# Chiron: Stream the coding

**Thasphon Chuenchujit** UIUC `chuench1`

**Baskar Rethinasabapathi** UIUC `rethina2`

**Vaijayanth Raghavan** UIUC `vraghvn2`

## 1 Abstract

Programming is a very competitive skill to have in the current job market, and as a result there are many online resources that offer training in programming and software engineering-related fields. These resources vary in form and functions, ranging from universities' lecture-driven online classes to project-driven classes offered by various other sources that cater to people who never wrote a single line of code as well as advanced programmers looking to expand their skill set. To supplement this, recently there has been a new trend in content livestreaming, where programmers stream their coding sessions to interested audience. These livestreams, however, are hosted by experienced programmers and are mostly intended for beginners, who want to watch real projects getting built from scratch. However, we think that in order for the beginners to get the most out of this education model, they too should be streaming. With that mindset, we present Chiron, a collaborative live-streaming platform with the ultimate goal to help beginners built projects with interactive code review.

## 2 Introduction

Following the success of Twitch.tv, a website that offers live-streaming services for gamers to stream their gameplay to users, "watching people code" could be the next big thing in online streaming community. Already, there are several websites such as *watchpeoplecode.com* and *livecoding.tv* that promote live coding session lead by experienced programmers coding out projects in their spare time. Their main goal is to allows programming enthusiasts to see coding of real-time projects in action from the perspective of the programmer, as many of the projects are built from scratch. While we agree that watching other people code is a great next step in the learning process, we also notice that (as in twitch) the streamers are either professionals or experienced users in their field and the viewers are mainly the inexperienced looking to learn from them. After further investigations, we find that the more beneficial model for the users who wish to learn more about programming should be the other way around.

Livecoding.tv seems to be the equivalent of Twitch in this field in terms of functionality and business model: providing streamer with monetary incentive to broadcast contents to viewers, who mainly just consume the content. We believe that there is room for improvement in this business model when applied to live programming: streamers can also benefit from the viewers. CHIRON is motivated by this insight, and seek to create a platform that encourage programmers of any skill level to broadcast their work early on by offering reputation incentive to experienced viewer for peer code review and mentorship. This allows both experts and learners to take part in streaming as well as consuming the content.

From the survey of literatures in the field of computer science teaching, we find that learning to program with an interactive peer review process appears to yield the best results for most students. Many novice programmers recognize that adhering to coding standard is important, yet they fail to comply with such standards in practice. Another study finds that novice programmers tend to be more motivated and perform better when they are exposed to live code review process, as they can think out loud and interact with another person. Moreover, with live code review, the novice programer can bypass the frustration of constantly searching for answers to simple obstacles, resulting in a more focused study of programming standard and practices. With this, we conclude that in a streamer-viewer setup, it will be more beneficial for the novice programmer to be streaming and the experienced programmer to watch and review.

We introduce Chiron, a livestreaming platform designed for code streaming and interactive peer-review. Chiron has four main functionalities: video live streaming, interactive chat, persistence code storage, and users' rating system. Each component will be described in details in the following sections.

## 3 Video Streaming

The video streaming portion of the Chiron platform can be broken down into three main parts:

- a client broadcasting software for sending video stream to the server
- a server that accepts incoming video stream and forwards them to viewers' client and
- a front-end client for video stream viewing

In the sections below, we go over available broadcasting software choices, describe our stream server, and discuss possible clients for video viewing.

### 3.1 Broadcasting Software

There are many free and paid software in the market today that support live video streaming from a client using the Real Time Messaging Protocol (RTMP). Based on information from similar service platform such as Twitch.tv and livecoding.tv, we found that the broadcasters on both platform recommend Open Broadcaster Software (OBS) as a broadcasting software of choice since the software is free, open-source, and under active development by its community. Other broadcasting software suggested by both services are xsplit, ffsplit, and wirecast. As OBS is the only open-source option, we use it as the broadcasting client during development.

