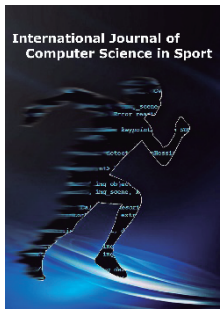
International Journal of Computer Science in Sport

Volume 21, Issue 1, 2022

Journal homepage: http://iacss.org/index.php?id=30

sciendo

DOI: 10.2478/ijcss-2022-0004



Optimizing and dimensioning a data intensive cloud application for soccer player tracking

Gergely Dobreff, Marton Molnar, and Laszlo Toka

MTA-BME Information Systems Research Group, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, Budapest, Hungary.

Abstract

Cloud-based services revolutionize how applications are designed and provisioned in more and more application domains. Operating a cloud application, however, requires careful choices of configuration settings so that the quality of service is acceptable at all times, while cloud costs remain reasonable. We propose an analytical queuing model for cloud resource provisioning that provides an approximation on end-to-end application latency and on cloud resource usage, and we evaluate its performance. We pick an emerging use case of cloud deployment for validation: sports analytics. We have created a low-cost, cloud-based soccer player tracking system. We present the optimization of the cloud-deployed data processing of this system: we set the parameters with the aim of sacrificing as least as possible on accuracy, i.e., quality of service, while keeping latency and cloud costs low. We demonstrate that the analytical model we propose to estimate the end-to-end latency of a microservice-type cloud native application falls within a close range of what the measurements of the real implementation show. The model is therefore suitable for the planning of the cloud deployment costs for microservice-type applications as well.

KEYWORDS: CLOUD NATIVE, MICROSERVICE, DIMENSIONING, SOCCER PLAYER TRACKING

Introduction

Although clouds provide several advantages such as elasticity and a pay-as-you-go model, such characteristics come at a price. One important drawback of clouds is how to estimate the amount of resources to deploy. Depending on the type of application, it may not be simple to estimate the necessary amount of resources. However, determining the amount of resources to be provisioned for the execution of expected workflows is key to achieve cost-efficient resource management and good performance.

There are several related work that tackle the dimensioning of cloud applications via analytical models. In Pietri et al. (2014), a performance prediction model is presented to estimate execution time of scientific workflows for a number of resources. Their results show that the proposed model can predict execution time with an error of less than 20% for over 96.8% of the experiments. The authors of Salah et al. (2015) present an analytical model based on Markov chains to predict the number of cloud instances needed to satisfy a given Service Level Objective (SLO) performance requirement such as response time, throughput, or request loss probability. In order to avoid over-, or under-dimensioning, Coutinho et al. (2017) address the problem of dimensioning the amount of virtual machines in clouds for executing high performance computing scientific applications. In pursuit of faster development cycles, the preference has moved to small decoupled services over monoliths. Following this trend, distributed systems made of microservices have grown in scale and complexity. Creating models of services and systems for characterization and formal analysis can alleviate the issue of dimensioning. Response time is the main interest of Correia et al. (2018): the authors focus on bottleneck detection and propose a method for modeling production services as queuing systems from request traces. Their results show that a simple queuing system with a single queue and multiple homogeneous servers has a small parameter space that can be estimated in production. Jindal et al. (2019) addresses the challenge of identifying the maximal rate of requests that can be served without violating SLO, individually for each microservice. The authors argue that finding individual capacities of microservices ensures the flexibility of the capacity planning for the whole application. The evaluation of their microservice performance models show predictions with mean absolute percentage error less than 10%.

Data analytics in sports has been gaining steam: with novel means of collecting data, applying creative data mining methods and the rise of cloud-deployed big data technologies, both the complexity and the importance of sports analytics, especially in team sports, are steeply increasing. The predominant fraction of sports analytics findings, more complex than box score and play-by-play statistics, rely on the positional data of players throughout the game, whether the investigation targets a team or an individual sport. Hence, player tracking is important for data analysts, and the application scenarios of the positions measured serve technical goals of trainers, business goals of scouts, and entertainment of viewers. For several of the potential use cases it is important to provide tracking data as fast as possible, possibly in real-time, e.g., for on-screen statistics in TV broadcasts, real-time odds on betting portals. Those real-time scenarios would not work, if tracking computation takes too long.

In this paper we propose a football player tracking system that grasps the vision of cheap sports analytics for the masses. The methodological details of the video processing has been published in our earlier paper Csanalosi et al. (2020). Our focus in this paper is to optimize the cloud native operation of the system in terms of accuracy, end-to-end processing latency and costs. Our contribution is three-fold: i) we propose a cloud native design for elastic scaling of the most resource-consuming components; ii) we demonstrate how to optimize the most important parameters of the system, e.g., video resolution, image segmentation (we find that a good quality video stream is essential for an accurate tracking); iii) we build and validate a theoretical queuing model of the system for planning the amount of resource provisioned for the processing of high resolution video streams. The model is then not only used to dimension the cloud application, but also to estimate the expected end-to-end processing latency. The latter is of the utmost importance for the potential application user, especially when real-time analytics is built on the positional data output.

Methods

In this section we discuss the cloud deployment design and the parallelization we have made for speeding up video processing. Furthermore, we present the components of our proposed system along with the most important system parameters, and we list the most recent results of the vast body of research on player tracking in sports and the most widely known commercial tracking products.

Cloud-native system design

In order for our video processing system to meet the requirements to be portable, scalable and inexpensive to operate, we are deploying our system in the cloud. In the following, we describe our considerations and the available virtualization techniques. Then the components of the system, their tasks and the way of communication between them are described.

Our system is implemented using microservice architecture Pautasso et al. (2017). The essence of this pattern is that a complex application is built from independent, separately operated components, where the components communicate with each other through well-defined interfaces. It is also important that the individual components of the system are separately scalable. The resource requirements of various components are not the same, e.g., video processing is a CPU-intensive task, while ingesting the video stream is an I/O intensive task. System components are implemented in Docker Docker (2021) containers, from which multiple instances can run, managed by Kubernetes Kubernetes (2021).

The architecture of our system is shown in Figure1. The system is designed to be able to process a live video stream. This stream is buffered in the Streamer component for a specified period in preparation for possible slow processing and the fact that the image streams of the cameras may not be received at the same time. Addressing the latter problem, the Streamer is responsible for resolving the possible delay between the cameras from the first few seconds of the live image and concatenating the images from the cameras horizontally so that players can be seen throughout the pitch without distortion. Player detection and tracking are performed on the entire image. In addition, this module also defines a mapping between image and real-world coordinates so that later the detected players in the image can be positioned. In addition, the Streamer component includes an RTSP Schulzrinne et al. (1998) server that is responsible for making the concatenated image available after encoding. Encoding is lossless with H.264 codec Richardson (2011).

