CloudCats

Rory Smith
University of Bristol - Department of Computer Science
Email: rs14369@my.bristol.ac.uk

I. Introduction

The New York Times described cat images as ”that essential building block of the Internet” [1]. In 2015, more than 2 million cat videos existed on YouTube, with an average of 12,000 views each [2] - a higher average than any other category of YouTube content.

Blatantly, the internet enjoys cats, but with 1 billion hours of video watched on YouTube per day [3], there is still plenty of scope for more cat content. Videos of humans currently monopolise the internet's video sites, but Cloud Cats aims to solve that, by converting these to cat videos in near real-time.

After replacing *youtube.com* with *cloudcats.cloud* in a YouTube video's URL, users can stream the alternate version of their chosen video, in which human faces are replaced by their feline counterparts. By making use of the MPEG-dash streaming protocol, playback can begin before the entire video is processed, and given enough compute power, in real-time.

For an example of the system in practice, go to https://cloudcats.cloud using Google Chrome [1].

Fig. 1. *The Pussycat Dolls.*

II. Design

A. Cloud Provider

Cloud providers often offer PaaS (Platform as a Service) solutions that are sold as reducing the time needed to develop and maintain parts of a system. AWS (Amazon Web Services) have many of these PaaS such as S3 object storage and elastic beanstalk which aids application orchestration. While reduced development time is a valid benefit, these have the side-effect of locking users in to the cloud provider. If the need arose to switch to a different provider, significant resources would be required to rewrite the application for another provider's PaaS.

Cloud providers typically sell IaaS (Infrastructure as a Service) too, whereby customers can rent virtual or physical machines, to which they have comprehensive access. This provides a much higher level of control for users, with the drawback that more time is required to setup and maintain individual components.

For example, Oracle's object storage PaaS automatically grows its disk space and distributes requests to ensure high availability. Setting up an object storage server in an IaaS context however, may require manually solving those tasks on an individual basis.

One of the core goals of CloudCats, was to build a system capable of being deployed to any cloud provider. Thus, the system is heavily containerised, with minimal provider-specific dependencies. As a consequence, when looking for an appropriate cloud provider for this project, we examined IaaS offerings rather than PaaS. On this basis, we concluded that using OCI (Oracle Cloud Infrastructure) would meet our requirements the best, as we could apply a student discount for free compute credit and its offerings did not seem significantly less than more established competitors [2]. Although other providers such as AWS and Google provide free offerings, we had already exhausted these in the past.

B. Video Transcoding

Another key goal of CloudCats was to perform real-time rendering and streaming. For this, it was necessary to either render and serve the entire video in milliseconds or render the video in real-time and stream it to the client as frames were completed.

After investigating different video formats, we found that a new format known as MPEG-Dash could allow streaming of chunks as they were produced. A codec like MPEG4 on the other hand, cannot be streamed as it is rendered, because of header dependencies in the video container. We quickly decided to stream using MPEG-Dash, rather than perform a complete render in milliseconds, due to the lower computational requirements.

To manipulate individual video frames, we knew that we would need to distribute them across numerous workers to obtain a real-time bandwidth. To do this, we do not wait for the entire video to download, rather, we use UNIX pipes

[1]Due to limitations of the underlying video container.

[2]We would later find this difference to be far larger than initially estimated.

to redirect the download stream from YouTube into a media utility called *ffmpeg*. As the video downloads, ffmpeg outputs frames on stdout, which we upload to object storage and post worker jobs for.

In the mean-time, the video's audio stream is downloaded separately and shaped into 2 second chunks in the MPEG-Dash format. These are then written to object storage. The time it takes to process the audio stream is negligible in comparison to the video process, and so we will spend our time examining the video process for the most part.

Frame workers take the frame jobs and apply face detection to replace human faces. For this, we use an open-source framework called OpenFace [4]. We don't go into specifics about the face detection due to this paper's focus on the cloud aspect. The accuracy of the face detection suffers a little because of this, but is still smile-inducing.

The individual frame workers receive jobs in frame chunks, to amortise the cost of data transferal and job communications. This means a worker may process 1 to n many frames per job. Finding n is a difficult task, as a value too low will result in IO-dominated work, whereas a value too high would increase the rendering latency as the whole chunk must be processed before any frames can be used in the final video. Processed frames are then placed on object storage and a frame chunk job finish event emitted. We found through empirical trials, detailed in fig 2, a value around 5 to provide the best balance between time-per-frame and chunk time.