### 3.2 RTMP Server

The server is implemented with nginx using an RTMP module add-on. RTMP is a TCP-based protocol which maintains persistent connections and allows low-latency communication between the two endpoints. Initially developed by Macromedia (now owned by Adobe) as a protocol for playing flash video, RTMP has become one of a popular protocols for video streaming as it has a very small overhead and provides multicast support for one-to-many streaming. Although there may be availability issue as RTMP uses port 1935, which may be blocked by tight firewalls, the stream can be encapsulated in HTTP with a protocol name RTMPT. RTMP stream can also be encrypted with TLS/SSL for added security in case of private stream. Using RTMP module on nginx allows for ease of deployment for RTMP server with the scalability of nginx. The RTMP server is very light, and the bottleneck in the system is in the network bandwidth as the server essentially serves as a router that routes the video stream from the broadcasters to the viewers. In order for the server to scale with the number of broadcasters and viewers, a cluster of nginx-rtmp server can easily be deployed as the rtmp module has a built-in relay functionality that allows a server to continuously pull a video stream from another server so that it will be available to serve locally. Additionally, there is a functionality that allows a server to split the stream and push it to another server to increase availability. We are currently exploring possible topological setup of the cluster and best dynamic internal routing scheme of the cluster, so we do not have any experimental data to report at this time.

### 3.3 Viewing Client

Since RTMP is originally intended for delivering flash videos, the viewing client has to support flash in order to view the video. Looking at similar platform, we found that Twitch.tv uses flash player and livecoding.tv uses JW-Player11, which is an enterprise solution video player for browser. Using enterprise solution such as JWPlayer would ease development at the cost of licensing fee for the player, so instead we use videojs, an open source HTML5 video player, as the main video player on our platform. videojs supports rtmp stream playback using HTML5 on supported browser with a flash player fallback on unsupported browser, making it the ideal player for our project.

## 4 Live Chat

One important objective of the website is to provide viewers with the functionality to interact with the broadcaster. On other platforms with similar service, this is implemented as a comment thread or a live chat box which allows registered users to send post message to the broadcaster. Traditionally, such system would be implemented with a database and a server, and each message sent to the server would be inserted into the database. On the other side, each instance of the chat client (each browser tab) would periodically poll the server for new messages to be displayed in its chat windows. This traditional approach does not scale easily, as the chat client has to constantly poll the server for updates even though the updates may be very sparse. There is also a delay between the time the message is sent and when it will be poll, which entirely depends on the refresh rate of the browser client. This causes tradeoff between message delay and frequency of requests being sent to the server. Additionally, this approach incurs a storage overhead for the messages as the database has to store the messages until it is safe to be deleted (e.g. when all chat clients polled the server). Since this traditional solution does not scale well, Chiron's chat system is implemented using Socket.IO. In comparison to similar platform, Twitch.tv does not store chat messages and a user viewing a stream can only see the messages that are sent after they loaded the page, which is similar to our model. Livecoding.tv, on the other hand, seems to store the messages for one broadcasting session.

Socket.IO is a javascript library that enables real-time bidirectional event-based communication between clients and server. Socket.IO is built on top of websocket, which provides full-duplex communication channel over a single TCP connection. SocketIO server create a websocket for bi-directional communication with each client, so when a client sent a message to the server, the server simply broadcast it to all sockets connected to a particular chatroom, then discard the message. The server is written in javascript on the node.JS framework, fully integrated to the web server that we use to server web contents. In the future, once the scaling demands is high, we can easily partition the Socket.IO server out of the web server to take advantage of CDN for static content delivery. On the client side, implementing the client that uses Socket.IO is as simple as including the Socket.IO front-end javascript file and creating a socket for communication with the server. The communication between the server and each client is defined in term of event emission, where each event is composed of an event