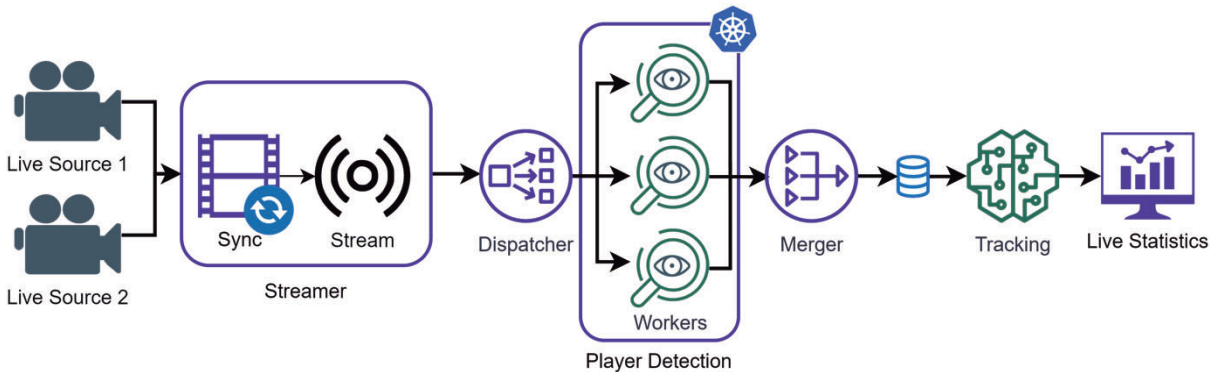


Figure 1. The architecture of our system

Our proposed detection algorithm is not capable of real-time operation, with current video settings and on a single CPU core. Therefore, the system is designed using separate detection and tracking components, and run multiple detection instances in parallel. A requirement on the part of the tracking component is to receive the detected players' positions in ordered sequence. Since the detection algorithm processes individual images as input, multiple detection algorithms can run in parallel to process multiple images in order to achieve the desired processing speed. We perform temporal fragmentation, e.g., if the video is recorded at 30 fps and 5 detection instances are running, each detection instance receives every 5th image, thus, it is as if each gets a video recorded at 6 fps. Thus a Dispatcher component is needed that splits the input video stream coming from the Streamer and a Merger component that queues the results of the detection instances. The actual Dispatcher logic also considers the video encoding characteristics, e.g., number of P frames after an I frame, while striving for minimal buffer time in order to keep end-to-end latency minimal. Once the Merge component has arranged the detection results, it writes the position of the players into a database. The Tracking component reads this database asynchronously and performs the player tracks. Our design choice was to keep the computer vision logic stateless, i.e., the Workers do not consider the previously detected positions of players when they are processing their current batch of frames. Although taking information about the previous frames into account during image processing, the detection algorithm could search players in the neighborhood of a past object, the cloud-native design of our application dictates that states should be externalized from functions as much as possible. Previously detected objects could be read by Workers from the database, but that would introduce a data access delay on par with the potential time saving in terms of detection. The Dispatcher component reads and decodes incoming frames from the Streamer via RTSP. Workers connect to the Dispatcher using message queues. The Dispatcher forwards the frames in a round-robin fashion, ensuring that each Worker receives the appropriate frames. Workers send detection results to Merger via a message queue.

Optical tracking of soccer players

In order to produce the position of each player with fine time granularity and high accuracy, various high-precision player tracking solutions are used: systems that apply wearable devices on the players Catapult (2021); STATSports (2021); PLAYERTEK (2021) and optical tracking solutions that are based on multiple cameras' feeds and video processing techniques Linke et al. (2020); SportVU (2021); ChyronHego (2021); Sentio (2021). Nowadays both categories consist of relatively expensive machinery and expertise, which restricts the usage of those solutions solely in the top level competitions.

We argue that camera-based tracking can be made cheaper by switching from expensive camera systems to fewer and low-cost cameras. Obviously this step results in poorer video quality, but we propose to leverage today's low prices of cloud computing for processing the video streams, and gaining back on the accuracy. Instead of installing high quality custom camera arrays ChyronHego (2021) and spidercams Spidercam (2021), one can install regular action cameras. Only a few of them might be enough, positioned as to get a bird's-eye view over the whole football pitch. The necessary elevation of the cameras allows to install them among the seats of a modest stadium, so the installation costs are negligible compared to what any currently available optical tracking product would cost.

In this section we discuss the related work that propose optical tracking solutions similar to ours, then we provide details about the input video settings that we feed to our system, finally, the detection and tracking methods are described.

Related work on optical player tracking

Several studies have proposed using image processing models for optical tracking of players, and producing input data for sports analytics. Point tracking was studied in Li and Flierl (2012) by proposing scale invariant feature transforms. The player is tracked by calculating the median of motion shifts. Players can be tracked by searching the optimal path in a graph, based on the proposal in Pallavi et al. (2008): a directed acyclic graph is constructed, whereas the nodes are probable player candidates and they are linked in the current frame with the next two consecutive frames. Then the problem of finding the longest path in the graph is formulated as dynamic programming. Another work Muthuraman et al. (2018) proposed detecting players based on the information on jersey color, and a tracker was used for robust tracking of the players. A more advanced solution for this problem is what Iwase and Saito (2004) used with a total number of 15 cameras on both sides of the pitch and got over 50% of the players fully tracked without any errors. Our image processing module implements background/foreground separation, which was applied for the same problem in Baysal and Duygulu (2016), stating that the disengaged foreground can be distracted with the occluding players, however the method is a key step in motion detection.

Input video settings and player detection

In our system, we get the input video from two basic sport cameras SJ7 (2021) and merge these two images together, so we could perform the object detection for the players on each frame on the merged image. The camera placement and alignment can be seen in Figure 2. The video resolution is 2560x1440 pixels for each camera, and its frame rate is 60 frames per second (fps). One important step is to get the points corresponding to players from the video, and another step is to transform these points to a coordinate system which represents the pitch with a width of of 107m and a height of 68m.

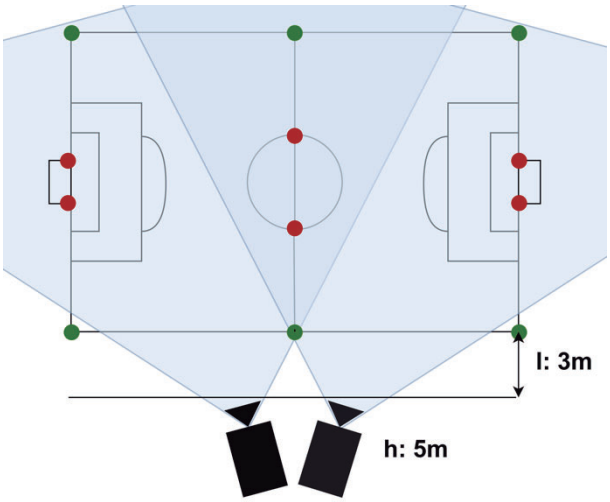


Figure 2. The camera system designed and used for this experiment Csanalosi et al. (2020)

We use the Python library OpenCV OpenCV (2021) to process the video input, frame by frame. On each frame, we perform the object detection in the following steps. We crop the playfield from the image. Then we set up a background model, which helps us detect any kind of movement on the pitch by separating what is in the „foreground“ and what is in the „background“. Next, we remove the noise from the subtracted foreground by applying a

morphological transformation on the image. Finally we seek the exterior contours in this binary image. By the end of the detection procedure, proposed in Csanalosi et al. (2020), a decision is made whether a contour in a given point of the field is possibly a human, by the size of each contour, e.g., at a certain point in the image, the size of a human can be estimated in function of its distance from the cameras. After all, we transform the contours' points to birds-eye view with a perspective transformation method. We measure the color within the contour, and with the knowledge of the current jersey colors, we decide which team the player belongs to.