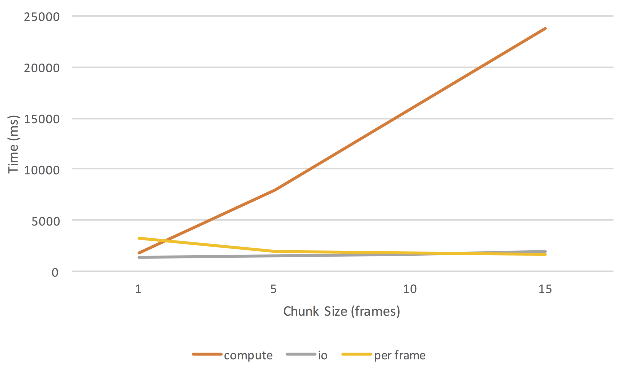


Fig. 2. Time taken for a frame worker to process a single frame chunk across different frame chunk sizes.

The final video transcoder component awaits these frame chunk job finish events, and constructs the final video from the processed frames. It must maintain correct frame order, as frame workers do not necessarily finish in the order they were started. If this is not corrected for, then a video may appear to skip back and forward frames. We must also ensure that we glue frames back together at the same framerate as the original video played. If we choose another framerate, the audio stream will not end at the same time as the transcoded video. The processed frames are then piped into another ffmpeg instance which periodically outputs MPEG-Dash video chunks which we place in object storage.

Unfortunately, due to our use of MPEG-Dash, and current browser compatibility, CloudCats must be browsed using Google Chrome. Luckily, Google Chrome currently enjoys a 58.83% market share [5]. Future work may add additional media containers to allow for multiple browsers by supporting many concurrent data streams, which MPEG-Dash natively supports.

III. Implementation

To help containerise individual components and reduce any dangling dependencies, each component exists as its own Docker image. This way, we were able to run the system in its distributed form locally, during development, and then easily deploy the containers later. Docker also maintains a public registry of images available to the public. These are maintained by the Docker community and allow anyone to use the images as the starting point for their own images. This meant that CloudCats can take advantage of pre-built images to bypass complex setup dependencies and compilation times. There are also security disadvantages to this, as discussed in section III-I

A few components have shared library dependencies. For example, video/audio download and video transcoder all require the ffmpeg library. To manage this, these images are built on top of our own custom pre-compiled image with the required ffmpeg dependencies. This reduces the time needed to compile each component's own image as Docker will only spend time compiling the *extra* component-specific steps.

All components interact with datastores and a message queue. Thus, we maintain shared coding interfaces for these that hide the implementation details from components. In this way, we can easily iterate different implementations of, say, a message queue, without requiring the refactoring of components.

The architecture of CloudCats is defined visually in fig 8.

A. WebApp

The front-end of CloudCats is served using a NodeJS framework known as ExpressJS. This software stack is desirable because as of December 2017 [6], the NodeJS module repository NPM contained more than 500,000 modules. Combined with ExpressJS, this provides the needed libraries to carry out much of our required functionality. For example, vertical scaling can be achieved with ExpressJS and the *cluster* module [7]. The horizontal scaling of instances is discussed later in section III-H. We also make use of socket.io to maintain websocket connections with individual clients. We use this to communicate status updates in the video's rendering process with the client and maintain a healthy user experience.

The webapp also acts as the entrypoint for video render requests. It must start a rendering job when no such job exists and make sure not to start a video job if the video is already being processed. To communicate jobs, the webapp communicates with a standard messaging queue, using AMQP (Advanced Message Queuing Protocol). This provides a generic queuing interface so we have the potential to use many different queue providers.

The same queuing protocol is used to listen for updates throughout the render process. By subscribing to the appropriate queue, the webapp can listen to updates published by other parts of the system. As this uses a pub/sub system, many webapp instances are able to subscribe and consume the same message from a queue and so many client browsers can wait for the same video to complete.

The video stream itself is rendered with the help of DashJS [8], which takes MPEG-Dash stream segments, and manages the video state. This lets us focus on the server-side rather than client-side implemenation details. On top of this, we also use Twitter Bootstrap to accelerate development of the visual elements on CloudCats. Resource dependencies such as Bootstrap and DashJS are loaded through CDNs (Content Distribution Networks) to reduce load on our servers and decrease total page load time.

To prevent running multiple jobs for the same video simultaneously, the webapp communicates with a Redis database. Redis stores key-value mappings in memory, and is used by many for caching due to its low latency. We simply store values on a per video basis, to form a semaphore-like system with low latency. Due to the sparse amount of data stored in the database, we use a single Redis server, but Redis can easily be clustered by sharding a database across multiple Redis instances, which we could autoscale according to some metric such as available memory. As the Redis database only contains records that try to prevent wasted concurrent computation, it is not strictly vital to the running of CloudCats [3]. Thus, we do not implement any data persistence techniques.

The webapp must also correctly sanitise user input. Due to the error-resilient nature of the architecture, an invalid video ID would end up being processed in perpetuity as the system fails to find a video matching the required and ID and retries later. While catching this sort of error at a component level for every component is a viable solution, it is considerably more work than what we do: performing a check at the point at which a user inputs the video ID.