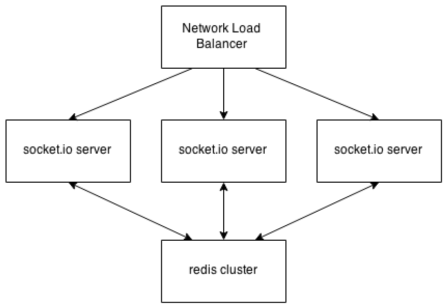

Figure 1: Socket.IO cluster architecture

type and a payload. The event type allows the server and other client to distinguish between the type of messages that it cares about, which in our case composed of two types of events: a new user joins a chatroom, and a new message is sent to a chat room. When a new browser tab opens a particular chat room, the chat client emits a ”new user” event to the user. The server adds the user to the broadcast list of that chatroom, but does not notify other clients of a new member. When a user sends a message, a ”new message” event is emitted to the server, and the server then broadcast that event to all clients in the chat room. This results in each message being sent to only the clients currently joined in the chat, so any new clients that joins the chat room after the broadcast is executed will not see the message.

Another advantage for using Socket.IO is its scalability. Socket.IO has a dedicated module name socket.io-redis that allows us to run cluster of Socket.IO servers that can broadcast and emit messages among themselves through the use of redis service. redis is a distributed key-value cache and storage system, and often referred to as a data structure server. This allows for an easy cluster deployment as each Socket.IO server will be deployed as an edge server, which broadcast messages to the clients connected to it, and push the message to redis to be re-broadcast by all other Socket.IO servers. The chat system architecture is depicted in Figure 1. Redis system may seems to be the single point of failure in this model, but Redis itself can also be deployed as a cluster, which minimizes the chance of total crash stop. Also, as the purpose of the group chat messages is simply for communication from the audiences to the broadcaster, we do not concern much about message ordering that may be different on each client when their messages are processed by different edge servers. Evaluation of the chat system will be discussed later in the Evaluation section.

## 5 File Storage and Persistence

One important feature apart from streaming the code is actually storing the code. This is needed because, the streamers will mostly not be working on a single file at a time, and most projects involve working with multiple files. For example, a user trying to code an android application will have his logic split between manifest files, activity files, xml and other custom files. When a reviewer starts addressing a particular user’s needs by watching his stream, he will only see what the streamer is currently trying to achieve with the code snippet showed on the stream. If the logic is too big or complex, the reviewer might want to take a look at the whole file or the whole project wherever the workflow touches upon. Hence, this functionality provides two important uses:

- The user need not switch between windows or scroll through his current view to clarify the reviewer via the stream. This increases productivity and removes miscommunication errors

- The reviewer can look at the code offline before getting into live help or whenever he is free, so as to get a holistic picture of the user’s project

### 5.1 Feature Decisions inside Google Cloud Storage

To achieve this, we utilized Google Cloud Storage through its Python based utility called gsutil and the Cloud Storage JSON API to store the project files as soon as the user saves it. Following are the main design decisions for choosing the Google Cloud Storage which explains how this particular challenge has been addressed efficiently:

- Files are treated as Objects inside Buckets and supports authentication, opaque storage, immutability and metadata. This translates to: exposing files only to the reviewer, impossible to read code content in an insecure manner, updating files in an atomic way and defining files clearly for ease in retrieval and listing. We currently maintain only one bucket due to Google Cloud's restrictions on number of API calls and cost factors. We can add new buckets as the system scales.

- Strong consistency of code files: it is a read-after-write and read-after-update model to ensure consistency upon user saving the file using the PUT operation. However we do not maintain versions of updates because it is not in the problem scope. GET operations gradually becomes consistent for listing objects across buckets. This happens when one user's files get dispersed into different buckets. However, the possibility of this happening is remote due to a restriction on the amount of data one user can upload. The problem will be addressed when a user pushes a code base large enough to overflow a bucket.

- Authentication is done through OAuth 2.0 and we enforce HTTPS connection for secure transfer of file updates

- Support for resumable upload: which proved to be an advantage when the user's network is already congested with streaming uplinks and their payload data is too heavy to persist in a single API call.