In Csanalosi et al. (2020) we compared the results of the player recognition algorithm with the original GPS coordinates from the players wearable sensors (from their Catapult vests Catapult (2021)). We tested the algorithm for 2 matches. We calculated the minimum distances between the two sets of points, i.e., between GPS coordinates and the tracked points. The high level statistics are: average distance is 11.6 meters, the median of distance values is 9.0 meters.

Tracking players with Kalman filter and Hungarian method

With the player detection method proposed above, we have a labeled dataset, with the team colors assigned to every player position. There might be missing positions for certain players, this is why we utilized Kalman filter Csanalosi et al. (2020) for tracking. Kalman filter Welch et al. (1995) was designed for navigation control systems, particularly for aircraft and ships. This means that the original application of this filter was on big objects that rarely had quick direction changes. Therefore, in our system, we might face difficulties with a rectilinear motion model for the players, since the human movement - in football, especially - is really altering in every second both in velocity and in direction. However, Kalman filter is one of the simplest methods for tracking multiple objects with a relatively large Gaussian noise rate. The way we approach the problem might produce less accurate results in some cases, but overall we can experience more of the positive effects of Kalman filter, the main steps of which are depicted in Figure 3.

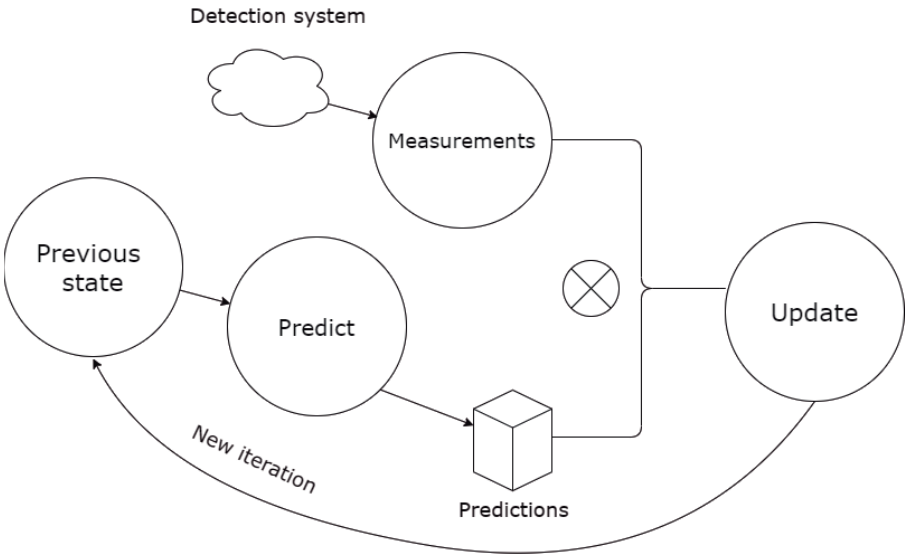


Figure 3. Kalman filter steps in our system

The filter has two methods, predict and update. The prediction step predicts where the next positional coordinates should, or might be. The update step puts these new set of points on the tail of the existing tracks. Kalman filter is used to estimate states based on linear dynamical systems in state space format. Every k-th state comes from the following calculation:

$x_k = Fx_{k-1} + Bu_{k-1} + w_{k-1}$,

where F is the state transition matrix, multiplied with the previous state vector, B is the control input matrix, multiplied with the control vector for the previous state, w_{k-1} is the noise vector, that is assumed to be zero-mean Gaussian distribution with a given covariance Q. The other part of the model is the measurement model which comes in the form of:

$z_k = Hx_k + v_k$,

where z_k is the measurement vector, H is the measurement matrix, and v_k is the measurement noise vector, also assumed to be zero-mean Gaussian distribution with a covariance R.

The goal is to provide x at time k. The initial values x_0, z_1, z_2, ..., z_k are the series of measurements, plus the system information is described by the matrices. Tuning the Q and R matrices may give the desired result for the system. Assuming the matrices are time invariant is key in this application. With these parameters given, the algorithm can start to execute the two steps: prediction, and update. The prediction step is guessing the next estimated state of the system, and predicting an error covariance.

The update step is going to be executed with a calculated variable called Kalman gain, which means the error rate in the measurements. One can instruct the system either to rely on the predicted values, or to use the input data from the measurements with a larger weight. An optimal solution in our case is to adaptively change it, but we have to consider an average noise of 1 meter for every measured point, because our transformation and detection system produces this 1 meter (-0.5m/+0.5m range) error, with the distance from the camera having a large impact on the accuracy of the detection.

This tracking method needs a certain amount of frames to identify the individual tracks, but if we leave it to run for too long, the computation time and the delay will increase. For this issue, we use segmented tracking, in which the algorithm runs for a specific number of consecutive frames, then stops. At this point, we need an association between the player tracks identified in consecutive segments. To solve this problem, we used the Hungarian method Kuhn (1954), which gives us a complete pairing between every single track from the i-th iteration and from the i+1-th.

We chose 50 seconds for the length of a tracking segment as we measured the accuracy with longer intervals and found that points started oscillating when two or more players were in a close range, and although the produced points were correct position-wise, they usually belonged to the wrong player.

In our prior work Csanalosi et al. (2020) we proposed a player detection and tracking system of which we examined the performance by a comparison to an R-CNN-based detection and DeepSort-based tracking on 2 matches worth of data. We created a histogram to illustrate the distribution of the number of recognized players in a video frame, shown in Figure 4.

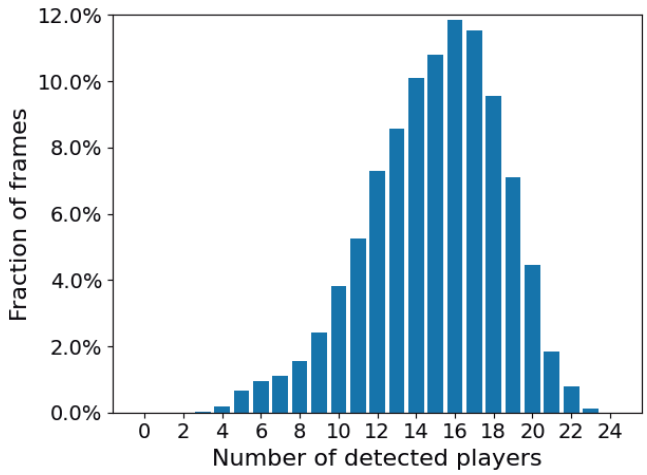


Figure 4. Histogram of the number of detected players per frame yielded by our proposed method

The metric for evaluation of the tracking part is the amount of tracks per frame. The histogram of followed tracks can be seen in Figure 5.