B. Audio/Video Download

When a video job is requested, two different types of NodeJS workers will accept the job. Due to the way that YouTube delivers its content, audio and video streams are commonly downloaded separately. Thus, we use two different workers. Both pipe their download stream into an ffmpeg instance, but the audio downloader outputs the final MPEG-Dash audio chunks. The video downloader, instead produces individual image frames which are bundled into chunks. Both downloaders upload output to object storage, although video frame bundles are stored in a private bucket as they are only needed internally. Restricting access to the bucket like this may not seem necessary at first glance but is good practice as following a system of least privilege, we can minimise the consequences of any unauthorised, malicious access.

[3]Internal APIs to the Redis database implement soft-fail techniques so that if the database is unavailable, the requesting application can continue cleanly as if the intended record did not exist.

C. Frame Worker

Frame jobs are accepted by a large number of frame workers. These are written in Python, as OpenFace (the face detection framework) only includes Python bindings. Frame workers still interact with object storage and the AMQP queue. After faces are recognised, we apply landmark detection to find the degree to which the face's mouth is agape. Then we can then place a cat face in correct location and scale, as well as open its mouth to mirror the human's. Finished frames are uploaded to internal object storage and jobs placed on the queue to indicate finish.

OpenFace comes with built-in support for GPU computation, which dramatically improves performance. During development, we were never able to get hold of a GPU to test this locally or indeed on the Oracle cloud, but the author of OpenFace, Brandon Amos, claims using a GPU can cut inference time in half, compared to an 8-core CPU [9]. The OCPUs (Oracle CPUs) we use in practice are roughly equivalent to about 0.5 CPU after virtualisation. Thus, gaining direct access to an entire GPU would reduce times from ~ 5s per frame chunk to ~ 156ms.

Thus, to achieve real-time rendering using virtual CPUs, we require large numbers of instances. To this end, we exhaust our CPU quota to bring real-time speed on a single video at a time. While multiple videos can be rendered simultaneously, neither video will process in real-time. As discussed in the last paragraph, this would easily be alleviated via horizontal scaling given a few GPUs or simply more CPU instances.

D. Video Transcoder

The finished frame bundles are finally collated by a transcoder, whose job it is to order the frames correctly and output the MPEG-Dash video chunks. These chunks are then uploaded to public object storage, in the same bucket as the audio chunks. After the first chunk is produced, we also produce an MPEG-Dash manifest which describes the video, and publish a message to indicate that the video rendering has started, which is picked up by the webapp and communicated to the client browser via websockets.

E. Object Storage

To implement object storage, we use an opensource object storage server called Minio [10]. Minio can be setup in distributed mode which allows pooling of multiple drives into a single object storage server. This also increases availability as Minio maintains data integrity so long as half or more of the designated disks are online.

As we had abstracted the object storage implementation from the rest of our components we were able to experiment with using Oracle's object storage API, but found UL/DL speeds to be too slow for real-time video. We experienced 4 second upload durations for single 22Kb frames and 40 seconds for 10 frames, showing no amortisation through batching requests. As a consequence, we use our own distributed Minio cluster, which performs much faster. This also aligns well with our core goal, which is to reduce our dependencies on provider-specific platforms.

F. Message Queuing

As with object storage, Oracle provides a Messaging Cloud Service for customers to use. Rather than spend time developing a custom client to interface with their message queue, we chose to run our own using a mature implementation of AMQP called RabbitMQ [11].

Like Minio, RabbitMQ also has a distributed mode. We setup RabbitMQ in clustered distributed mode to connect multiple machines to form a single message broker. Doing so allows us to increase availability and message throughput, which is important when we want to achieve a high rendering framerate.

Choosing a message queue, rather than another communication method such as database polling or raw socket connections, provides us a highly abstracted communication channel, with a low overhead. RabbitMQ scales very well with thousands of messages, as CloudCats is likely to produce [12]. By disabling RabbitMQ's persistent disk storage of messages, we can dramatically improve the message throughput by about a factor of 5 [13]. The decoupled nature also means that we are able to implement parts of the system in different languages without having to worry about language or framework-specific communication interfaces. This let us write a system written predominantly in NodeJS, with a single Python component that still communicates flawlessly.

An important feature of using AMQP with RabbitMQ is the ability to acknowledge messages. By only acknowledging messages when a process has successfully executed, we are able to harden our infrastructure against instance failures or code crashes. In the case that one of our components breaks after consuming a message to download a video, for instance, another video download instance will start the job after some timeout because the original message was never acknowledged.

