä hyökkäysvektoreilta, kuten Distributed Denial of Service (DDoS) ja Denial of Wallet (DoW). Toinen tavoite oli tutkia skaalautuvia hosting-arkkitehtuureja (serverless-funktioita ja auto-scaling-ryhmiä), sovelluksen sisäisiä optimointeja (välimuistitus, tietokantaindeksointi ja kyselyoptimointi) sekä lisäturvatoimenpiteitä, kuten ratelimitointi ja CAPTCHA-varmennus.

Ennakoivan optimoinnin yhdistäminen kerroksellisiin suojausratkaisuihin on vähentänyt turhia pilvikuluja ja parantanut järjestelmän sietokykyä. Lisäksi se on vähentänyt liikeriskejä, lisännyt pitkäaikaista asiakastyytyväisyyttä ja mahdollistanut jatkuvan kasvun elastisen skaalautuvuuden kautta.

Asiasanat: julkinen pilvi, aws, waf, cloudflare, ddos, kyberhyökkäykset

Contents

Figures

List of abbreviations

ALB	Application Load Balancer
API	Application Programming Interface
ASG	Auto-scaling Group
AWS	Amazon Web Services
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DNS	Domain Name System
DoW	Denial of Wallet
FaaS	Functions-as-a-Service
GCP	Google Cloud Platform
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
RDS	Relational Database Service
RFC	Request for Comments
S3	Simple Storage Service
SDK	Software Development Kit
SPA	Single Page Application
SSR	Server-Side Rendering
TTL	Time To Live
WAF	Web Application Firewall

1 Introduction

In today's world, the significance of cybersecurity is ever-increasing. Websites and online platforms are facing threats from all over the world, from individuals to government-sponsored hacker groups.

The objective of this thesis is to explore different methods for building secure and protected web platforms and APIs on public cloud platforms, that are protected from attack vectors such as Distributed Denial of Service (DDoS) and Denial of Wallet (DoW) attacks. Another objective is to explore various techniques for making web applications scalable, reliable, and resilient in the event of an attack or legitimate traffic spikes.

Public cloud is an infrastructure model where a cloud provider, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft's Azure, leases out cloud resources, such as compute and storage space, to customers. The cloud provider is responsible for the physical infrastructure and security of the servers, while the customer is responsible for the application and software on those servers. Public cloud platforms provide a variety of services, from raw compute and storage to more specialised services such as managed databases and container clusters [1]. The main benefits of public-cloud platforms include reliability, scalability and ease of development and deployment. Services leased out on public clouds are generally billed using a pay-as-you-go model, meaning cost varies based on the amount of traffic and resources used. This type of billing model can be very cost effective but can also become a liability when an attack such as DDoS or DoW is targeted at a company's cloud services.

Distributed Denial of Service, commonly known as DDoS, is a type of attack where the attacker attempts to overwhelm the resources of the targeted service. The goal of a DDoS attack is to exhaust all of the target's processing and/or bandwidth capacity, resulting in the service being slow or unavailable for legitimate users. The word "Distributed" in DDoS refers to the attack originating from thousands of different sources, each sending only a small amount of traffic, making it difficult to discern between legitimate and malicious traffic.

A Denial of Wallet (DoW) attack is a relatively new type of attack that is aimed at causing financial stress on the target [2]. DoW attacks are usually targeted at websites and platforms hosted on public cloud services. These services are especially vulnerable to DoW attacks because bandwidth is expensive, and additional charges are incurred even if unwanted requests are blocked by services such as AWS WAF.

When designing application and network architecture, being ahead of the curve pays off. Setting up effective defences against cyber threats is much easier when security is considered during the initial design and development, rather than as an afterthought. Threat actors actively scan the internet and public IP space for targets that are susceptible to common attack methods. Once an attack begins, it can be challenging to stop it effectively. When a DDoS attack is launched against a company's infrastructure, it may be difficult or impossible to access the servers, for example, to edit config files to allow additional firewalling.

A DoW attack against a company's public cloud infrastructure can be devastating. Once the billing alarms start firing there may already be thousands of dollars' worth of malicious traffic. During an attack, determining where the cost is coming from and how to stop it can take a considerable amount of time, while the cloud bill continues to stack up.

If a web platform is well defended against attacks, public cloud platforms such as AWS still offer an interesting value proposition in terms of reliability, ease of development, and system observability. This thesis will also look at steps to take when developing and deploying applications to make them secure and scalable.

This thesis will explore the following subjects:

- Protection at the network edge (Cloudflare)
- Scalability on the application server level (AWS)
- Security & performance in the application (PHP/TypeScript)

Although these topics cover different layers of an application stack, they reflect the three main steps the author has taken to build a highly resilient, high-throughput service. The author's business and infrastructure experience weekly DDoS attacks, and since the implementing the measures described in this thesis, none has caused service downtime. The platform provides service to tens of thousands of unique clients every day with an average of nearly three million daily API calls (Figure 1) across a dozen micro- and macroservices. Such scale would not be possible without using scalable hosting stacks and application-level optimisation.

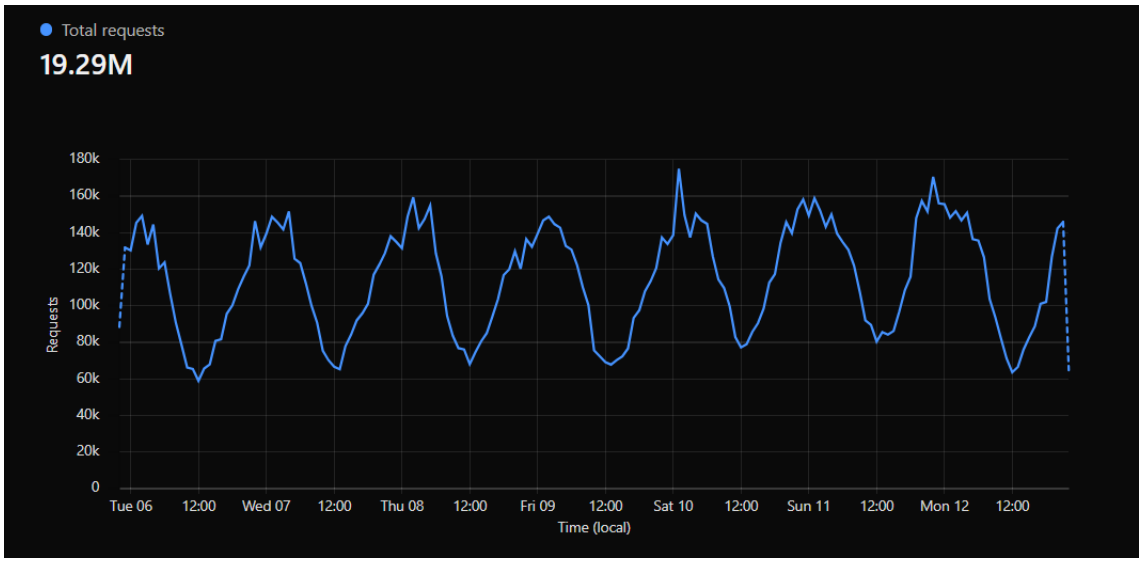

Figure 1. Requests through Cloudflare over a 7-day period, May 2025.