- We are specifying different bucket locations for replicating files across different geo-centers

- Though the platform supports CRC32C and MD5 hashes, we do not hash the files. We rely on the single authentication that will be performed in the beginning.

We use the Google Developer Console and gsutil APIs to monitor the system load in real time and as a collector of data in the end

### 5.2 Tool for Push-Style File Storage

We have developed a custom bash script which acts as the core of the automatic push-style file storage utility. This is to simplify the usage. The only action required by the user is to download our script and run it in a separate terminal. This bash script called 'ChironPush' and it performs two important functionalities. First it checks whether the user has gsutil installed with necessary credentials to use the storage. If not present, it installs gsutil with all necessary credential information. It then runs the event monitoring script which is written using the GNU inotify module. The user decides which project folder or file he wants to sync to cloud and enters the command: 'bash ChironPush folderpath/filepath'. This spawns a new process which runs in the background and starts monitoring the specified folder or file for the following events: modify,attrib,close,write,move,create and delete. Once it detects an event it calls the appropriate gsutil API to mimic the changes in the cloud. The user will not have to take pushing it manually into account and this is the fundamental difference from traditional version control clients like git. Mainly, the main purpose of this functionality is to allow the viewers of the stream to be able to view the most recent version of the files that the streamer is working on without the streamer having to frequently pushed files out to version control. However, this does not mean that the streamer should rely simply on our service and ignores the use of version control completely.

## 6 Rating System for Users

The expert programmers are rated by the novice learners on a scale of 1 to 5 stars on multiple parameters: technical expertise, helpfulness, clarity in communication, presence of examples and timeliness. These are the factors that effectively measure the success of the instruction session and from these ratings, the overall rating is calculated. This rating is carried out on completion of the session. A table called *Rating* is maintained in the database with the overall rating and rating on these different parameters. This information is used by the future users to assess the efficacy of the instructors. The Rating table only consists of aggregate ratings of the instructor on different parameters. The individual rating per session is only used to update the aggregate rating of the instructor. There is another table called *PeriodicRating*, which has monthly aggregates of the rating information for each instructor. This information can be used in the future to provide rating information in the form of a line graph , that show the rating trend of the instructor based on various parameters. The user interface for this feature is in progress and will be added in the future. In order to preserve the per session rating information of instructors, we are planning to move the information to a low-cost cold storage system called Cloud Storage Nearline.

## 7 Evaluation

### 7.1 Live Chat

We conduct an experiment to see the scalability of the chat by evaluating the performance of the system with 3 Socket.IO servers and 1 redis server. The client consists of three separate VM instances, each running 100 threads of chat clients that emits 10 messages per second, for a total of 10000 messages per VM instance. The system topology for the chat client is exactly as described in the earlier section. We run experiment on two cases: one where there is