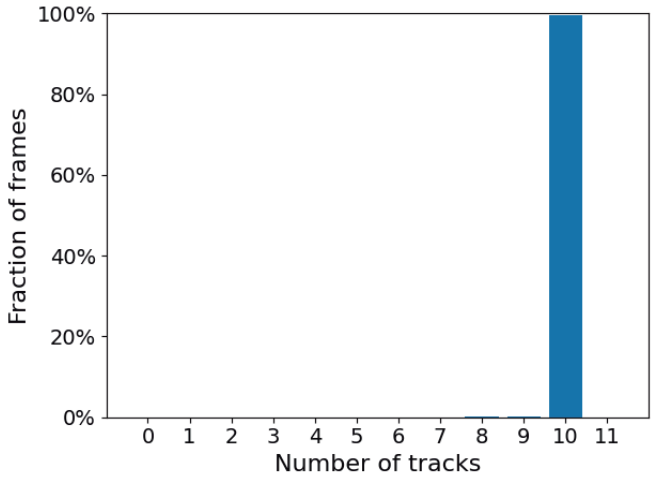


Figure 5. Histogram of the number of tracks per frame yielded by our tracking method

We measured that the average of this metric is 9.8 for the 10 outfield players of a team in the observed 2 matches. We note that the missing recognition of players on the opposite side of the pitch may lead to oscillating tracks and bad results. However, the Kalman filter strives to fill in this missing information and predict where players could have been on the frames we could not detect them. The false-negative detection of R-CNN was high because distant players and players in unusual poses - such as while jumping - were not recognized. The average number of detected players per frame for the test videos was 5 per team, significantly poorer than our system's result, i.e., 7. The median distance between the players detected on the image and the GPS coordinates was 10.5 meters (to be compared to 9.0 meters of our solution). Our solution's results seemed to be slightly more precise in this evaluation aspect. The advantage became

obvious when we took a look at the third performance metric, the number of tracks per frame. While our tracking method continuously tracked all players, the deep learning followed only a fraction of those in many frames. The DeepSort tracking maintained 9.3 tracks on average per frame in the 2 match videos for the 2 teams, goalkeepers (and referees) included; on average a track ended after 42 seconds.

The deep learning solution performed poorly in terms of ID switches as well. Even if the R-CNN model found players accurately, based on the GPS measurements we found that the tracking was not effective because the tracks were often swapped. Therefore despite the fact that a track lived for a long time, it was often associated with multiple players. The number of ID switches were much lower in our proposed method. We examined the performance of the deep learning approach on three manually annotated players. Our observation was that the closer the player to the camera, the better the tracking. During the tracking the three players suffered 44, 34 and 27 ID switches, and an ID lasted on average only for 1.6, 2.1, 2.5 seconds, a much worse result than our proposed system's.

Results

In this section we provide the optimal system parameters for the best achievable accuracy and the shortest runtime of the applied methods. We also describe our model for the end-to-end latency of the video processing pipeline and we summarize the results of our proof-of-concept implementation in terms of accuracy, latency and costs.

System parameter optimization

When it comes to processing a large amount of data with many parameters throughout the system, we might not be able to tell which combination of them is going to give the best possible outcome. Also, the system might be customized for various use cases. One might be able to compute a faster result with lower accuracy, or request slow runtime but with maximized accuracy. We used a grid-search to find the combination of input variables yielding the best results.

Our optimization model consists of four parameters:

- frame rate of the video input [fps],
- video resolution, i.e., image width and height [pixels],
- number of background models applied in the detection,
- state/measurement variance in Kalman filter of tracking.

Frame rate: the more frames we process the runtime naturally increases, but we have the ability of getting more players recognized. Since our Kalman filter predicts on every ``tick'', the results showed us that the accuracy may not benefit from a larger frame rate since we predict most of the coordinates when we have no detected positions.

Image width: When lower resolution images, e.g., 1280x720, are transported and analyzed, the average time needed to process one frame is fairly smaller than with a larger (2560x1440) image. However there is a negative effect in the detection accuracy of the lower resolution, since we lose information.

Grid size of background models: Our image processing contains a phase of background-foreground separation which is processing heavy step. Since this might set a performance bottleneck for our system, we created smaller background models for different segments of the image. We used a grid of i x j in the horizontal and vertical dimensions of the image, where i

and j mean the number of segments in the given direction. Player contours were detected on each background model separately. Obviously processing them in parallel creates a significant performance boost in latency.

State/measurement variance: is connected to the trustworthiness given to each detection by the Kalman filter. At a large parameter value, the filter takes the predictions with larger weight rather than the detected points, but setting it to a small value, the detections are believed to be mostly correct.

The evaluated values of the parameters are the following:

- frame rate: 6, 10, 15, 20 or 30 fps;
- resolution: 1280 x 720, 1536 x 864, 1920 x 1080, 2048 x 1152 or 2560 x 1440;
- grid size: 1, 2, 4, 8 or 16;
- state / measurement variance: range from 0.9 to 1.6.

When evaluating the results, we saw no significant changes in the accuracy when iterating through the domain of the state / measurement variance, so we used the value of 1m.

Accuracy is defined by two aspects: i) detection accuracy, denoted by d in [0,1], ii) tracking inaccuracy, denoted by t in [0,1]. The total number of on-field players are usually 20 (without counting goalkeepers and referees), so we calculate d as the percentage of detected players compared to 20. The tracking inaccuracy t's worst possible value is 1 as this term reflects the average distance of the tracks from the their ground truth locations (scaled to [0,1]). We calculate t relative to the GPS coordinates of the player tracks, collected via wearable sensors Catapult (2021) on each frame. We treat 16-meter and beyond errors as extreme inaccuracies, hence we scale the errors by 16m, and distances above 16m are floored to 1 after scaling.

Let alpha be a variable which sets the weights of d and t in the overall accuracy following the formula:

a = alpha d - (1-alpha) t.

A large alpha means that we expect the best possible detection, a small alpha will give us the closest possible results of the tracks.

The evaluation of our system with various parameter settings were performed on 3x10 minutes of video input from a competitive soccer match. We executed the detection and the tracking for each parameter value combination. We illustrate the impact of parameter values on the accuracy: the results are shown in the left plot of Figure 6 where all parameters share the same x axis: the frame rate, resolution, and grid parameter values are all mapped to the range of [1,5] starting from the smallest value of each parameter, e.g: if the frame rate is 6, its mapped value is 1, if the grid size is 8, its x tick is 4, etc. Note that the steps between two values of a variables are not equal, this projection only gives us a visualization easy to grasp.

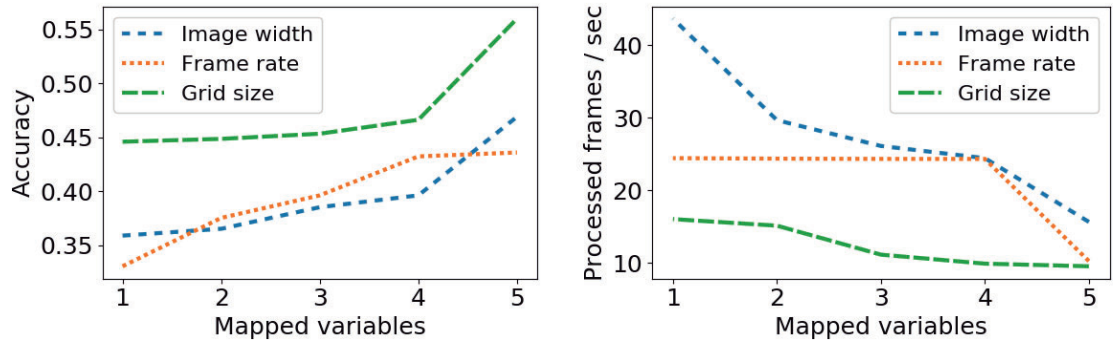


Figure 6. Accuracy and number of frames processed per second vs. parameters

With larger resolution, the accuracy is better. Also, with an increased number of background models (respective to the grid size) used, the accuracy also improves. However, the accuracy is not strictly monotone increasing in function of the frame rate, interestingly we measured the frame rate of 20 to produce the most accurate results, and the larger frame rate did not help to improve the results.