2 Protection at the network edge

Cloudflare is a US-based publicly traded tech company, specialising in CDNs, DDoS protection, cybersecurity and cloud computing. Cloudflare's main services are DNS, automatic DDoS protection, web application firewall (WAF), Pages, Workers.

Cloudflare operates by offering a DNS service, where a domain's nameservers are pointed to Cloudflare. On Cloudflare, the domain can be set up and configured as on any other DNS provider. The difference is that Cloudflare will dynamically update the domain's DNS to point towards Cloudflare's global network, which acts as a reverse proxy in front of the back-end origin. All requests will be routed to Cloudflare, where Cloudflare will inspect the request, determine if it is a legitimate or a malicious request, then either block it or forward it to the origin servers.

The most common use case for Cloudflare is their automatic DDoS protection, which is sufficient to mitigate a good chunk of most attacks. Specialised attackers with advanced DDoS capabilities can bypass these automatic filters. In such cases, Cloudflare offers more advanced tools. The most important of these is their highly configurable WAF service.

2.1 Front-end hosting for resiliency

Many online applications are what are known as Single Page Applications, or SPAs. This means the front-end is a single application which gets loaded on the initial page load. SPAs are usually built with common frameworks such as React, Angular and Vue. The application consists of static files, and all the dynamic content, including page navigation, on the site is rendered using JavaScript.

The advantage of SPAs is that they only have static files, which are easy and cheap to host. Most object stores, such as AWS's S3, offer the ability to host static sites. However, S3 has fees for object retrieval and outbound (egress) traffic and thus is vulnerable to DoW attacks.

Cloudflare offers a static website hosting platform called Pages, that offers unlimited free requests and bandwidth. Cloudflare Pages can be deployed from a ZIP file or from a connected GitHub repository with an option for automatically building a site on commit.

Hosting a front-end on Cloudflare Pages is probably the single easiest step in mitigating low-skilled DDoS and DoW attacks. The attacks experienced by my company are nearly all directed towards statically hosted front-end resources. These attacks can essentially be ignored, as they do not cost anything and only improve the front-end loading times, as the static resources will be cached in more datacentres.

Figure 2 shows a normal pattern for DDoS attacks where tens of millions of requests are thrown at the target website. A vast majority of these requests hit the Cloudflare cache, and the rest hit Cloudflare Pages where no additional cost is incurred.

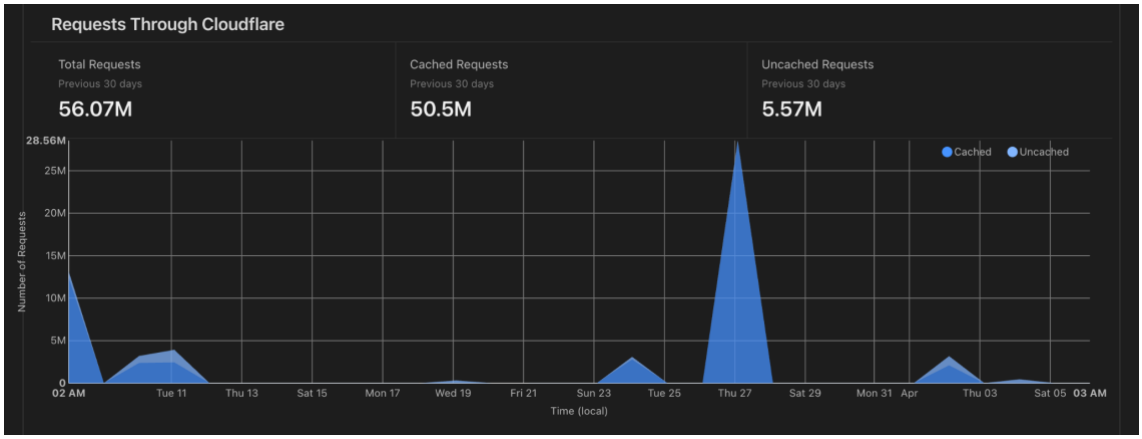

Figure 2. Cloudflare Pages traffic over a a 30-day period.

SPAs used to be more common a few years ago but are now being traded for other rendering patterns such as server-side rendering (SSR), hybrid rendering, and others. The downside of these architectures is that they require server-side compute on the initial page request, which again exposes the application to DDoS and DoW attacks. It is important to weigh these downsides against the benefits and prepare smart mitigation tactics in advance, otherwise the required server compute might become a liability under attack.

2.2 Back-end resiliency

Protecting back-end APIs is one of the most important parts of online cybersecurity architecture. Back-end APIs require server and database compute, which incurs additional server costs. Thus, APIs will always be a target for attacks and there is no free and easy solution for protecting them. Therefore, making attacks more difficult and less effective is crucial.

The starting place for protecting back-end APIs should be an edge web application firewall (WAF).

2.2.1 WAF configuration for protecting APIs

WAF providers such as Cloudflare and AWS allow full customisation through a set of rules, which will be matched against incoming requests. If a request matches a rule, the rule can have an action that will be taken. Common options include blocking the request, requiring a CAPTCHA, or skipping remaining WAF rules. This allows creation of complex behaviours, such as allowing internal requests with a specific authorisation header, requiring specific request attributes, and requiring a CAPTCHA for suspected bots or automated requests.

2.2.2 Referer header checks

The referer HTTP request header indicates the origin from which the request was sent [3]. The referer header is a misspelling of the word “referrer,” which accidentally slipped into the RFC for HTTP/1.0 and was thus set in stone [4]. The referer header can be used for determining the origin of a request, which means it can also be used for security purposes. Requests expected to come from a front-end application, such as an SPA, will include the origin page of the request in the referer header. For example, if a user on an example website navigates to https://example.com/dashboard, which performs an API GET request to https://example.com/api/v1/dashboard/status, that request will contain the header: `Referer: https://example.com/dashboard`
Low level DDoS attacks usually include a blank or faked referrer (Figure 3).

Fake referrers are usually popular sites where real traffic could originate from such as Google and Facebook.