Crashed processes can be detected quickly by a broken connection to bypass a timeout, but sometimes network conditions can cause nodes to become isolated. To help protect against adverse network issues, we use heartbeats to detect broken connections. By sending pings back and forth every n seconds, a missing ping can be used as a signal of bad health. This helps us recover from failures much faster than otherwise. From various experiments, we found a heartbeat interval of 20s gave us the fastest recovery time without producing too much chatter on the network.

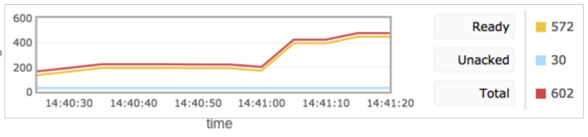
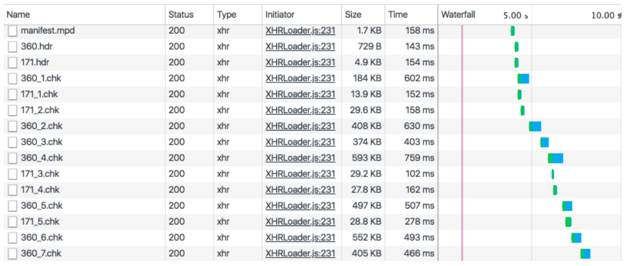


Fig. 3. A graph for the frame jobs queue, showing an increasing backlog of frame jobs, when they should remain constant over time. This was the result of an insufficient number of frame workers.

RabbitMQ's online management console also proved helpful when finding bottlenecks in the pipeline. By examining the publish and acknowledgement rates of different queues, it was easy to find components that were not processing data fast enough, like the frame workers in fig 3.

G. Reverse Proxying

As a design decision, we did not want to publicly expose Minio bucket URLs. In regards to security, we deemed it useful to hide our internal architecture, and we simply did not want to expose Minio's relatively verbose URLs. Instead, we opted to reverse-proxy object storage requests through our webapp, which allowed us to map a simpler URL scheme to Minio's. Although Minio's object storage server provided fast responses, we found that reverse-proxying download requests through our webapp introduced significant delays. In an attempt to mitigate these slow requests through the ExpressJS webapp, we introduced an Nginx reverse-proxy. This way, we are able to filter off all object storage requests straight to Minio using a more mature platform designed specifically for the task. We can also introduce response caching as this is built in to Nginx at its foundation.



Name	Status	Type	Initiator	Size	Time	Waterfall 5.00 s 10.00 s
manifest.mpd	200	xhr	XHRLoader.js:231	1.7 KB	158 ms	
360.hdr	200	xhr	XHRLoader.js:231	729 B	143 ms	
171.hdr	200	xhr	XHRLoader.js:231	4.9 KB	154 ms	
360_1.chk	200	xhr	XHRLoader.js:231	184 KB	602 ms	
171_1.chk	200	xhr	XHRLoader.js:231	13.9 KB	152 ms	
171_2.chk	200	xhr	XHRLoader.js:231	29.6 KB	158 ms	
360_2.chk	200	xhr	XHRLoader.js:231	408 KB	630 ms	
360_3.chk	200	xhr	XHRLoader.js:231	374 KB	403 ms	
360_4.chk	200	xhr	XHRLoader.js:231	593 KB	759 ms	
171_3.chk	200	xhr	XHRLoader.js:231	29.2 KB	102 ms	
171_4.chk	200	xhr	XHRLoader.js:231	27.8 KB	162 ms	
360_5.chk	200	xhr	XHRLoader.js:231	497 KB	507 ms	
171_5.chk	200	xhr	XHRLoader.js:231	28.8 KB	278 ms	
360_6.chk	200	xhr	XHRLoader.js:231	552 KB	493 ms	
360_7.chk	200	xhr	XHRLoader.js:231	405 KB	466 ms	

Fig. 4. The Google Chrome inspector pane shows the faster request responses after adding an Nginx reverse-proxy. Prior to this, it was common for response times to exceed 5 seconds.

Examining Google Chrome's networking inspector pane in fig 4 allowed us to see the improvements gained from this approach. By combining this with relocating our datacentre closer to the UK, we were able to reduce some requests from 15 seconds to 1 second. We also store static Javascript and CSS content on Nginx rather than the ExpressJS webapp to help speed up initial page loading and reduce server load were CloudCats to experience higher traffic.

H. Orchestration

Up until this point, we have only discussed individual components, each residing in its own container. During development these could be run together using Docker compose but to run these on the cloud though, requires the management of nodes and distribution of these containers across nodes.

