Scaleable Fault Tolerant Chess Analysis

Daniel Jones[1 [ae18592@bristol.ac.uk]], Ethan Williams[1 [sk18181@bristol.ac.uk]], and Matthew Stollery[1 [jo18163@bristol.ac.uk]]

University Of Bristol, UK

Abstract. Chess has seen a large increase in popularity in the last year. With many players looking to try chess for the first time, a system for playing the game is needed to be scalable to cope with demand. In this, we lay out the architecture for a scalable, fault-tolerant cloud application for server side processing of chess games, aimed to help new players learn the best moves to play via game analysis.

Keywords: Cloud computing · Scalable · Fault-tolerance.

1 Introduction

1.1 Background

Since the start of 2020, online chess has seen a boom in popularity. Popular online chess server Lichess saw their player count double between February 2020 and February 2021[4], and viewership saw more than a 2,200% increase on the popular streaming site Twitch.tv[10]. Lichess primarily uses their own co-located servers, costing $5,208.75 a month [3] to run. As Lichess is a non-profit [1] organisation, it is important that their costs are minimised. A cloud based implementation could improve scalability and availability and reduce costs. Therefore, our goal is to use microservices along with pay as you go cloud services in order to optimise cost. Microservices, in comparison to monolithic applications, allows us to have small, independent components, using the modularity advantages to lower the scalability costs by only scaling the bottlenecking components. While Lichess rents entire physical servers for use [3], we will only rent portions of servers dependant on load.

1.2 Overview

To represent and analyse Chess games, we will use two representations - FENs and PGNs. A FEN (Fortsyth-Edwards Notation)[2] is a way of encoding the state of a chess board as a string. Board analysis will need to be processed as a stream, where boards may originate from many different users and require near-immediate processing[1]. PGNs (Portable Game Notation)[6] are a way of

[1] As per the definition of streaming data in the context of this coursework: "on event data that arrives at fast pace and requires processing close to event time"

representing entire games rather than single board states with a list of moves and other metadata. These will be processed by workers depending on the task to be performed, using Redis queues, MongoDB databases and S3 buckets to assist in the pipeline. Google Cloud will be used along with GKE (Google Kubernetes Engine) to deploy Kubernetes clusters.

1.3 Aims

The aim of this project is to implement a scalable, fault tolerant system which can:

1. Analyse a chess board state for the best move continuation sequence and score of current position
2. Allow a client to play against a bot
3. Generate analysis data for a whole game
4. Be fault tolerant
5. Scale to a theoretically unlimited number of users (budget permitting)

As a GUI is not pertinent to demonstrating a scalable backend, this project assumes an existing web or mobile app calling the backend to request analysis. This will not affect performance, for example in the case of a web-app this is distributed via a CDN [2]. Simulated clients will be used to stress test the system.

2 Architecture

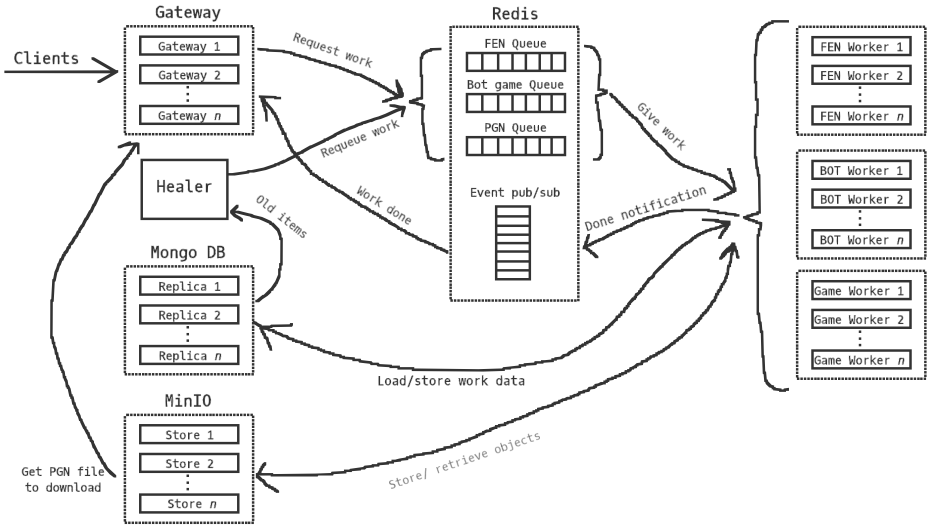

Fig. 1. Cloud Architecture Diagram

[2] Content delivery network. Such as: `https://cloud.google.com/cdn/` and `https://www.cloudflare.com/en-gb/cdn/`

2.1 Google Cloud

Google Cloud was chosen over competing platforms, such as AWS, because of its ease of use, intuitive interface, and abundance of documentation. As well as having a generous free tier and offering $300 of free credit to new users.[3]

2.2 Kubernetes

Kubenetes is used because of its ability to orchestrate a large number of microservices, allowing services to scale independently base on configured metrics all whilst not affecting over services.