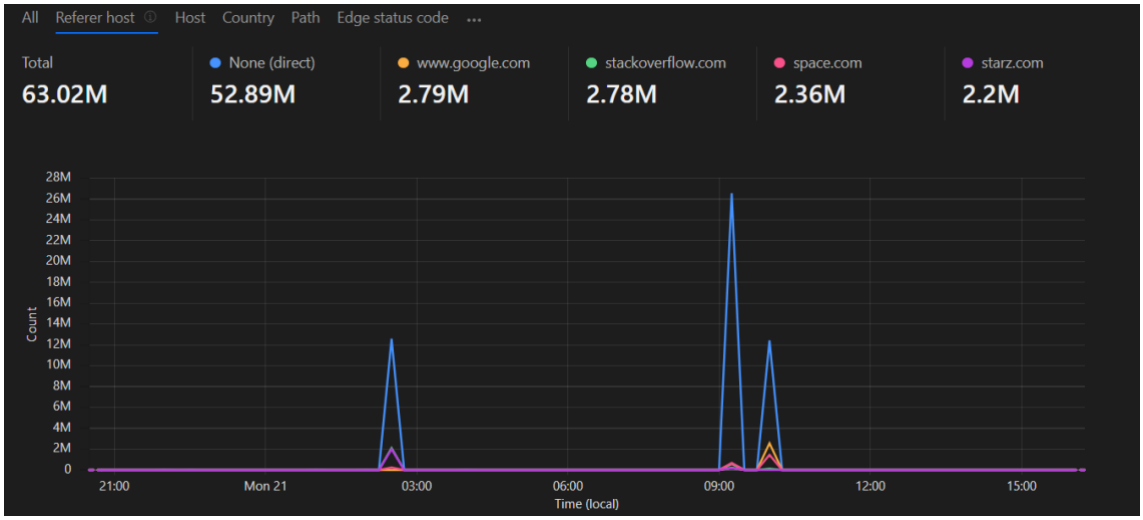

Figure 3. Website traffic, colour separated by the referer header.

Blocking based on the referer header is easy via Cloudflare WAF. The Figure 4 shows an example of a simple block rule that checks if the request is targeting the back-end APIs but has not been referred by the front-end.

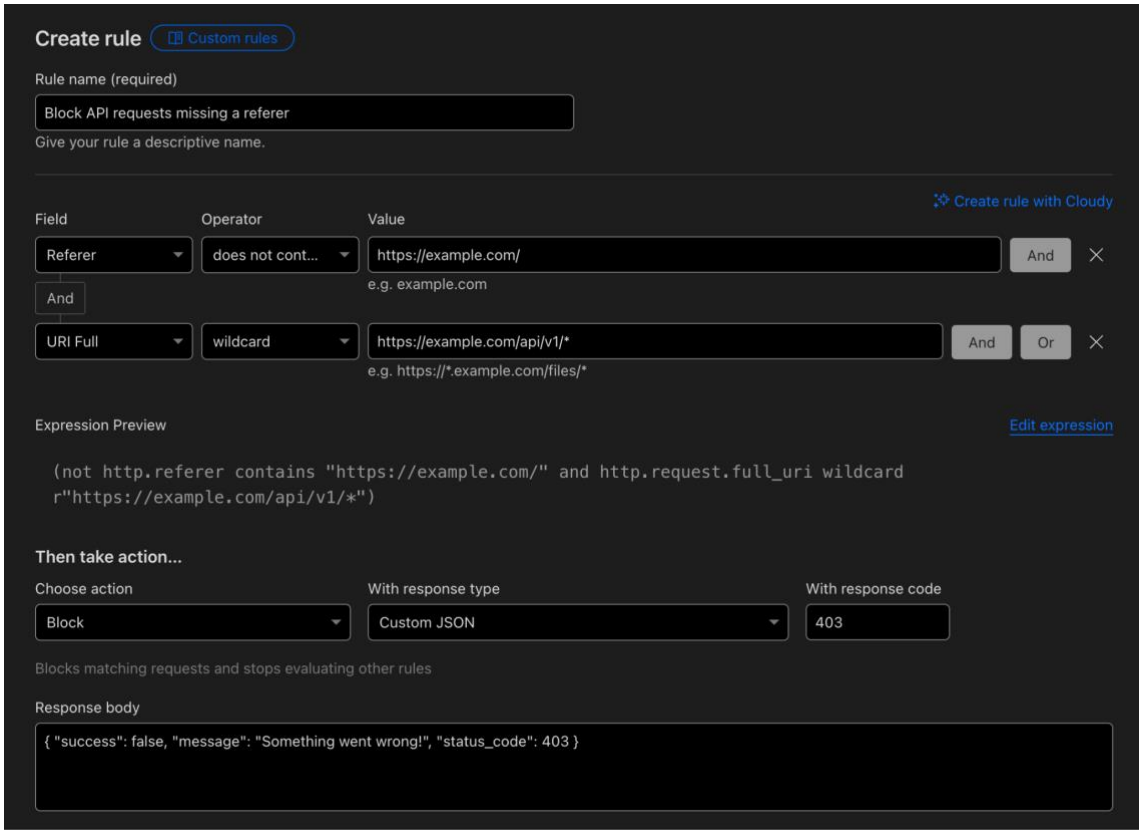

Figure 4. WAF rule blocking API requests missing a valid referer header.

2.2.3 WAF IP rate limiting:

Cloudflare WAF offers built-in rate limiting based on IP address. Cloudflare's Enterprise plan offers advanced rate limiting with more characteristics, such as limiting based on request headers. All other plans, Free, Pro and Business only offer rate limiting based on the client IP address. A simple rate limiting setup can be seen in Figure 5, blocking the client for one minute if they reach over 200 requests per minute.