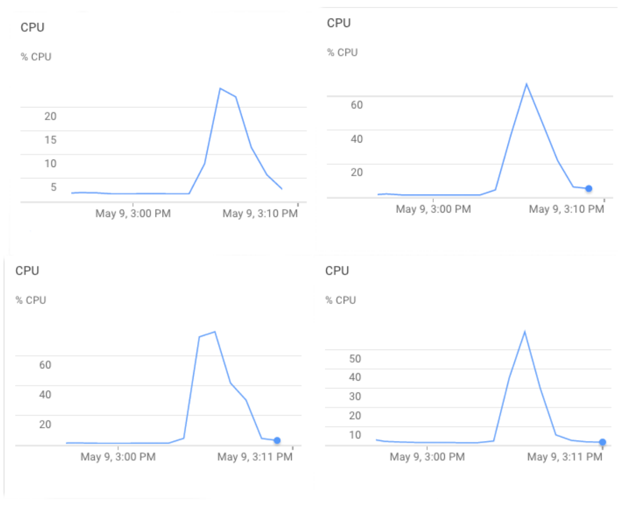

Figure 2: 3 servers cpu benchmark

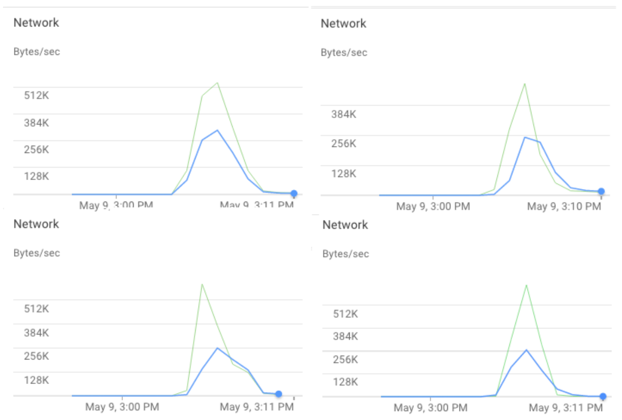

Figure 3: 3 servers network benchmark. Green line is outgoing network and Blue line is incoming network.

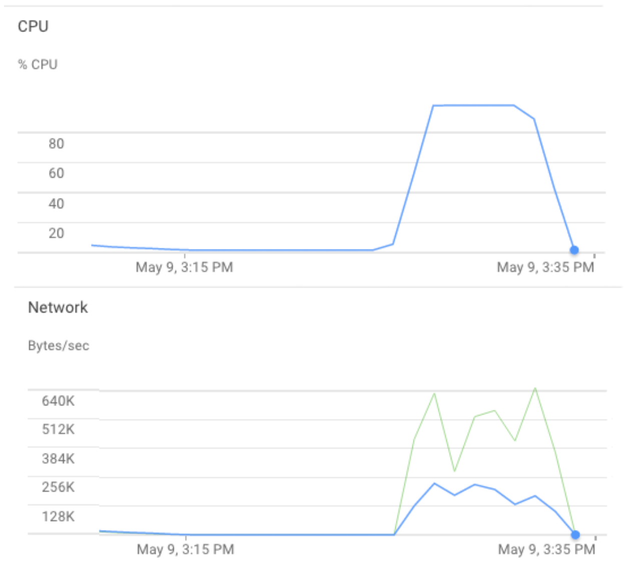

Figure 4: 1 server benchmark

only one Socket.IO server, and another where three servers are deployed with redis. Once all the clients finish sending the messages and terminate, we extract the cpu utilization and network utilization of each instance from the google cloud console. The message frequency used here is for a rather extreme case of spamming as it should normally takes more than 0.1 second for each user to write down a message before they send it. Also, we are putting all 300 clients into the same chat room, which means that each message will be amplified by 300 when it is broadcast by the servers. The result is shown in figure 2 - 4. From the plots, it is clear that this burst of messages overload the cpu in the single server case as the CPU utilization is quicky pushed close to the maximum utilization along with a spike in the network utilization of the instance. It is worth noting here that the egress traffic is higher than the ingress traffic because each message coming in to the server is broadcast to all connected client. On the other hand, When the same traffic is instead distributed evenly among three border servers using Google's network load balancing service that automatically balance the load between a pool of servers, each server's CPU utilization is around 60% and the network utilization on each server is roughly the same as the single server case, which we suspect is the maximum network utilization allowed for each instance imposed by the google cloud system. However, it is clear from the plot that the network spikes last much shorter than the single server case. It also worth noting that the cpu and network utilization of the server that hosted redis is very low compared to the Socket.IO servers, which implies that one radis server can serve more than three Socket.IO servers, and we projects that redis cluster size will scale linearly with the number of Socket.IO servers. As the messages are only cached in the redis servers and not actually saved in long-term storage, we do not have to worry about storage allocation of the messages, which make the system highly scalable.