To carry this out, we run Kubernetes, which can automate the deployment, scaling and management of containerised applications [14]. We setup a master Kubernetes node, whose job it is to manage work across subsequent worker nodes. Kubernetes takes descriptions of a system as YAML files called deployments. These take the forms of specifications that describe different components, such as the transcoder or web app in the case of CloudCats. Along with details on how to start a component such as the Docker image, these specifications also detail the number of instances that should be running. Kubernetes calls these *pods*, where a single node can run many pods. We find it is easiest to compare Docker containers to Kubernetes pods, with the exception that pods are also networked into the Docker networking system, detailed below.

Alternatives to Kubernetes exist, such as Docker Swarm [15]. While these are mature in their own right, and may arguably fit with our other design choices (such as using Docker Swarm with our Docker images), Kubernetes appeared to have the strongest online community and a large selection of documentation. In retrospect, this worked well, as online discussions helped considerably during development of CloudCats.

To network together the containers across nodes, we use an overlay network called Calico [4]. Calico helps direct communications between containers using a private subnet separate from the one the nodes are networked upon.

Along with the Kubernetes DNS, this allows us to apply service discovery in our system using a key-value etcd datastore pod. Using this technique, Kubernetes resolves custom names such as *obj-store-master* and *messaging-master* to the correct internal address of the service. Beyond name resolution, service discovery also acts as an internal load-balancer, routing requests for names across all available pods of that type. This vastly reduces the complexity of manually networking together the containers.

Kubernetes performs periodic health checks upon nodes to ensure availability. If it finds an unresponsive node, it will redistribute work to other nodes to maintain the number of requested instances specified in deployment configuration files per instance type. By using a system to automatically verify node health, we can minimise downtime significantly in comparison to manually verifying nodes.

We make use of the public Docker Hub to host our instance images [16], which we can reference in our deployment configurations for Kubernetes to build instances from.

Although Kubernetes provides support for auto-scaling nodes up and down based on the current workload, this requires interfacing with the cloud-provider's API to do so. Kubernetes cloud provider interfaces exist for many popular providers such as AWS and Google but lacks one for OCI. Although OCI includes an API to add/remove compute instances, it was outside the scope of this project to write a plugin for Kubernetes and so auto-scaling remains unimplemented. Our choice of Kubernetes though, means that auto-scaling works out-of-the-box on other platforms and would work on Oracle easily, given an applicable interface.

To reduce manual work with individual instances, we created bootstrapping scripts to prepare new instances for Kubernetes [5], along with deployment scripts that perform the commands required to update existing Kubernetes pods from the master node.

[4]We used Flannel for most of the development until it became clear that it was causing serious network disruptions at seemingly random intervals. Perhaps it did not mesh well with the Oracle image's networking configuration.

[5]Tasks took the form of installing library dependencies and uploading authorization tokens.

I. Networking

CloudCats makes use of the OCI [17] networking facilities. Instances are run within a VCN (Virtual Cloud Network) with security lists that limit ingress traffic to specified ports. Doing so protects our public instances from malicious attacks on ports that the public does not need access to. On top of security lists that essentially act as network firewalls, we also implement firewalls locally on a per-instance basis. At the low level this is accomplished using iptables, which Docker also uses to route data. We spent considerable time ensuring that our own iptables security rules meshed well with those of Docker and did not disrupt traffic.

While on the topic of security, it is notable that we based our own images on top of those in the publicly available Docker registry. This opens us up to potentially malicious activity running inside our own images, if we are not careful about our image choices. We were conscious of this threat, and so actively checked the dependency chain of images we use.

Although using HTTPS instead of HTTP may seem like overkill, we found that it was necessary to induce a clean user experience. As YouTube serves all of its content under HTTPS, simply replacing *www.youtube.com* with *cloudcats.cloud* would fail if we did not support HTTPS. To this end, we generated a certificate using Let's Encrypt [18], and added a HTTPS listener to our load balancer along with the certificate. This way, the Nginx instance sitting behind the load balancer need not implement SSL, and it remains completely oblivious to the entire process.

We place separate subnets across distinct availability zones to help protect against system failures. By including a public load-balancer and performing health checks, we can easily route traffic to a healthy instance of CloudCats. The load-balancer accepts different routing rules; a round-robin strategy attempts to allocate requests across all internal instances equally, whereas IP hashing tries to route the same client to the same internal instance across subsequent requests. As maintaining distributed state is not typically a fast operation, and our system requires low latency, we end up storing user state in small regions before propagating later. Due to this, we use IP hashing so that data computed for a user in one availability zone can still be accessed by that user in subsequent requests. Later, this data is propagated to other zones and made available as a pre-computed resource. As we use a single public load-balancing entrypoint, we are able to map our cloudcats.cloud domain straight to the load-balancer's public IP address. This means that even if internally, IP addresses change, the public address can remain constant.