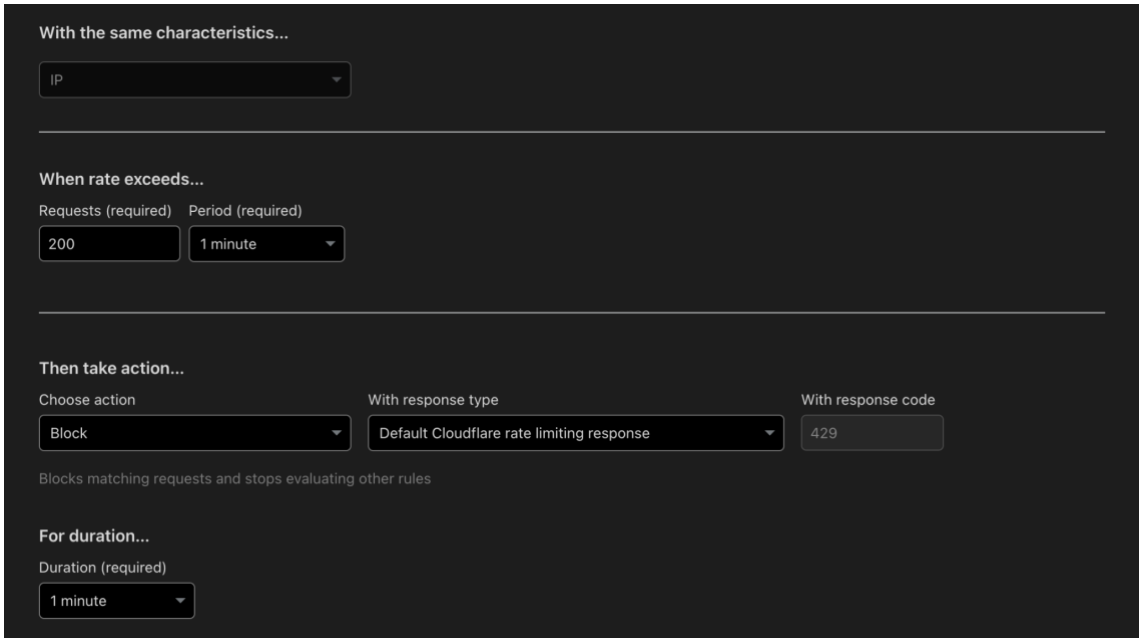

Figure 5. WAF rate limiting rule allowing up to 200 requests per minute.

2.2.4 Header based rate limiting with Cloudflare Snippets

Cloudflare's new feature called Snippets allows running JavaScript code on Cloudflare's edge servers before each request. This feature is available for Cloudflare Pro users without additional cost with unlimited requests. It allows us to perform advanced logic, for example, rate limiting based on custom characteristics such as HTTP request headers. In Figure 6 there is a Snippets code snippet which utilises the cache API to rate limit requests based on the x-verify header. I opted to use the cache API for a few reasons: firstly, it is free for unlimited requests unlike Durable Objects and other Cloudflare state management APIs. Secondly, it is fast since the cache is local to the datacentre, so it does not require propagation across the internet. The only disadvantage is that the rate limit will be datacentre specific [5], but that should not be an issue, when individual datacentres may be dealing with millions of requests under a DDoS attack and only a few hundred are let through by the rate limiter.

```
const CACHE_DURATION_SECONDS = 60;
const RATELIMIT = 100;
const INCLUDE_HEADERS = ["x-verify"]; // Headers to include in the cache key

export default {
    /**
     * @param {Request} request
     */
    async fetch(request, env, ctx) {
        const cache = caches.default;

        const cacheKey = createCacheKey(request);
        console.log(`Retrieving cache for: ${cacheKey.url}.`)

        const cached = await cache.match(cacheKey);

        if (!cached) {
            console.log(`Cache miss for: ${cacheKey.url}. Creating new...`);
            const reqTimesJson = JSON.stringify([ Date.now() ]);
            const reqTimesResponse = new Response(reqTimesJson);

            console.log(`New cache ${cacheKey.url}: ${reqTimesJson}`);

            await cache.put(cacheKey, reqTimesResponse);
        } else {
            console.log(`Cache hit for: ${cacheKey.url}.`);

            const reqTimes = await cached.json();

            // if there's already +1000 entries just skip this and wait for the cache to expire, this is probably a ddos anyways
            // also avoids the snippets 5ms cpu timeout
            if (reqTimes.length > 1_000) {
                return new Response(JSON.stringify({ success: false, code: 429, message: "You are being ratelimited" }), { status: 429 });
            }

            const now = Date.now();
            const activeRequests = reqTimes.filter((x) =>
                now - x < CACHE_DURATION_SECONDS * 1000
            );

            activeRequests.push(now);

            const activeRequestsJson = JSON.stringify(activeRequests);
            console.log(`Updating existing cache ${cacheKey.url}: ${activeRequestsJson}`);

            await cache.put(cacheKey, new Response(activeRequestsJson));

            if (activeRequests.length > RATELIMIT) {
                return new Response(JSON.stringify({ success: false, code: 429, message: "You are being ratelimited" }), { status: 429 });
            }
        }

        // Allow the request to continue
        return fetch(request);
    },
};

/**
 * @param {Request} request - The incoming request object
 * @returns {Request} - A valid cache key based on the URL
 */
function createCacheKey(request) {
    const url = new URL(request.url);
    const cacheKey = new URL(url.origin);

    // map the included headers into the cache key url
    if (INCLUDE_HEADERS.length > 0) {
        const headerParts = INCLUDE_HEADERS.map(header => `${header}=${request.headers.get(header) || ""}`).join("&");
        cacheKey.searchParams.append("headers", headerParts);
    }

    return new Request(cacheKey.toString(), {
        method: "GET"
    });
}
```

Figure 6. Cloudflare Snippets code for rate limiting requests based on headers.

Snippets, however, do have some limitations such as maximum size of 32 KB of code, limited requests to external resources and maximum of 5 milliseconds of CPU execution time [6].

2.2.5 HMAC signing front-end requests

Cloudflare WAF offers an option to validate HMACs at the edge using the *is_timed_hmac_valid_v0* function [7], [8]. The HMAC headers can be used to

validate the request's authenticity and freshness.
The front-end client would add two headers to every request sent to the back-end: the current time and an HMAC signature that includes the request time, method and path. The WAF can validate the request signature and block the request if the header is missing, invalid or expired.

The HMAC hash should include the request path and time concatenated to each other, as seen in Figure 7. The same time variable also needs to be added to its own header so the WAF can generate the same hash and validate it.

```
private addSecurityHeaders(url: string) {
  const now = Math.floor(Date.now() / 1000).toString();
  const urlPath = new URL(url).pathname;

  const hash = CryptoJS.HmacSHA256(
    urlPath + now,
    String.fromCharCode.apply(null, this.hMacSecret)
  );

  const headers: Record<string, string> = {};

  headers['X-Verify'] = encodeURIComponent(
    hash.toString(CryptoJS.enc.Base64)
  );

  headers['X-RequestTime'] = now;
  return headers;
}
```

Figure 7. TypeScript code for generating Cloudflare WAF compatible header HMAC headers using CryptoJS.

Adding the HMAC and time headers should be done using a request middleware or interceptor, so that all requests include the headers and this also avoids potential issues with client-side request caching.