We have also made measurements regarding the runtime of the processing pipeline to complement our study on the accuracy. We looked at processing performance in function of the parameter values, i.e., the number of frames that can be processed in one second. Runtime has a monotone increasing trend in function of the resolution, the number of background models, and the frame rate, too. For depicting the runtime dependency, we use the same mappings and draw the trend in the processed frames in a second. The measurement results are presented in the right hand plot of Figure 6.

To measure how the alpha parameter impacts the performance, we iterated through its 0-1 range with 0.1 steps. For the best accuracy with different alpha values, we got slightly different combinations for our variables, results are shown in Table 1.

Table 1: Optimal parameter values

alpha	0-0.2	0.3-0.6	0.7-1
frame rate	6	10	20
image width	2560 x 1440	2560 x 1440	2560 x 1440
grid size	1	16	16
accuracy (d, t)	(0.61,0.44)	(0.79,0.47)	(0.86,0.53)
throughput [fps]	12	7	9

In summary, to achieve the best performance of the system in terms of accuracy, it is necessary to have a high quality video input, alternatively, one can use more background models to achieve better results on smaller image parts. The accuracy peaks at a frame rate of 20, indicating that a larger frame rate will not make any improvements on the performance of the Kalman filter based tracking algorithm, since every missed detection gives a chance for the Kalman filter to predict

the current coordinates - often not so precisely. The price we have to pay to make a successful detection is to stream high quality images which require more time to be processed. Next, we demonstrate how they can be processed so that the whole pipeline of the system becomes real-time.

Resource provisioning and latency model

Our proposed soccer player tracking solution is a particularly resource-intensive system: processing 2560p video streams from two cameras at 30 fps results in a data flow of 663MB per second. In order to estimate the resource requirements of our system, to define the end-to-end latency of the video stream processing, and the utilization of reserved resources, we created an analytical model of the system using queuing theory. Here we discuss the approximations needed to develop the model, we present the network model itself, and then compare the analytical results with the measurement outcomes in our proof-of-concept prototype.

We model our cloud application with an open Jackson network Jackson (1957) of M/M/1 and M/M/c interconnected servers. Video frames are encoded according to the H.264 codec and these encoded frames reach the processing components via the Internet, so the frames can arrive in bursts as stated by ParandehGheibi et al. (2011). Therefore, our approximation that the arrival rate of the frames follow a Poisson distribution seems acceptable. We make a coarser approximation by modeling the service times with an exponential distribution: the service time of a frame rather follows a normal distribution according to our measurements. As the great benefit, after making these assumptions each performance metric can be obtained in closed form, as among the conditions for a network of several interconnected queues to form a Jackson-network are that the arrival rate of frames from the outside must follow a Poisson distribution, and the service time of each server in the network must follow an exponential distribution. The queuing network of our cloud architecture can be seen in Figure 7, our model is similar to that of Vilaplana et al. (2014). We model the delay induced by the hectic Internet bandwidth by the Streamer Node M/M/1 queue. The Dispatcher receives the images to be processed from the Internet and passes them to the appropriate Worker. The selected Worker performs all resource intensive detection tasks. All Workers are identical having the same service rate and are modeled as M/M/c queuing systems. Each Worker then sends the result of the detection to Merger, which sorts them and then writes the results to a database. Both Dispatcher and Merger are modeled as M/M/1 queues. We use MongoDb MongoDB (2021), a source-available cross-platform document-oriented database server, as it is one of the most widely-used databases for cloud-native applications.

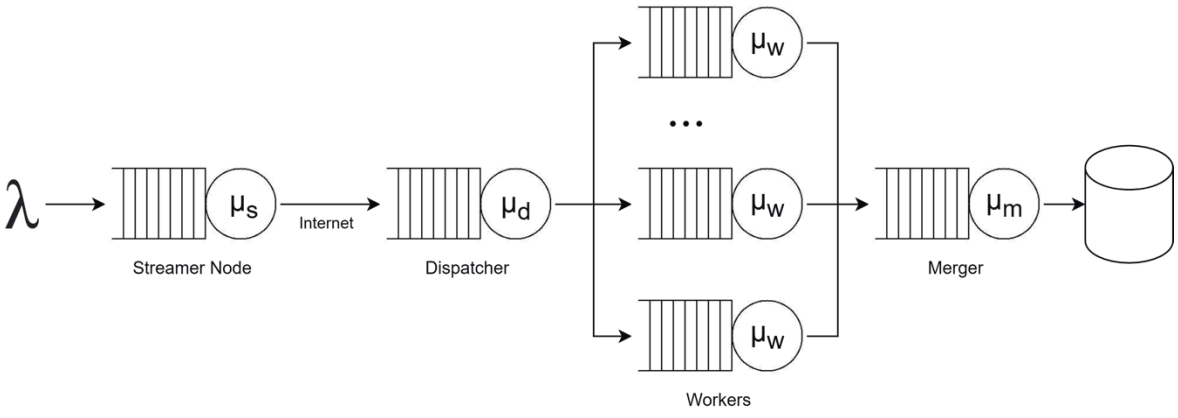


Figure 7. The queuing network of our cloud architecture

The interconnection and behavior between the queues are ruled by Burke's Burke (1956) and Jackson's theorems Jackson (1957, 1963). One of Burke's important findings is that we may

connect many multiple-server nodes together in a feed-forward network and still preserve the node-by-node decomposition when arrival and service times are modeled by exponential density functions. Consequently, Kubernetes pods forming our cloud application can be analyzed independently. It follows that the arrival rate of each line in the network is the same as the arrival rate of the frames coming from the encoder. Therefore only the arrival rate of the frames coming from the encoder and the service time of each queue need to be measured.

The service time of the Streamer Node is defined as B/F, where B is the average Internet bandwidth speed between the cameras and the cloud, and F is the average size of transmitted frames. The average size can be calculated from I and P frames of the encoder: according to our settings, every 244th frame is an I-frame, and e.g., at 2560p resolution the average size of an I-frame and of a P-frame are 8.66 MB and 3.92MB, respectively. For the other queues in the network, the service rate is calculated as the reciprocal of the mean service times. The arrival rate is approximated by a Poisson distribution fitted to the empirical distribution of the number of frames encoded per second.

We examined three main performance metrics of the queues in the queuing network: the response time (waiting and service time), the average number of frames in the system (both in the queue and being processed), and the average queue length. The response time of the entire queue network, as a result of being considered as an open Jackson network, is the sum of each queue's response time. These performance metrics can be obtained in closed form Sztrik (2016).

We implemented the cloud application, and we measured the arrival and service rate of each queue in several experiments. From these values, we calculated the performance metrics for the queues. Among our experiments, there were cases when the system was thrashing Denning (1968): in most of these cases the grid size parameter was 16 or 8. Thrashing occurs when the system is overloaded. In these cases, the measured response time of the system was over 5 seconds, and based on our queuing model, the system was not stable, i.e., the stability condition of the Workers queue was not met. For the rest of the cases, the real response times and those yielded by our queuing model are shown in Figure 8 (left). As long as the Worker Queue utilization (lambda / (c mu)) is below 80%, our analytical model can correctly tell the expected response time of the system, with an average percentage error of 32%. Respectively, our model correctly describes the phenomenon of thrashing when the stability conditions are not satisfied. Based on these results, we state that despite our approximations, we can describe our system with the queuing model at an acceptable error rate.