GKE autopilot was chosen as it charges based on pods deployed therefore making each pod act as a traditional VM. This is cheaper and more environmentally friendly as pods can be packed onto underlying servers more efficiently.

All microservices are built as Docker containers and ran on a GKE autopilot Kubernetes cluster. Services which require persistent storage, such as MongoDB, are given a "persistent volume claim". Autoscaling in kubernetes is utilised by using HorizontalPodAutoscaler objects for each of the deployments of microservices which comprise the system.

These ensure that the system can keep up with high load when many clients are playing long chess games or requesting analysis, but also will ensure that the components scale down when load is minimal and therefore lowers expenses.[4].

2.3 Gateway

The gateway provides both a REST API and a WebSocket API to interact with the back-end. It is built on top of Quart, an ASGI[5] implementation of Flask, as Quart is able to handle large numbers of requests better than Flask, so is therefore more suitable[9]. As the gateway is stateless, clients can be load balanced between multiple gateways which gives better availability should one gateway crash. The gateway publishes work to be done on the respective work queue, inserts the work into the NoSQL database, and finally returns the completed item. It also will download, via a stream to reduce peak memory use, the generated PGN of a previous game from the Object Store.

2.4 Redis

Redis is used to hold work queues as well as the publish/subscribe mechanism to communicate when work is completed. There are three work queues in total, one corresponding to each worker type (fen analysis, live games, pgn analysis). The pub/sub mechanism is achieved by having requests wait on data to be published to the work item's `uuid` from the respective worker. This notifies the gateway

[3] https://cloud.google.com/free
[4] `https://cloud.google.com/kubernetes-engine`
[5] Asynchronous Server Gateway Interface

that work has been completed, and allows it to fulfil the clients request. Each of these queues in Redis has a unique key which the gateway thread subscribes to while waiting for the data required to be returned to the client.

Our implementation uses only a single Redis instance, even though Redis can be configured to run in a cluster using the leader-follower replication paradigm[7]. This is because a Redis cluster yields no fault tolerance, instead improving latency by offloading $\Omega(n)$ operations to a replica[7]. This would not provide any tangible benefit to our solution as all operations on Redis are $O(1)$, so it would be ineffective to expend energy running Redis as a cluster. Instead, Redis acts as a communication channel between the different microservices. If it should ever fail, it will be restarted by Kubernetes, with its state being restored from disk, and the state will also be kept consistent by the healer pod (Section 2.8). This all means that in the event of failure, it should never go out of sync with the wider system for more than a couple of seconds, and the system can recover quickly.

2.5 MongoDB

MongoDB was chosen as the database choice primarily because of its ease of use, its use of JSON, and it being a noSQL database as relational data is not created in this system. MongoDB is resource efficient in all aspects, cpu, memory and disk storage. Mongo is also fault tolerant as it writes all transactions to fault tolerant disks, which can be used to restore the state in the event of a failure (once mongo is automatically restarted by Kubernetes). MongoDB stores data in JSON format entries called records, which are then inserted into and fetched from collections which represent separate databases. In this application, Mongo serves the purpose of storing the state of games currently in play in the live games collection, the analysis performed on board positions that users have submitted in the fens collection, and the analysis from PGN files in the PGN collection, as well as the status of work on each of these jobs. This information can then be fetched and returned to a user on request, or the stored state can be used to repair the state of the work queues in the case of a failure.