IV. Analysis

A. Scaling

Due to CloudCats' orchestration using Kubernetes, autoscaling can trivially be implemented. Instead of defining a deployment in which we define a static number of instances, we can make the number of instances a function of some other metric. It is common to use CPU utilisation for this, but doing so may be problematic for our use case. Since the number of available frame jobs can vary rapidly, so can the CPU utilisation, and so autoscaling may rapidly up and downscale. Instead, we can use a custom metric, such as the number of currently processing videos, to steer our autoscaling at a more controlled rate.

Combined with our use of message queues, CloudCats can then scale horizontally with ease, as the jobs will be split across all available nodes. As well as the ability to horizontally scale, Kubernetes also gives us the power to scale up individual components, rather than the entire system as a whole. This means that the number of frame workers may scale rapidly for a single high-framerate video, while the number of other component instances remains low, which helps us to keep our running costs as low as possible.

Although autoscaling is unimplemented for OCI, we are still able to provide examples of how the system scales manually. Fig 5 shows that as we add more OCPUs to the system, it can process a single video at a faster rate. The components that support the workers, such as the RabbitMQ messaging queue and the Minio object storage, cope with the increased throughput, indicating that the bottleneck lies with the available processing power. The same experiment can be repeated for two videos, requested in parallel. In this case, results are almost identical to that of the first experiment, only with constantly higher processing times [6]

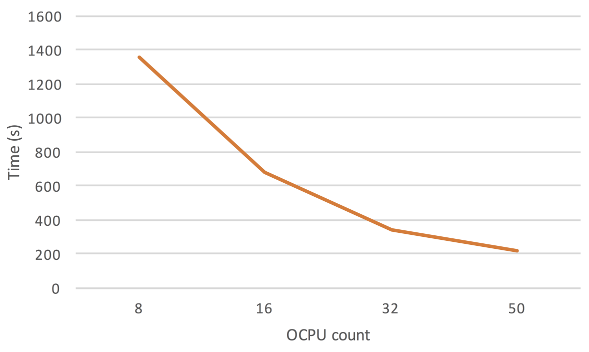


Fig. 5. Time taken to process a single video (of duration 206s) by the system when using different numbers of OCPUs.

As we are able to manually increase the number of clustered RabbitMQ instances successfully and there is significant evidence [13] that the platform can scale under such a setup, it seems fair to argue that the messaging component can scale [7]. As with the processing pipeline, this could be configured to autoscale, based on some metric such as incoming packet count.

In regards to video properties, many do not affect our ability to process in real-time. The only property that does affect us is the video framerate. As is visible in fig 6, we are able to achieve a framerate that hovers around 25fps. Thus, when processing video filmed at 30fps (or higher), CloudCats struggles to process in real-time.

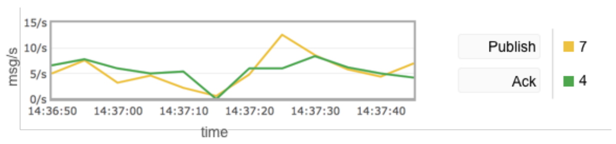


Fig. 6. Throughput of frame jobs showing an average 5/s rate. This was recorded with a frame chunk size of 5 and so equates to 25fps.

Although the load balancer points to a single port when routing traffic to an availability zone, our use of Kubernetes services means that this traffic is then distributed to as many webapp instances as are available. Due to this setup, we can safely increase the number of webapp instances to scale horizontally, allowing Kubernetes to perform service discovery and route to the instances.

Kubernetes efficiently deals with horizontal scaling across many nodes, but it also manages to successfully vertically scale on single nodes. As mentioned previously, Kubernetes spawns a series of pods per node, which each run their own Docker container. In practice, this means we are able to run an ensemble of different machines, with differing hardware, and Kubernetes will distribute pods across these in a manner that reflects the hardware available on each. Consequently, when running a virtual machine instance with 4 OCPUs and a bare metal instance with 36 OCPUs, we find that of our 46 pods, 40 are placed on the bare metal instance. This gives us great power by removing the need to use identical nodes and upgrade single nodes if we wish.

Even though we do not use it, Kubernetes also supports node labels. Combined with GPU nodes, this would let us tell Kubernetes to place certain deployments (such as frame workers) on GPU nodes to make use of the node-specific hardware. We experimented with using labels to define *pipeline* and *supporting infrastructure* nodes so that we could ensure the supporting infrastructure always had sufficient compute power available. Doing this had no positive effect on performance though, and as it required extra effort, we decided not to carry it forward.

B. Provider Independence

One of the goals of CloudCats was to be deployable on any cloud IaaS provider. To measure the success of this, we ran a small scale test on AWS EC2. The automatic node setup scripts which we created to ease deployment of nodes worked flawlessly to configure each instance. We found the

[6]Due to the increased load.