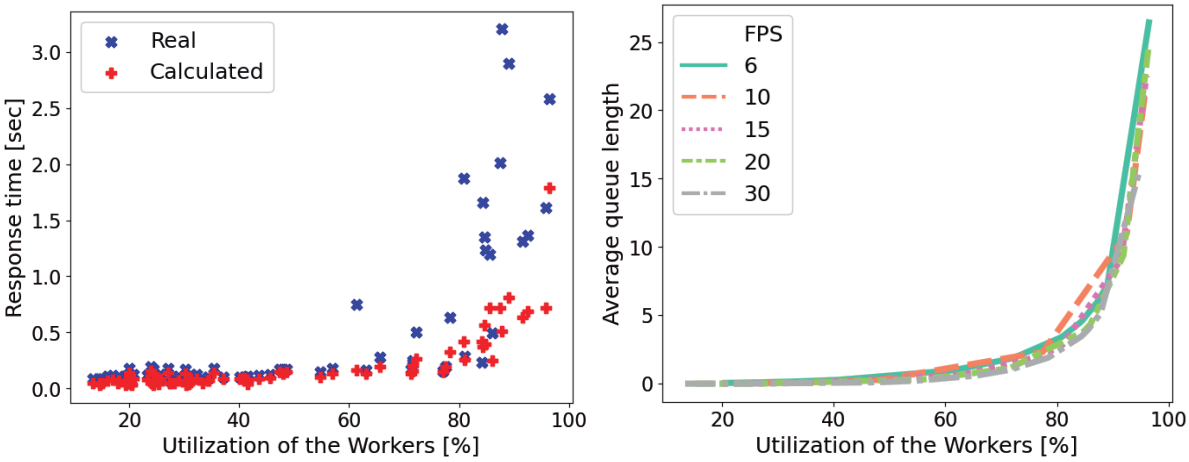


Figure 8. Response times and queue length in function of Worker pod utilization

Using our analytical model, we examined how the stability conditions can be met. Figure 9 shows the minimum number of Workers that are required so that the system becomes stable, given that the grid size is 16. The average queue length is an important metric when designing a system, as it shows how much memory the system will require. The queue length can grow in front of the potentially slowest component, the Workers. Obviously both the length of the queue and the number of frames in the system show similar increase with the response times. Figure 8 (right) shows how the average queue length changes with different frame rates in function of the Worker utilization. Beyond an average utilization of 80%, the queue length starts to increase rapidly in all cases.

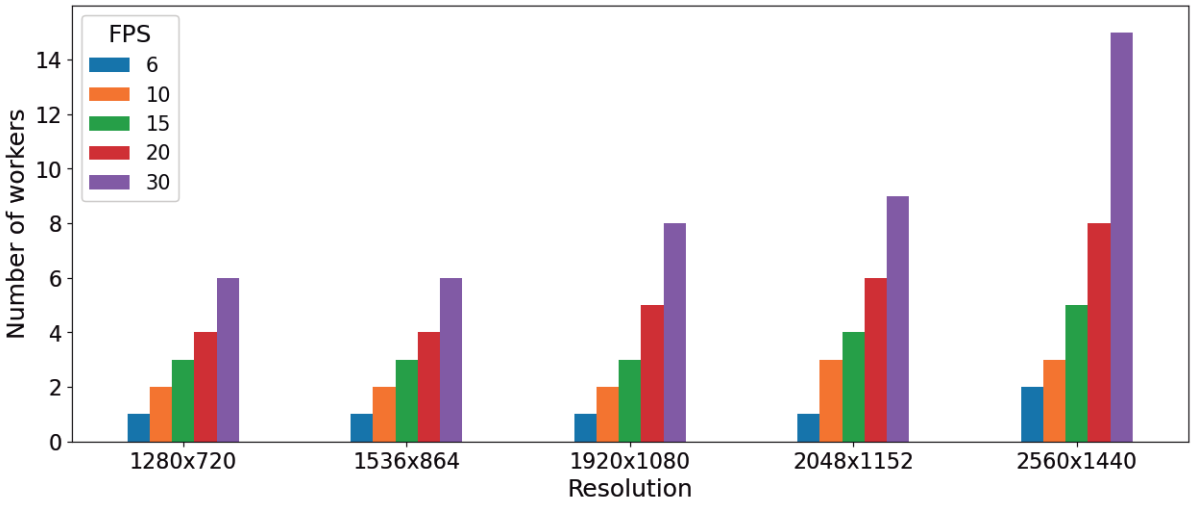


Figure 9. Number of Workers required for stability with grid size of 16

Bottleneck analysis

There are three possible bottlenecks in our system: the encoder, the Internet bandwidth, and the number of parallel Worker pods. The arrival rate of the whole system depends on the encoder, as discussed above. We examine how the system responds when the encoding rate can be arbitrary while performing lossless compression and the sizes of the I and P frames do not change. In our measurements, the system was capable of Gbps order of magnitude of communication bandwidth, however, this is not always the case. We examine how the system performs at different bandwidths. The most resource-intensive part of the system is the detection, the fast processing was achieved by parallelization. We examine how the system responds to different numbers of working pods. In the following, we examine the change in system performance as a function of the value of these parameters.

The existence of a bottleneck can cause undesired phenomena, such as thrashing and running out of memory. This means that the frames to be processed are congested in one of the queues, because the given server cannot serve the requests at the same rate as they arrive, so the queue of the frames to be processed may grow to infinity. Since a frame is relatively large, the server can quickly run out of memory. Considering the queuing model, this occurs when the given queue is no longer stable. Stability condition in case M/M/1 is lambda < mu, and in case of M/M/c it is lambda < c mu.

We examine the bottlenecks for the most resource-intensive configuration: 2.5K@30fps. Changing the encoding speed, the frames arrive between the Streamer and the Dispatcher at such

a rate that the system cannot transmit at the given bandwidth thus the frames are congested in the queue. In Figure 10, we can observe this phenomenon for different bandwidths.At a given encoding rate, in order for the system to remain stable and the aforementioned phenomenon not to occur, the system must have more than a certain bandwidth, as can be seen in Figure 11.

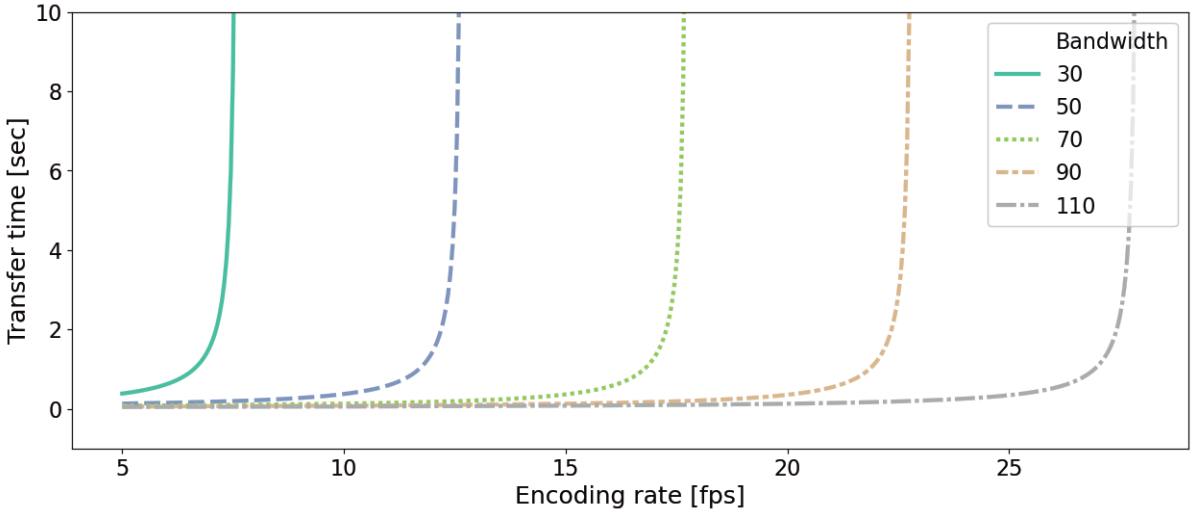


Figure 10. Transfer time as a function of encoding rate at given bandwidths [MB/s]