## 7.2 File Storage

### 7.2.1 Cost Effective Storage in Google Cloud

Initially, one of the important observations was that the bucket creation operation is expensive. Originally, the system was designed to create one bucket per user and different project folders will be treated as objects. Seemingly logical at first, however as the system scales up minutely by batch user creation we encountered a problem. We found that bucket creation, listing and updates are all rate limited and class A operations. Since the monthly storage of our model cannot be predicted, the average monthly usage cost estimation will be infeasible. As a result, the system is revised to create objects for each user inside the same bucket and the replicas in the second bucket. This reduced the overall cost to an extent, from a standard storage model which costed \$0.026 for 1GB a month, to a Nearline Storage model which now cost only \$0.01 for 1GB a month. This adoption has become possible due to two factors: infrequent access of files by user and extended file retention. We allocated 100 MB per user and we

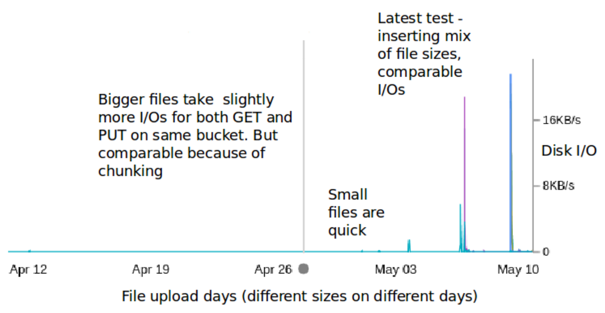

Figure 5: stable disk I/O for different file sizes

have not scaled enough to test anyone exceeding this limit. We can impose extra charge upon request of storage.

#### 7.2.2 Stress Testing

We started with testing every bucket and object APIs first since there are many updates with the versions. Google Cloud Storage is not as mature as Amazon S3 and needs thorough check for methods. We ran multiple instances of our scripts to create objects in the same bucket simultaneously to test delayed responses and HTTP503 responses. This would also cover tests for failure paths because the requests originate from the same location. The system can respond in a timely manner without any errors. Overall read and write throughput is satisfactory, with hindering rates for larger files. The reason we inferred is that Google fragments larger files into smaller chunks so that the upload process can be resumed in case of intermediate failures. Some users who work with image processing or machine learning code files have big files which needs to be uploaded. In those cases, the response is slightly slow but not adverse. One important metric we could not test was how the system handles loads when the request came from different geographic regions. This is because the request will hit different compute engines in that case. We would include that in the future work as we grow our user base across the continents.

### 7.3 Usability testing

The most important factors for determining the success of a consumer facing applications and projects are ease of usage and the likeability of the application. Usability testing is the best evaluation method to gauge these factors. Usability testing is a technique used in user-centered interaction design and refers to evaluating the product or application by testing it with representative users.

#### 7.3.1 Participants

For the user study to be successful, the background of the participants taking part in the usability testing should closely mimic the background of the target audience the application is meant for. There are two kinds of audience our application mainly targets: the novice user, who does not have much experience with programming and has necessary computer skills, and the expert user, who is proficient with different programming languages. For evaluating the novice user side of our application, we requested 10 college students, who did not have Computer Science as their major or minor. They are students from a range of different majors and programs like Physics, Mathematics, Civil Engineering etc. All the students participating in this role had limited or no experience with programming and were enthusiastic about trying out our product. Evaluation of the expert programmers' side of the applications was carried out with the help of 6 students who are pursuing their Masters in Computer Science and 1 student, who was pursuing his Phd in CS. They all had extensive programming background, both in an academic and professional setting.