2.6 MinIO

Object storage for the system is handles by MinIO. This is an open source implementation of S3, and is fully compatible with the S3 API, so can be used as a drop-in replacement running on Kubernetes. It is used in this case for storing large PGN files. It can be scaled through "server pools" which are groups of servers which can be set up within the pool for more storage or by adding new pools for geo-replication or higher throughput. Data is distributed using erasure coding, enabling MinIO to be both scalable and fault tolerant. We have a small cluster of four MinIO server instances with 8GB of storage each, but this could be scaled up to thousands of servers with petabytes of storage with ease. In this project it is deployed as a stateful set with node anti affinity, ensuring that the pods are restarted on the same node with the same pod and ensuring that any new pods are created on nodes which don't already have a MinIO instance. [5]

2.7 Workers

Fen Worker The job of the fen worker is to retrieve FEN UUIDs from the Redis queue, and process them using the Stockfish 14 chess engine. It queries MongoDB database using the obtained UUID for a FEN and related information and, if the corresponding document is both non-empty and with status "pending", changes the status to "processing" and begins to work. The FEN it obtains is passed into the engine, and upon receiving the result stores the obtained move sequence with key "result" in the corresponding document. The status is then updated to "done", and the result is pushed onto a separate Redis queue to send the analysis results to the gateway, to then be returned to the client.

Game Worker The job of the game worker is to analyse user uploaded game PGN files. This is done in a similar way to the fen worker, except PGNS are stored in an S3 bucket on MinIO instead of in the database. This is because PGN files can be large (for example, Lichess exports multi-game PGNs that can be in excess of 24GB[4]), and so that PGNs can be downloaded by users again. The game worker then analyses each move in turn, adding the analysis result to an array which, when complete, is stored back in the corresponding document associated with that file in the database.

Bot Worker The job of the bot worker is to take on the role of a chess-playing bot to play against a human. The bot reads in in-progress games with an update from a Redis queue, checks for a termination condition and, if there is none, executes the players move as well as the move indicated by the FEN (corresponding to the best move sequence). The board state is then updated in the database, after which a check is performed to find if the game has a termination condition (i.e. draw, checkmate etc.). This termination condition (if present) is returned under the "state" key in the returned JSON (with 'ongoing' used otherwise). An advantage of the bot only checking for game updates is that a player could initialise a game, close a web socket, then at a later date reopen a websocket and continue the game by supplying a game ID. In addition to this, bots continuously check for any ongoing games with updates, rather than assigning one bot to one game. This is so that bots aren't sitting idle, and games will still get near instantaneous responses.

2.8 Healer

The healer is responsible for re-queuing items which have not been completed within the expected processing time and essentially provides fault-tolerance to the system. Worker failure during processing can be found by querying all items older than 30 seconds with the status "processing", and will re-queue this item for processing. The healer also provides fault tolerance for the pub/sub event notification system, republishing lost notifications.

Only one healer needs to be active at a time and should it fail, another instance can be spun up. This new instance will re-queue all items that were missed in the time it was down.

3 Analysis

3.1 Scaleabilty

Table 1 shows how well the fen analysis worker performs when scaled up to 8 pods. 8 was the maximum achievable for this test because google cloud applied a vCPU limit to accounts using free credit.

Table 1. FEN analysis throughput based on number of workers

Total workers	FENs analysed	Seconds run	Fens per second	Scaling factor vs 1
1	122	600	0.2033333333	-
2	247	600	0.4116666667	2.024590164
4	496	600	0.8266666667	4.06557377
8	952	600	1.586666667	7.803278689

Playing games against a bot with bot-workers is benchmarked using an open source websocket benchmarker with some slight modifications to send chess moves[8]. A worse case scenario[6] is simulated where a client requests a game, and plays a move before closing the connection. If the client kept the connection open and just sent another move, the response time of that next move would be lower as there is less to initialise. The benchmark spawns 128 clients total and performs the request sequence with 16 clients concurrently. It is clear to see that the average time per sequence does not scale linearly, it instead conforms to Ahmdahl's law[7]. The reason for this is likely the overhead required in initialising the game against the bot to start with, though this is part of the problem so cannot be disregarded. Table 2 does however show that as the number of workers increases, the average wait time per client decreases and starts to converge on a constant time, which is the expected behaviour.

Table 2. Latency of FEN analysis

number of bot workers	average time	min time	max time
1	1554.10ms	173.79ms	3100.49ms
2	450.77ms	131.52ms	1758.13ms
4	303.14ms	125.31ms	1413.36ms
8	240.43ms	124.90ms	779.06ms

[6] In terms of being latency expensive
[7] https://en.wikipedia.org/wiki/Amdahl's_law

3.2 Fault tolerance

Fault tolerance is built into every component, and Kubernetes will automatically restart any worker from the FEN, Bot or PGN worker pools. The healer will then re-queue any work that was lost when the pod died. MongoDB and Redis both save their state to redundant disk during operation, and Redis is being run in AOF mode (append only file) so no transactions are lost. All state can be recovered when these pods are restarted but the failure is still disruptive as the system could only function normally when they are running, causing a delay of a few seconds to restart. MinIO uses erasure coding for redundant storage. In the current configuration there are four nodes, and two can be lost while still being able to read data from the rest. The backing storage are GCP persistent disk storage classes which are known to have hardware redundancy. The gateway is run with two or more instances, and therefore is able to load balance incoming requests. When a failure is experienced, all traffic is routed by the Kubernetes service to the pod(s) which remain up. The healer is fault tolerant in that it is not a mission critical component to have running. It is completely stateless and can be killed and restarted at will, carrying on with exactly the same job as it usually would when restarted (which takes seconds with Kubernetes).