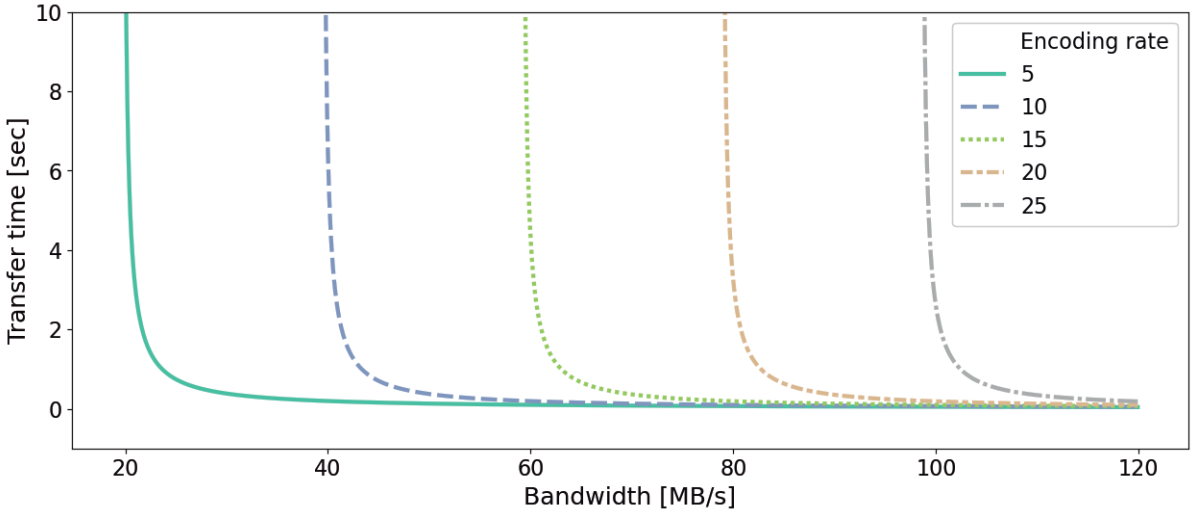


Figure 11. Transfer time as a function of bandwidth at given encoding rates [fps]

Workers are a potential bottleneck, if there are too few processing units working in parallel, each frame will be processed more slowly than the rate it arrives with. Figure 12 shows the response time for different numbers of workers at a given encoding rate. Where no response time is indicated, the system is not stable. In the most resource-intensive configuration, i.e., 2560p@30fps, the response time of the queue does not decrease significantly beyond 4 Workers, but below 3 Workers the system is thrashing.

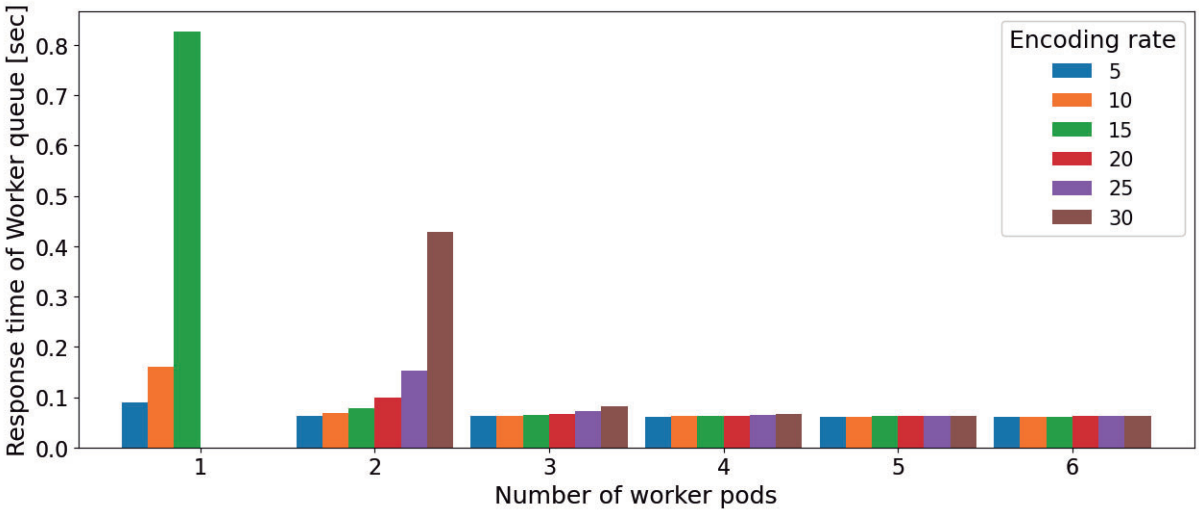


Figure 12 Worker queue response time as a function of the number of Workers at a given encoding rate [fps]

Discussion

Based on our findings, we conclude that the most accurate tracking is obtained with the following parameters: 2560p@20fps when the grid size is 16. Based on our observations (see Figure 9) in the previous section, at least 8 Worker pods are required for this configuration. In this case, processing a complete 90-min match can be done real-time with a delay of 1.08 sec in the system based on the response times of the components. Thus, with Amazon Web Services, processing a full match would cost $1.2 (using the smallest available compute-optimized VM (c5.large) at $0.1 per hour Amazon (2021)).

Private and public cloud systems are increasingly becoming the leading execution environment of applications due to their configurability, robustness and elasticity. Providers are able to set and change the resources needed for their application dynamically thereby optimizing the operating cost. Indeed, one of the most important aspects of resource management in cloud environment is scaling, i.e., how one can change the amount of resources under the application. Horizontal scaling changes the number of instances (pods, virtual machines, etc.), while vertical scaling modifies the resources under a given instance (CPU, memory, etc.). This can be done manually by the provider, however several cloud systems provide auto-scaling functionality out of the box. Although our presented study assumes that the initial configuration of components fits the load produced by the incoming video streams, in case the input intensity becomes so dynamic that the initial capacity cannot process the increasing buffers, Kubernetes' autoscaler feature HPA (2021) can scale out the bootleneck components, e.g., Workers.

In this work we deliberately opted for CPU-based video processing, as we aimed for a cost-efficient processing pipeline suited for the visual analysis of a single football match, recorded by a few cameras (2 cameras in the presented example). GPU-paired cloud instances are relatively expensive, e.g., compare the compute optimized c5.large AWS instance with 2 CPU cores 4 GiB of memory at $0.085 per hour with the cheapest GPU accelerated instance g4ad.xlarge at $0.379 per hour (4 CPU cores, 16 GiB of memory) Amazon (2021). For a large-scale operation where a high number of video streams must be processed concurrently, the GPU-based processing might be a better solution than scaling out CPU-based video processing units.