#### 7.3.2 Task identification

The key step in our usability testing was identifying the crucial tasks that were unique to our application and were key in determining the success of our application. For example, tasks such as *Create a new user account*, *logging out of the account* etc. which is are important tasks,

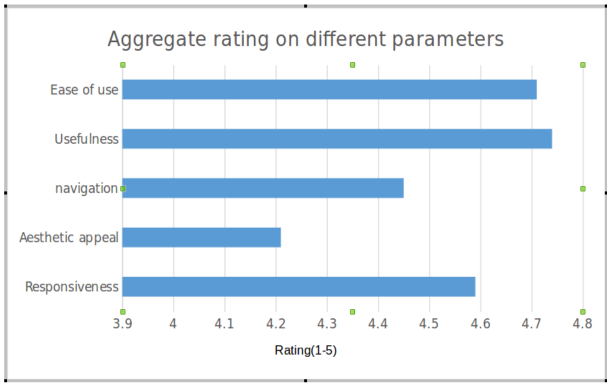

Figure 6: Aggregate rating on different parameters.

but are prevalent in almost all the websites and could be done with relative ease were discarded from the testing process. For both the set of target users, a set of important tasks were identified and were presented in different combinations, to avoid learning effects. The testing for a novice user and an expert user was coupled because of the involvement of both kinds of users in almost all the tasks.

#### 7.3.3 Study

The participants were given a thorough explanation of our intention of conducting this study. They were given the individual tasks based on the role they were playing and their actions were keenly observed to identify the tasks during which they faced difficulty. If faced with hurdles, the members of our project assisted the participants to achieve their intended actions. The users were asked to think out loud and notes taken. Towards the end of our study, we had the participants fill out a feedback form for rating our application on a scale of 1 to 5 based on various parameters: responsiveness, aesthetic appeal, navigation, usefulness and ease of use. The feedback results were used to make sure that the effort spent was used in the right direction.

The application received relatively low ratings in terms of aesthetic appeal and it was understandable as our effort during the initial iteration was spent mostly on functionality. The suggestions from the users were also taken into account to improve the user interface design of our application. The high ratings for *Ease of Use* and *Usefulness* were good indicators that we were heading in the right direction.

## 8 Future Work

### 8.1 RTMP server

The nginx-rtmp server that is currently deployed does not scale as well as we'd hope because the system does not support dynamic cluster configuration change. Although the server does support dynamic push and pull of streams inside a cluster, the cluster topology has to be specified at the time the server spins up, and the server will have to be restart when there is a configuration change. The next steps in the implementation is to find alternative server implementation that support dynamic cluster topology change, or implement a video load balancer that controls the nginx-rtmp server configurations in the cluster.

### 8.2 File Storage

The manner in which the automatic push of user's project files works, is dependent heavily on the GNU inotify module. Having said that, we also need to keep in mind that, at times the user creates a long tailed directory structure (possibly by copy pasting from a different place) and then immediately the user makes a new file addition or change in the last directory. From the man page of the inotify module, it's clear that there are possibilities of race conditions occurring in the recursive directory watching code. This can cause events to be missed and hence may not be pushed properly to the Google Cloud. This is probably not fixable as of now unless we come up with an in-house library to monitor file system events. We treat this case as a very rare event and hence sticking with the current engineered solution. It is also assumed that the inotify event queue will never overflow and the user's system has enough RAM to handle this.

We will also be testing the storage system's response as the user base grows across continents. This will possibly create high traffic in the buckets and will give ideas to think further on how to distribute the objects efficiently.

## 9 Conclusion

Programming is a very competitive skill to have in the current job market, and as a result there are many online resources that offer training in programming and software engineering-related fields. One trend that is becoming popular in this market is video live-streaming of coding session, where experienced programmers broadcast their coding session for others to watch. Based on this learning model, we introduces Chiron, a video live-streaming platform aims to provide interactive code review by encouraging novice programmers to stream their coding session with experienced programmers acting as mentor. The platform consists of a rtmp-based video streaming service, live chat system for communication between the mentors and the streamer, persistent code storage system for real-time read access to the project's source code, and a rating system for each user. Chiron is built with scalability as the main priority, and each component of the system is chosen such that the system will be able to quickly scales to meet with the demand of the users.