With Cloudflare WAF, the message can be validated using the previously mentioned *is_timed_hmac_valid_v0* as can be seen in Figure 8. The function takes in a string HMAC secret, a string which contains an HMAC to validate, in

format {message}{client_time}-{client_sent_hmac} [9], an integer for how long the HMAC is valid and an integer for the current server-side time.

```
is_timed_hmac_valid_v0(
  "hmac-secret", -- hmac key as a string
  concat( -- message mac start
    http.request.uri.path, -- request path
    http.request.headers["x-requesttime"][0], -- request time
    "-", -- separator
    http.request.headers["x-verify"][0] -- hmac to verify
  ), -- message mac end
  90, -- message ttl, 90 seconds
  http.request.timestamp.sec, -- current timestamp
)
```

Figure 8. Cloudflare WAF Ruleset engine rule to validate HMACs.

This makes automating requests and DDoS attempts more challenging. An attacker would have to manually reverse engineer the HMAC signing code and signing secret out of the front-end JavaScript, which can be challenging depending on the complexity of the application, the libraries used and optionally code obfuscation.

This architecture could also be improved by having logged-in users with a valid session request a signed token from the back-end, for example, every 10 minutes and also validating that on the edge. The API endpoint for requesting the signing token could then be heavily rate limited and cached to prevent it from becoming a target.

2.2.6 Validating public facing APIs with HMACs

If a platform offers a public facing API for which clients can obtain credentials to automate tasks or query data, the API credentials should be delivered in an Access-Key and Secret-Key format, where the Secret-Key is a signed HMAC of the Access-Key. This way the credentials can be cryptographically validated at the edge WAF, without requiring any database queries. This can essentially prevent all DDoS and DoW attacks aimed at public APIs. Once the request has been validated at the edge, the back-end application will, of course, still need to validate if the API credentials are active and have proper authorisation.

Cloudflare WAF requires a time-to-live (TTL) on all HMAC validations. Using zero for the request time, one for the TTL and zero for the current timestamp may seem like an easy solution, however, the WAF Ruleset engine does not accept a hard coded static value for the “current timestamp” argument, so a dynamic value must be used instead. This can be resolved by setting a huge TTL and picking a static timestamp that will be used when generating the API credentials and validating them at the edge. In Figure 9, the selected timestamp is Jan 1, 2030, which is also used for the TTL, making the HMAC valid from 1970 to 2090. That should be sufficient.

```
is_timed_hmac_valid_v0(
  "hmac-api-secret", -- hmac key as a string
  concat( -- message mac start
    http.request.headers["x-accesskey"][0], -- access key
    "1893448800", -- request time
    "-", -- separator
    http.request.headers["x-secretkey"][0] -- secret key
  ), -- message mac end
  1893448800, -- ttl, 60 years
  http.request.timestamp.sec, -- current timestamp
  0, -- length of the separator between request path & request time (op)
  "s" -- flags (optional), s = base64 uri encoded mac
)
```

Figure 9. WAF Ruleset code for validating HMAC based API credentials.

The downside of this approach is that the HMAC’s time-based validation is eliminated, effectively turning it into a traditional API key without an expiration

date. If an application provides an SDK or similar, to interact with the API, it would be recommended to use the secret key to compute a HMAC with the current time for each API request, same way as the front-end client previously described. This approach would allow for the same WAF verification with the added security of timed HMACs but come with additional complexity. The approach of having hard coded, static timestamps offers a lower barrier to entry and makes the developer experience better, as they do not need to write HMAC generation code in their respective coding languages.

2.2.7 Country blacklisting and/or CAPTCHA requirements:

DDoS attacks usually originate from countries with weaker technical infrastructure and countries that aren't traditionally western aligned, such as Indonesia, China, Russia, and India, with the United States being a considerable exception [10]. If a business does not operate or have customers in these countries it may be advantageous to block these countries entirely. Figure 10 below shows the countries with the most traffic to a website hosted on Cloudflare Pages, that has been targeted by multiple DDoS attacks.

Top Traffic Countries / Regions

Previous 30 days

Country / Region	Traffic
Indonesia	14,209,989
United States	9,307,559
China	4,335,645
Russian Federation	2,747,311
India	2,561,334

Figure 10. Request origin countries, 30-day time period, April 2025.

WAF rules should be built as a "black box", where it is not possible for an attacker to determine which specific rule is blocking the request, this can be accomplished by each rule having the same HTTP response code and body.

3 Scalability on the application level

Hosting applications in scalable environments is crucial, as it ensures that during attacks and legitimate traffic spikes the application is still available and able to serve traffic. Applications are generally bottlenecked by two things: the web server serving the traffic, traditionally Apache or Nginx [11], and the database, most commonly MySQL or MongoDB.

These traditional architectures can be bottlenecked by multiple factors, the web servers will first be bottlenecked by the maximum number of active workers, then by the size of the connection pool and finally by number of available file handles. The database will be limited by similar factors, size of the connection pool and so on. Scaling traditional servers is not easy nor efficient, and it will be difficult to know what the next bottleneck will be until an attack is in progress and that bottleneck is being hit.

Two modern solutions for addressing these traditional scalability challenges are serverless environments and managed load balancers with auto-scaling groups.

3.1 Serverless functions

Public cloud platforms, such as AWS, offer Functions as a Service (FaaS). Despite the name “function”, these platforms can host entire applications. AWS Lambda offers a set of supported back-end runtimes, such as NodeJS, Go and .NET (C#) [12]. Lambda also supports running Docker containers, which allows for custom languages such as PHP and Rust.

Serverless functions will not be limited by traditional factors as each request will run in its own isolated environment. AWS Lambda has a default limit of 1,000 concurrent function executions for each AWS region [13]. For example, assuming each request to the Lambda function takes an average of 50 milliseconds, at a concurrency level of 1,000, an AWS region would be able to sustain an average of 20,000 invocations per second [13]. And that is for requests that reach the Lambda, it is not counting requests blocked or cached by edge services such as Cloudflare or AWS CloudFront.

Thus, Lambda's benefit is that its throughput is bottlenecked only by a single pre-determined factor at the HTTP request level: the maximum concurrency. This makes pre-planning and preparation easier as the bottlenecks are already well known. The concurrency limit can also be increased further with approval from AWS.

Even though AWS Lambda makes it easy to determine the bottleneck points, it is not advisable to allow malicious requests through and call the function. A significant consideration with Lambda is its cost. On Lambda there is an expense associated with each request and every millisecond of execution time [14], which can quickly become an issue during a DDoS attack, or act as an attack vector DoW attacks. During a DDoS attack, if edge protections like Cloudflare or AWS CloudFront fail to filter out and cache the malicious traffic, the surge in requests and execution time can lead to rapidly rising costs. Lambda and other FaaS platforms should always have protections in place, even if the concurrency limit would be sufficient to handle a small DDoS attack. The takeaway is that FaaS platforms offer back-ends that are easily manageable and scalable, and that can withstand legitimate spikes in traffic.