The application scenarios of our developed system are real-time use cases for one or a few football matches, e.g., real-time data analysis assisting coach decisions during a match.

Limitations

The evaluation of our system with various parameter settings were performed on 3x10 minutes of video input from a competitive soccer match. In our prior work Csanalosi et al. (2020) we evaluated the accuracy of player detection and tracking on video segments produced in 3 different matches, and we found out that it is indeed problematic to evaluate the system quality on one match only, since the quality depends on the optical situation, e.g., light context. The cameras' distance from the pitch and the height of the camera position were fixed in all cases. In the current manuscript, however, we have not focused on the detection accuracy, rather on the cloud-native deployment design of the application. Consequently we have not experienced significantly differences on any of the metrics we focused on in this work depending on the match the input video was originated from.

The application domain of our proposed system is limited to such tactical analysis where no ball tracking is required.

Conclusion

As more and more applications are designed to be deployed in the public cloud, the application providers need to fine-tune the configuration parameters of the application with a great attention to the cloud-native design and its peculiarities. The configuration parameters not only define the quality of service, but the resource consumption as well. In this paper we demonstrated that a simple queuing theory-based analytical model is capable of pinpointing the potential bottlenecks of the microservice-type application by looking at the performance of the cloud native components separately. The result is somewhat surprising, as strong assumptions are made on arrival and service times so that important system performance indicators could be calculated easily with closed-form formulas. This modeling tool provides precious insights in the deployment planning phase and adds solid basis for the cost estimation of any real-time cloud native application.

As a showcase of such an application, we presented a novel low-cost optical tracking system for soccer players, which is based on custom visual object detection methods, Kalman filter and Hungarian algorithm for tracking.We demonstrated how our optical tracking system can be optimized in accuracy, processing latency and resource consumption. Most importantly, we validated the proposed analytical model with an implementation of the system: the measurements we performed in the Kubernetes deployment proved that the analytical model's results are correct.

Acknowledgements

Project no. 128233 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the FK_18 funding scheme.

References

Amazon (2021). AWS Pricing. https://aws.amazon.com/pricing/.

Baysal, S. and Duygulu, P. (2016). Sentioscope: A soccer player tracking system using model field particles. *IEEE Transactions on Circuits and Systems for Video Technology, 26*(7):1350–1362.

Burke, P. J. (1956). The output of a queuing system. *Operations research, 4*(6):699–704.

Catapult (2021). Wearable Technology. https://www.catapultsports.com/.

ChyronHego (2021). The leading sports tracking solution. https://chyronhego.com/products/sports-tracking/.

Correia, J., Ribeiro, F., Filipe, R., Arauio, F., and Cardoso, J. (2018). Response time characterization of microservice-based systems. In IEEE 17th International Symposium on Network Computing and Applications (NCA), pages 1–5.

Coutinho, R., Frota, Y., Ocaña, K., de Oliveira, D., and Drummond, L. M. A. (2017). Mirror Mirror on the Wall, How Do I Dimension My Cloud After All?, pages 27–58. Springer International Publishing, Cloud Computing: Principles, Systems and Applications.

Csanalosi, G., Dobreff, G., Pasic, A., Molnar, M., and Toka, L. (2020). Low-cost optical tracking of soccer players. In Workshop on Machine Learning and Data Mining for Sports Analytics (MLSA).

Denning, P. J. (1968). Thrashing: Its causes and prevention. In Fall Joint Computer Conference, Part I, AFIPS '68 (Fall, part I), page 915–922. ACM.

Docker (2021). Docker. https://www.docker.com/.

HPA (2021). Kubernetes Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Iwase, S. and Saito, H. (2004). Parallel tracking of all soccer players by integrating detected positions in multiple view images. In Proceedings of the 17th International Conference on Pattern Recognition (ICPR).

Jackson, J. R. (1957). Networks of waiting lines. *Operations research, 5*(4):518–521.

Jackson, J. R. (1963). Jobshop-like queueing systems. *Management science, 10*(1):131–142.

Jindal, A., Podolskiy, V., and Gerndt, M. (2019). Performance modeling for cloud microservice applications. In ACM/SPEC International Conference on Performance Engineering, page 25–32.

Kubernetes (2021). Kubernetes. https://kubernetes.io/.

Kuhn, H. W. (1954). The Hungarian method for the assignment problem. In *Naval Research Logistics Quarterly*, volume 2, pages 83–97.

Li, H. and Flierl, M. (2012). Sift-based multi-view cooperative tracking for soccer video. In IEEE International Conference on Acoustics, Speech and Signal Processing.

Linke, D., Link, D., and Lames, M. (2020). Football-specific validity of tracab's optical video tracking systems. *PLOS ONE, 15*(3):1–17.

MongoDB (2021). MongoDB: The most popular database for modern apps. https://www.mongodb.com/.

Muthuraman, K., Joshi, P., and Kiran Raman, S. (2018). Vision based dynamic offside line marker for soccer games. Technical report, arXiv:1804.06438.

OpenCV (2021). Wrapper package for OpenCV python bindings. https://pypi.org/project/opencv-python/.

Pallavi, V., Mukherjee, J., Majumdar, A. K., and Sural, S. (2008). Graph-based multiplayer detection and tracking in broadcast soccer videos. *IEEE Transactions on Multimedia, 10*(5):794–805.

ParandehGheibi, A., Médard, M., Ozdaglar, A., and Shakkottai, S. (2011). Avoiding interruptions—A QoE reliability function for streaming media applications. *IEEE Journal on Selected Areas in Communications, 29*(5):1064–1074.

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., and Josuttis, N. (2017). Microservices in practice, part 1: Reality check and service design. *IEEE Software, 34*(1):91–98.

Pietri, I., Juve, G., Deelman, E., and Sakellariou, R. (2014). A performance model to estimate execution time of scientific workflows on the cloud. In 9th Workshop on Workflows in Support of Large-Scale Science, pages 11–19.

PLAYERTEK (2021). GPS player tracking system. https://www.playertek.com.
Richardson, I. E. (2011). The H. 264 advanced video compression standard. John Wiley & Sons.
Salah, K., Elbadawi, K., and Boutaba, R. (2015). An analytical model for estimating cloud resources of elastic services. *Journal of Network and Systems Management*, 24.
Schulzrinne, H., Rao, A., and Lanphier, R. (1998). Real Time Streaming Protocol (RTSP). Technical Report 2326, RFC.
Sentio (2021). Sports Analytics. https://sentiosports.com/.
SJ7 (2021). SJ7 STAR Camera official website. https://sjcam.com/product/sj7/.
Spidercam (2021). spidercam FIELD. https://www.spidercam.tv/.
SportVU (2021). SportVU 2.0 by Stats Perform. https://www.statsperform.com/team-performance/football-performance/.
STATSports (2021). Apex Athlete Series. https://statsports.com/apex-athlete-series/.
Sztrik, J. (2016). Basic queueing theory: Foundations of system performance modeling. GlobeEdit.
Vilaplana, J., Solsona, F., Teixidó, I., Mateo, J., Abella, F., and Rius, J. (2014). A queuing theory model for cloud computing. *The Journal of Supercomputing, 69*(1):492–507.
Welch, G., Bishop, G., et al. (1995). An introduction to the Kalman filter. Technical report, University of North Carolina at Chapel Hill.