[7]The same argument can be made for the Minio object-storage component.

majority of our time was spent understanding and configuring the networking between the instances. After networking the instances together, CloudCats worked just as well as on OCI, albeit at a lower relative performance due to the fewer instances we deployed (to avoid high costs).

C. Current Limitations

Unfortunately, although CloudCats does perform well, achieving real-time rendering and streaming, it has been difficult to achieve auto-scalibility using Oracle cloud. As mentioned in previous sections, Kubernetes lacks an interface for the OCI API which would allow automatic node scaling operations based on resource demands. Further, the Oracle student discount limits the number of available CPU instances at a given time to 6 [8]. What this means in practice is that rendering of videos can sometimes fall behind real-time.

Another restriction of Oracle's student discount is the inability to use their GPU instances. By using a GPU rather than a node's CPU, the frame workers would be able to achieve a substantially higher throughput, reducing the number of required workers, while also increasing the rendering framerate. After email enquiries with our contact at Oracle, it became clear we would not gain access.

Before beginning video processing, we need to collect video metadata for information such as video framerate. This ends up manifesting itself as an extra sequential step in our pipeline, adding to overall loading times. Fig 7 tries to visualise this and highlight the impact this metadata loading has on initial video load times. Consequently, the initial loading for some videos can take several seconds, leading to a less than satisfactory user experience.

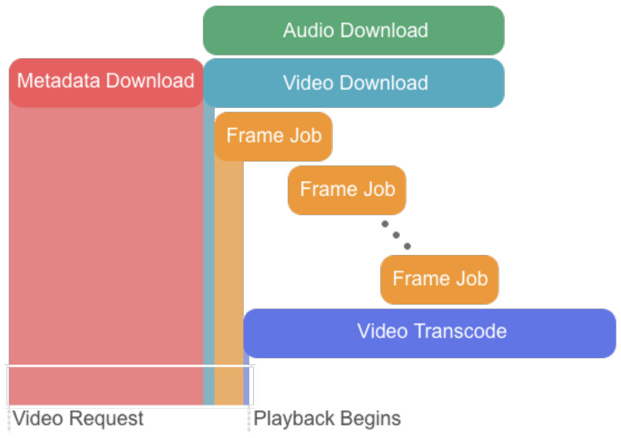


Fig. 7. The video processing pipeline can run components in parallel, but the majority of initial load time is a consequence of sequential metadata loading.

D. Future Work

Given more processing power, future work may involve the splitting of stream downloading. Too often the bottleneck of the pipeline is the rate at which video frames are dispatched as jobs. If we were to split the process across multiple download instances we should be able to achieve a higher processing throughput.

[8]Although a bug in Oracle's console allowed us to increase this to 50.

The method with which video frame chunks are distributed may not be the most efficient. Perhaps if frame chunks were sent directly to available workers, we could mitigate communication latency. During experiments, we found that the frame workers were CPU bound, rather than IO bound, so this is most likely a fine optimisation technique.

Rather than approaching CloudCats from an IaaS viewpoint, a Faas (Function as a Service) design may have proved more beneficial in terms of running costs and responsiveness. In this manner, each component could be re-written to run only when required, rather than constantly polling for jobs. Unfortunately, this service isn't provided by Oracle as of today, but others such as Google App Engine and AWS Lambda are conceivable choices.

V. Conclusion

CloudCats set out to change the online video landscape for the better, which we think it has. Users can now view their favourite YouTube videos in a new cat-like light without waiting nine lifetimes for the video to render. CloudCats message passing foundation and modular design means the system is highly resilient to internal failures and potential catastrophes.

The choice of cloud provider and Kubernetes means that some functionality from Kubernetes such as auto-scaling and internal load balancing are unavailable. As a silver lining, though, since the support exists in Kubernetes for other cloud providers and CloudCats is provider-agnostic, it is straightforward to fully set up using another provider.

References

[1] The New York Times. ”What the Internet Can See From Your Cat Pictures”. https://www.nytimes.com/2014/07/23/upshot/what-the-internet-can-see-from-your-cat-pictures.html.
[2] Science of Us. ”So Heres a Study About Internet Cats”. http://nymag.com/scienceofus/2015/06/heres-a-study-about-internet-cats.html.
[3] YouTube. ”You know whats cool? A billion hours”. https://youtube.googleblog.com/2017/02/you-know-whats-cool-billion-hours.html.
[4] OpenFace facial recognition library. https://github.com/cmusatyalab/openface.
[5] Browser Market Share. https://www.netmarketshare.com, 2017.
[6] Module Counts. http://www.modulecounts.com/.
[7] Monitoring & Vertically Scaling Node.js Applications. https://www.netguru.co/codestories/monitoring-vertically-scaling-nodejs-applications.
[8] DashJS. https://github.com/Dash-Industry-Forum/dash.js.
[9] OpenFace 0.2.0: Higher accuracy and halved execution time. http://bamos.github.io/2016/01/19/openface-0.2.0/.
[10] Minio. https://www.minio.io.
[11] RabbitMQ. rabbitmq.com.
[12] RabbitMQ Blog - Performance. http://www.rabbitmq.com/blog/tag/performance/.
[13] RabbitMQ Performance Measurements, part 2. https://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/.
[14] Kubernetes. https://kubernetes.io.
[15] Docker Swarm. https://github.com/docker/swarm.
[16] Docker Hub - Roarster. https://hub.docker.com/u/roarster/.
[17] Oracle Cloud Infrastructure. https://cloud.oracle.com/cloud-infrastructure.
[18] Let's Encrypt. https://letsencrypt.org.