3.2 Managed load balancers and auto-scaling groups

An alternative to serverless functions such as AWS Lambda is a managed load balancer such as AWS' Application Load Balancer (ALB) combined with an auto-scaling group (ASG). An ALB can handle many requests and distribute them to a group of compute instances running the application software. An ASG attached to the ALB can automatically scale up and down the number of compute instances handling the traffic based on metrics such as network or CPU load on the current instances [15]. The ALB will perform health checks against the instances to determine if they should receive traffic, if an instance starts failing, it will be shut down and replaced by a new instance, providing additional redundancy and capacity during traffic spikes [16]. However, spinning up new capacity can take multiple minutes, unlike Lambda, which can scale nearly instantly for each incoming request.

ALBs have a static running cost and an additional cost for traffic, however, since the ALB will only handle routing, the cost is much lower than Lambda [17], which also bills for execution time. The type of compute instance used will determine if there is additional costs for CPU time for processing the requests.

3.3 Database scaling

When choosing the hosting architecture for a database, the most important consideration is reliability. Having proper metrics to monitor usage, maintaining backups and having working and tested recovery procedures is essential.

Self-hosting a database on a compute instance can be risky and difficult, as you will have to set up and test backup routines yourself, performance metrics can be limited and difficult to view, and unexpected situations such as running out of disk space can lead to database corruption.

An easy way to achieve high reliability is by using a managed database hosting service such as AWS Relational Database Service (RDS) for relational databases. RDS is a service that runs managed database instances, where responsibility of the underlying system is entirely managed by AWS. RDS provides detailed metrics on the performance and health of the database, automated backups, point-in-time recovery and easy high-availablity multi availability-zone setups [18].

A recent addition to relational database hosting is serverless hosting. Services like PlanetScale and Cloudflare D1 provide SQL databases where even the database instance size is abstracted away [19]. They scale automatically and transparently without any user input and where billing is based on the resources used, such as the number of rows read and written. RDS also now offers serverless hosting with Amazon Aurora Serverless [20].

4 Security & performance within the application

Architectural choices can only go so far in scaling an application. The application must be well optimised to be able to scale with traffic. This applies even for normal traffic spikes which are not originating from an attack. Without caching or optimising database queries, an application may not even be able to handle moderate loads which could occur during real world scenarios. Just paying for more processing power may seem like an easy solution for scaling, but in reality, doing optimisation within an application will provide returns multiple times better than just continuously adding more raw horsepower.

4.1 SQL optimisation

SQL optimisation may seem like a simple topic, but sometimes it is anything but. A standard SQL database, such as MySQL, PostgreSQL, or SQL Server, can easily handle situations where tables contain less than approximately 50,000 rows, even without proper indexes. On SQL databases, an index is a structure that speeds up data retrieval by organising specific columns, similarly to an index in a book. Having an index will help the database only scan the rows relevant to a query, instead of having to look at every row to determine if it matches the query. When databases grow from thousands to millions of rows, even indexed queries can slow down if the SQL optimiser makes a bad call. If the optimizer decides to use the wrong index, it may hurt performance considerably.

Below is a demonstration and a concrete example of an SQL query in need of manual SQL optimisation. Figure 11 shows the schema for a table called “connections_idents” which contains around 85 million rows of data.

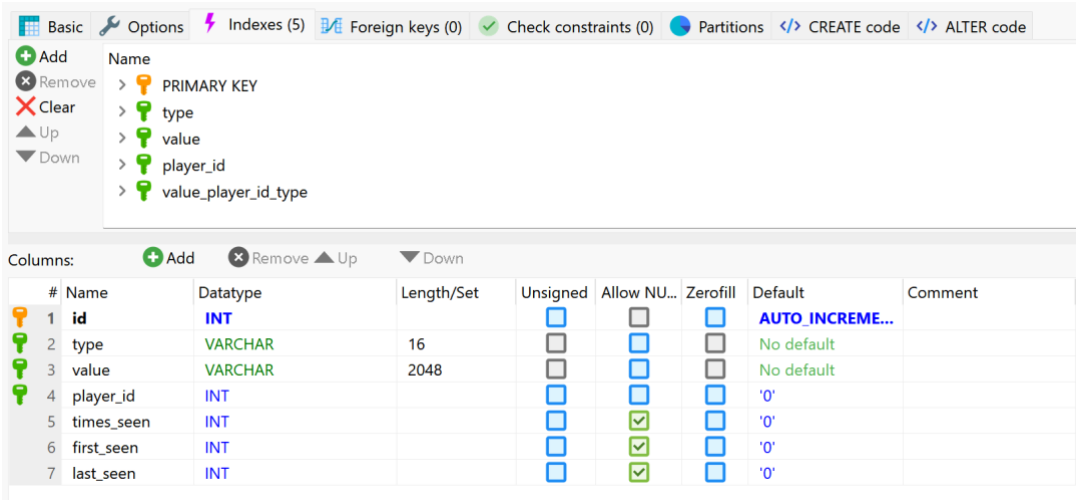

Figure 11. “connections_idents” MySQL table schema.

This is a query that is being run against the “connections_idents” table dozens of times per second:

```
SELECT DISTINCTROW `player_id` FROM `connections_idents` WHERE `type` = ? AND `value` IN (...) AND `player_id` NOT IN (...)
```

With this query the optimizer will use the index on the “type” column instead of the “value_player_id_type” compound index. As a result, this query will take about 90 milliseconds to execute, compared to about 2 milliseconds when manually forcing the compound index with: `USE INDEX (value_player_id_type)`

This is detrimental to performance. This is just a single example, but not a one-off case or limited to MySQL. The same issues apply to other SQL databases such as PostgreSQL, SQL Server and others.

Manually testing and benchmarking SQL queries using `EXPLAIN` and `EXPLAIN ANALYZE` MySQL commands is crucial in discovering issues such as this. Doing the proper optimisation work on database queries will yield much better results than just vertically scaling the hardware. As soon as an SQL database starts reaching these levels of size and traffic manual optimisation stops being optional and becomes mandatory.

4.2 In-app caching

A good way to improve performance and reduce load on the main application database is to implement caching within the application code. Examples of popular cache servers include Memcached, Redis, and a fork of Redis called Valkey. These caching servers usually store everything in memory, making data access much faster than accessing data on disk. They are also key-value based, which allows for fast O(1) lookup speeds; however, this makes them less suitable for complex database operations.

A key-value database stores data in a way where the key works as a look up hash on a table of data. For example, the key can be processed through a mathematical operation that turns it into a number and then that number is used as a row number on an Excel sheet. Instead of having to look through the sheet to find the key, the system can jump directly to the specified row number. This is a very basic explanation of how key-value stores can offer lightning fast read operations.