In testing the fault tolerance of the above components a chaos engineering method was employed, using the space invaders extensions of the lens IDE [8]. With this extension, one plays a game of space invaders but the aliens represent pods in the Kubernetes cluster. When one is shot, the pod is killed. During testing, the expected behaviour defined above occurs.

3.3 Estimated costing

The bot spends 100ms calculating its response to a player's move, using 0.5 CPU cores while doing so. It has some amount of overhead from retrieving info from the queue and database, but at most this takes an additional 100ms. Assuming 200ms of processing time and 300MB of memory for the bot worker, we can estimate we use 2.78×10^-5 cpu hours per move[9], and 1.67×10^-5 GB hours of RAM per move[10]. With each game lasting for an average of 40 moves, this evaluates to approximately \$0.0000524 per game[11], or \$0.0524 per 1000 games. Note that databases and Redis instances have flat fees, so do not incur additional cost per game. Analysis would scale with demand for the service and would be more computationally expensive than running a game, however the cost for such would not be related to games played.

In comparison, Lichess reports spending \$62,504 per year[3]. Assuming their best period, they spend roughly \$0.053 per 1000 games [12]. Our cloud based

[8] https://github.com/chenhunghan/lens-ext-invaders
[9] $(0.5\text{cpu} * 0.2\text{second}/3600\text{second}) = 2.78 \times 10^-5$
[10] $(0.3\text{gb} * 0.2\text{second}/3600\text{second}) = 1.67 \times 10^-5$
[11] $\$0.00000131 * 40 = \$0.0000524$
[12] In their strongest 12 month period, October 2020 - October 2021 [4]

approach reduces cost per game - however these figures cannot be directly compared as it is unknown the true workload performed on the servers.

4 Summary

To summarise, we succeeded in our goal to implement a scalable, fault-tolerant cloud application for server side processing of chess games. Each of the base aims laid out in section 1.3 have been covered: Chess board states can be analysed with results returned to the user, games can be played against a bot in the backend with a websocket client, users can upload a PGN file which is then analysed, fault tolerance has been tested and confirmed working, ability to scale has been tested and proven in the benchmarking section 3.1, and the architecture is designed such that the system can continue to scale.

There are some possible improvements that could be used to make the system more efficient or performant. The Prometheus metrics stack could have been installed on the kubernetes cluster, which would be used to collect metrics exported from each component of the system. This can be used to scale the number of workers based on other metrics such as queue length rather than CPU load. Another improvement would be to use a MongoDB cluster rather than a single instance, which would be most useful when the amount of data stored in the database has reached a point that is too big for one instance to manage.

A difficulty encountered during the production of the system was accounting for every failure case, it is difficult to forsee every way the system could fail and then build in methods of nullifying these events. Fixing just some of the more common failure cases required making another pod (the healer) which could repair the state of the work queues. One thing that went well while developing the system was that building and deploying containers made development and testing much easier, and a consistent environment within the containers ensured they would work wherever they were deployed.

References

1. Act of creation, `https://www.journal-officiel.gouv.fr/associations/detail-annonce/associations_b/20160025/818`
2. Forsyth-edwards notation, `https://chessprogramming.org/Forsyth-Edwards_Notation`
3. lichess costs.xlsx, `https://lichess.org/costs`
4. lichess.org open database, `https://database.lichess.org/`
5. Minio: Scalable object storage, `https://min.io/product/scalable-object-storage`
6. Pgn for dummies, `https://talkchess.com/forum3/viewtopic.php?f=7&t=78637`
7. Redis replication, `https://redis.io/topics/replication`
8. Websocket benchmarker, `https://github.com/healeycodes/websocket-benchmarker`
9. When to use quart instead, `https://flask.palletsprojects.com/en/2.0.x/async-await/#when-to-use-quart-instead`
10. TwitchTracker: Chess - twitch statistics, `https://twitchtracker.com/games/743`