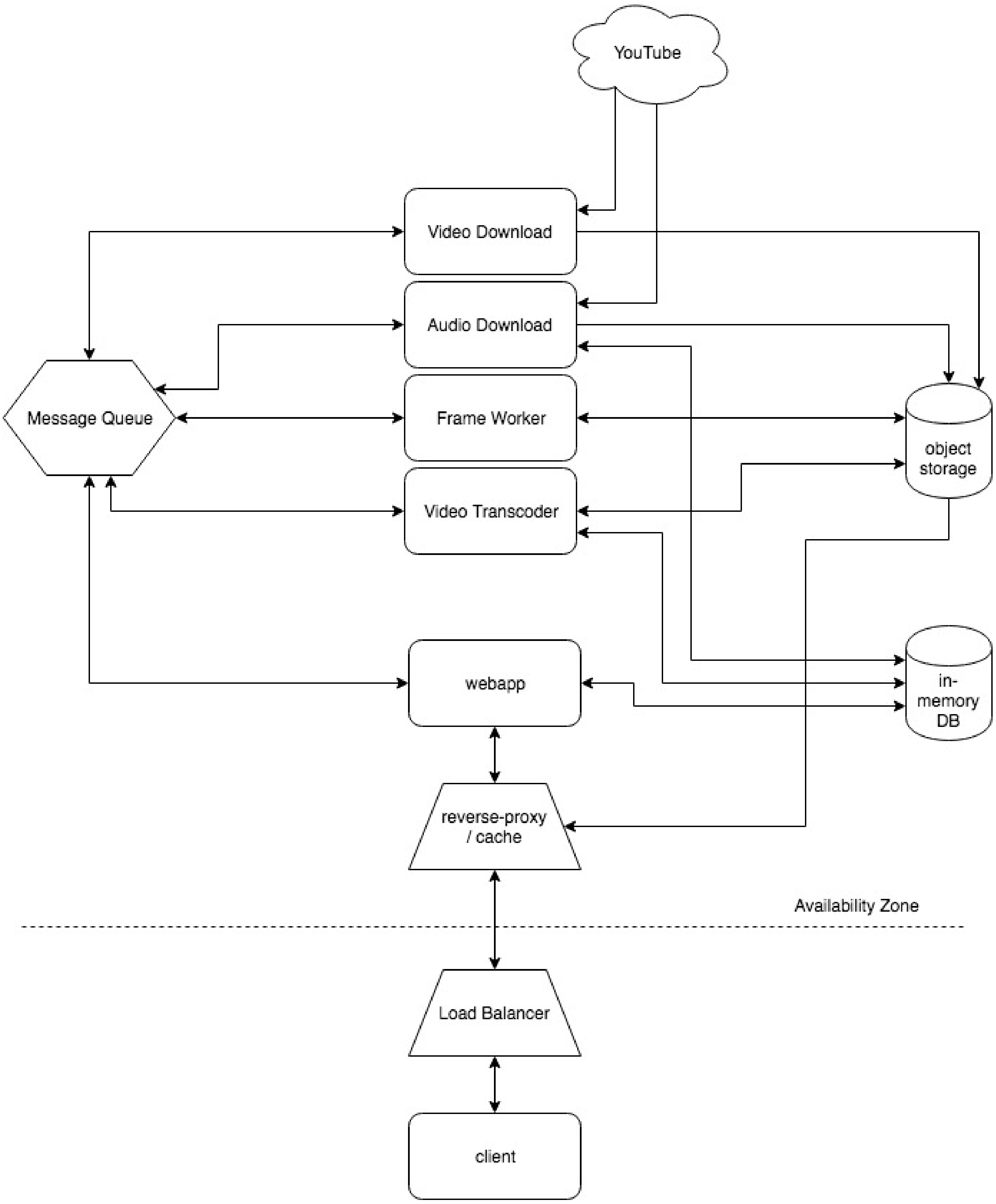


Fig. 8. Architecture of CloudCats.