The caching servers should be deployed in addition to a traditional database within an application architecture. Caching the results of common or complex database queries will improve performance as fewer queries will then hit the actual database, reducing load and also making the remaining queries run faster.

As an example, imagine an API that returns a user's payment methods. This endpoint might do another API call to a payment service provider, do some database queries to get the data into a specific format, and so on. Caching this data would be beneficial as it would reduce database queries, API calls to third party services and reduce loading times for end users. In cases where user-specific data is cached, it is important to keep the cache keys separated, for example by adding the user's ID or some other unique identifier to the cache key with a templated string as such:

```
await redisClient.get(`user:${user.id}:payments:methods`);
```

By separating the cache keys with the users' IDs cached items won't collide with each other and no sensitive information is leaked to other users.

In memory caching servers are built to handle a large number of active clients querying data [21], so it is beneficial to only query data from a caching server instead of a live database until new data is needed or there is a cache miss. Under a high stress situation, such as a traffic spike or DDoS attack the caching server has a much better chance of surviving and being able to serve traffic than a full database.

Platforms such as AWS offer managed caching services with ElastiCache. The cache server will be deployed on a managed compute instance with a static endpoint domain that an application can connect to, similarly, to managed database services such as AWS RDS.

4.3 Security within an application

Assuming the edge layer has now been secured against DDoS and DoW attacks with Cloudflare or some other edge network provider, and the application is using proper caching, and all database operations are optimised, it is time to look at what needs to be done within the application for robust security measures.

4.3.1 CAPTCHAs

CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Humans Apart. They're designed to protect sensitive API endpoints from automation. A CAPTCHA is one of those little boxes that you have to tick to prove you're a human. This may seem pretty easy for a bot to do, but under the hood the CAPTCHA provider will be doing a lot of additional checks to determine if the user is a human. If the user scores low based on metrics such as activity, IP address, etc., they may be given an additional prompt with a picture and told to click on all the fire hydrants or cross walks.

There are different CAPTCHA providers such as Google's reCAPTCHA [22]and Cloudflare's Turnstile [23]. Cloudflare's Turnstile markets itself as a CAPTCHA replacement, but in practice it is the same as reCAPTCHA for the end user.

A CAPTCHA is embedded onto a website by including a piece of JavaScript code in a HTML `<script src="...">` tag, which handles the rendering of the CAPTCHA and any validation checks. Once a user has completed the CAPTCHA the application code will receive a verification token, which should be sent with any subsequent requests which require CAPTCHA validation. On the back-end origin server the client sent token must be sent to the CAPTCHA provider's API to check its authenticity and validity. Only if the CAPTCHA token is valid should the request be handled. This verification workflow is demonstrated in Figure 12 below.

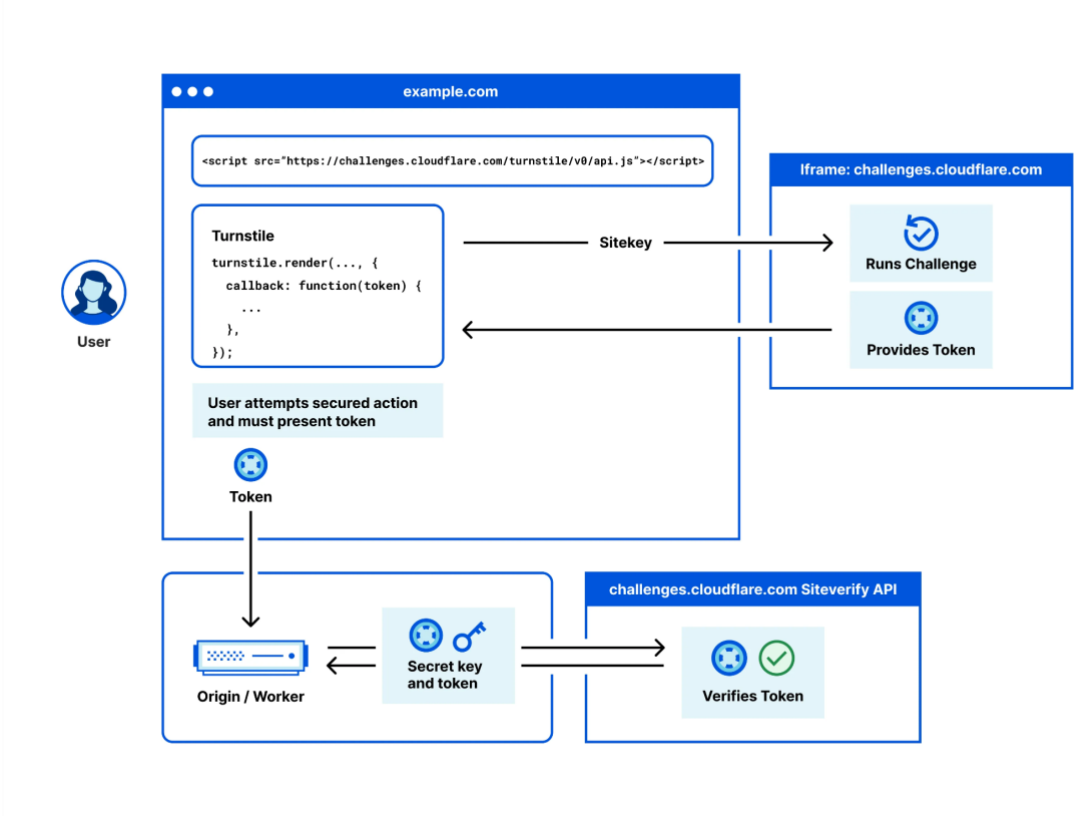

Figure 12. CAPTCHA verification workflow.

CAPTCHAs should be used to protect endpoints which may be security sensitive, require a lot of processing power or expose information that is

susceptible to scraping. For example, requiring a CAPTCHA for login and registration API endpoints is common, along with data search endpoints.

The downside of using CAPTCHAs is that they may be difficult or confusing for less technical people and those with disabilities such as visual or motor impairment. Modern CAPTCHAs are designed to be accessible, but they can still be challenging for certain groups.

4.3.2 Rate limiting

This thesis has already discussed rate limiting on the edge to prevent DDoS and DoW attacks; However, rate limiting within an application is also important. Threat actors may want to scrape data from an application via automation. As mentioned previously, requiring CAPTCHAs on endpoints which are susceptible to scraping is a good way to prevent scraping, but a rate limit is still required to protect against browser scripting and half automated attacks, where a human completes the CAPTCHA.

As an example, if an application exposes a data search endpoint, it may be necessary to apply strict rate limiting to prevent data scraping. An edge WAF might not be suitable for this purpose, as its rate limiting typically applies across the entire application, including requests for static website resources. As discussed earlier, such global rate limits must be set relatively high to avoid blocking legitimate users.