## References

"livecoding.tv" 2004. 6 Apr. 2015 http://livecoding.tv/

"watchpeoplecode.com" 2004. 6 Apr. 2015 http://watchpeoplecode.com/

"Twitch." 2004. 6 Apr. 2015 http://www.twitch.tv/

"Real-Time Messaging Protocol (RTMP) specification - Adobe." 2010. 6 Apr. 2015 http://www.adobe.com/devnet/rtmp.html

"Open Broadcaster Software - Index." 2013. 6 Apr. 2015 https://obsproject.com/

"XSplit - Free Easy Live Streaming and Recording Software." 2011. 6 Apr. 2015 https://www.xsplit.com/

"FFsplit." 2012. 6 Apr. 2015 http://www.ffsplit.com/

"Live Webcasting Software — Telestream Wirecast — Overview." 2011. 6 Apr. 2015 http://www.telestream.net/wirecast/overview.htm

"Tunneling with RTMP encapsulated in HTTP (RTMPT ..." 2013. 6 Apr. 2015 http://blogs.adobe.com/connectsupport/tunneling-with-rtmp-encapsulated-in-http-rtn

"NGINX — High Performance Load Balancer, Web Server ..." 2005. 6 Apr. 2015 http://nginx.com/

"jwplayer/jwplayer GitHub." 2013. 6 Apr. 2015 https://github.com/jwplayer/jwplayer

"Video.js: HTML5 Video Player." 2005. 6 Apr. 2015 http://www.videojs.com/

"Automattic/socket.io GitHub." 2014. 6 Apr. 2015 https://github.com/Automattic/socket.io

"Node.js." 2014. 6 Apr. 2015 https://nodejs.org/

"Socket.IO ? Using multiple nodes." 2014. 6 Apr. 2015 http://socket.io/docs/using-multiple-nodes/

"Cloud Storage - Google Cloud Platform". 2012. 6 Apr. 2015 https://cloud.google.com/storage/

"Objects - Cloud Storage Google Cloud Platform". 2014. 6 Apr. 2015 https://cloud.google.com/storage/docs/json_api/v1/objects

"Google Cloud Storage JSON API Overview - Cloud Storage ..." 2014. 6 Apr. 2015 https://cloud.google.com/storage/docs/json_api/

"Buckets - Cloud Storage Google Cloud Platform." 2014. 6 Apr. 2015,https://cloud.google.com/storage/docs/json_api/v1/buckets

"Objects: insert - Cloud Storage Google Cloud Platform." 2014. 6 Apr. 2015,https://cloud.google.com/storage/docs/json_api/v1/objects/insert

"API Reference - Cloud Storage Google Cloud Platform." 2014. 6 Apr. 2015,https://cloud.google.com/storage/docs/json_api/v1/

"OAuth 2.0 OAuth." 2010. 6 Apr. 2015, http://oauth.net/2/

"GoogleCloudPlatform/crc32c-java GitHub." 2013. 6 Apr. 2015, https://github.com/GoogleCloudPlatform/crc32c-java

"MD5 - Wikipedia, the free encyclopedia." 2003. 6 Apr. 2015, http://en.wikipedia.org/wiki/MD5

"Google Developers Console." 2014. 6 Apr. 2015, https://console.developers.google.com/

"gsutil Tool - Cloud Storage Google Cloud Platform." 2014. 6 Apr. 2015, https://cloud.google.com/storage/docs/gsutil

"Bash - GNU Project - Free Software Foundation." 2011. 10 May. 2015, https://www.gnu.org/s/bash/bash.html

"inotify - Wikipedia, the free encyclopedia." 2005. 10 May. 2015,http://en.wikipedia.org/wiki/Inotify

"Nearline Storage - Cloud Storage Google Cloud Platform." 2015. 10 May. 2015, https://cloud.google.com/storage/docs/nearline-storage

"AWS — Amazon Simple Storage Service (S3) - Online Cloud ..." 2006. 10 May. 2015, http://aws.amazon.com/s3/

"HTTP/1.1: Status Code Definitions." 10 May. 2015,http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

"Race Conditions." 10 May. 2015, http://en.wikipedia.org/wiki/Race_condition