Using endpoint specific rate limits within an application, allows for fine-tuned control. A search endpoint could be very limited, for example, all the way down to 10 requests per minute or 50 requests per day. Because the rate limiting is implemented in code within the application it can be fully customised to fit any purpose.

Rate limiting inherently requires storing a state between requests to determine how many requests are being sent. If an application is hosted in a serverless environment or is written in a language that launches a separate instance for each request (PHP), then an easy way to store the state is to use Redis or

some other caching server. Figure 13 shows example TypeScript code to implement a simple rolling rate limit mechanism using Redis. The rate limiter class is instantiated with the rate limiting key such as the client's IP address or their user ID or session ID/token, max number of requests and the rate limiting period. The `isAtLimitAndIncrement` method performs a Redis lookup on the rate limiting key to check requests within the rolling time window, and it returns true if the rate limit is reached, otherwise it adds the current request to the rate limiter and returns false.

```typescript
// rate limiter
export default class Ratelimiter {
  private redis: Redis;

  constructor(private readonly identifier: string, private readonly limit: number, private readonly interval: number) {
    this.redis = RedisService.getInstance();
  }

  public async isAtLimitAndIncrement(): Promise<boolean> {
    const key = `ratelimit:${this.identifier}`;
    const currentTimes = await this.redis.get<number[]>(key);

    if (!currentTimes) {
      await this.redis.set(key, [Date.now()], 'EX', this.interval);
      return false;
    }

    const now = Date.now();
    const filteredTimes = currentTimes.filter((time) => now - time <= this.interval);

    if (filteredTimes.length >= this.limit) {
      return true;
    }

    await this.redis.set<number[]>(key, [...filteredTimes, now], 'EX', this.interval);
    return false;
  }
}

// request handler
server.post('/api/v1/data/search', async (req, res) => {
  const { query } = req.body;

  // the rate limit could be based on the IP address or the user's ID or session
  const ratelimitKey = req.ip;

  const ratelimiter = new Ratelimiter(ratelimitKey, 10, 60_000);
  const isAtLimit = await ratelimiter.isAtLimitAndIncrement();

  if (isAtLimit) {
    return res.status(429).json({ error: 'Too many requests' });
  }

  const results = await search(query);
  res.json(results);
});
```

Figure 13. TypeScript code for rate limiting requests in-application.

A rolling rate limit means requests with their timestamps are added to a pool as they come in and removed as they go outside of the rate limiting time window. If the size of the pool exceeds the maximum rate limit, then new requests are blocked until older requests expire from the rate limiting pool.

5 Conclusion

This thesis set out to develop and document a layered plan for defence and optimisation for applications on public cloud platforms.

The primary objectives were to demonstrate:

- Effective and cost-efficient solutions for protecting applications and APIs against DDoS and DoW attacks
- Options for hosting scalable, reliable and resilient applications on public cloud platforms
- Security and performance optimisation techniques on the application-level

These techniques have been applied to the author's business and the technology stack that has been used. As a result, no malicious DDoS attack has been able to reach the back-end, or cause downtime over the last three years of operation.

The optimisation on the application-level has allowed growth from a few hundred to over 50 000 unique daily clients. The growth in the number of clients alone may not be impressive, but each client generates more data, which is retained indefinitely and must be queried each time a new client connects. This results in more and more queries being run against a constantly growing historical data set, risking query times ballooning to several seconds, degrading the experience for the clients. The increase in data and the number of queries searching through the expanding data set results in exponential growth in database usage. Handing this would not have been possible without proactively optimising the application and the database.

References

[1] Microsoft, "What is a public cloud?," 2025. [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-public-cloud.

[2] D. Kello, F. G. Glavin and E. Barrett, "Denial of wallet—Defining a looming threat to serverless computing," *Journal of Information Security and Applications,* vol. 60, 2021.

[3] Mozilla Corporation, "Referer," 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Referer.

[4] Wikipedia, "HTTP referer," 2025. [Online]. Available: https://en.wikipedia.org/wiki/HTTP_referer#Etymology.

[5] Cloudflare, Inc, "Cache," 2025. [Online]. Available: https://developers.cloudflare.com/workers/runtime-apis/cache/.

[6] Cloudflare, Inc, "Snippets," 2025. [Online]. Available: https://developers.cloudflare.com/rules/snippets/#limits.

[7] Cloudflare, Inc, "Functions," 2025. [Online]. Available: https://developers.cloudflare.com/ruleset-engine/rules-language/functions/.

[8] Cloudflare, Inc., "Configure token authentication," 2025. [Online]. Available: https://developers.cloudflare.com/waf/custom-rules/use-cases/configure-token-authentication/#option-2-configure-using-waf-custom-rules.

[9] Cloudflare, Inc, "Functions," 2025. [Online]. Available: https://developers.cloudflare.com/ruleset-engine/rules-language/functions/#messagemac.

[10] Yoachimik, Omer; Jorge Pacheco; Cloudflare, Inc., "Record-breaking 5.6 Tbps DDoS attack and global DDoS trends for 2024 Q4," 21 01 2025. [Online]. Available: https://blog.cloudflare.com/ddos-threat-report-for-2024-q4/.

[11] W3Techs, 2025. [Online]. Available: https://w3techs.com/technologies/overview/web_server.

[12] AWS, "Lambda runtimes," 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html.

[13] AWS, "Understanding Lambda function scaling," 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[14] AWS, "AWS Lambda pricing," 2025. [Online]. Available: https://aws.amazon.com/lambda/pricing/.

[15] AWS, "Auto Scaling groups," 2025. [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html.

[16] AWS, "Health checks for instances in an Auto Scaling group," 2025. [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-health-checks.html.

[17] AWS, "Elastic Load Balancing pricing," 2025. [Online]. Available: https://aws.amazon.com/elasticloadbalancing/pricing/.

[18] AWS, "Amazon Relational Database Service," 2025. [Online]. Available: https://aws.amazon.com/rds/.

[19] Cloudflare, Inc, "Cloudflare D1," 2025. [Online]. Available: https://developers.cloudflare.com/d1/.

[20] AWS, "Amazon Aurora Serverless," 2025. [Online]. Available: https://aws.amazon.com/rds/aurora/serverless/.

[21] Redis Inc., "Redis client handling," [Online]. Available: https://redis.io/docs/latest/develop/reference/clients/.

[22] Google, "reCAPTCHA," 2025. [Online]. Available: https://developers.google.com/recaptcha.

[23] Cloudflare, Inc, "Cloudflare Turnstile," 2025. [Online]. Available: https://www.cloudflare.com/en-gb/application-services/products/turnstile/.