Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Matti Jokitulppo

Real-time sentiment analysis of video calls

Master's Thesis
Espoo, March 17, 2019

Supervisor: Senior University Lecturer Ari Korhonen
Advisor: Professor Teemu Leinonen

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Matti Jokitulppo		
Title:	Real-time sentiment analysis of video calls		
Date:	March 17, 2019	**Pages:**	58
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Senior University Lecturer Ari Korhonen		
Advisor:	Professor Teemu Leinonen		

In recent years, with ever-increasing internet connection speed and bandwidth, video-focused software has become more and more popular for both work and pleasure. Examples of such applications include Skype, BlueJeans or iOS FaceTime. These applications, and the various interactions facilitated by them contain lots of interesting data that we feel would be very fruitful to gather and analyze. Within the context of this thesis, we focused on evaluating the potential of collecting sentiment analytics from video teleconferencing both on an individual and group level, for the purpose of helping people reflect on their own behavior and regulate their emotions.

To achieve this, we developed a composable, scalable microservice-based analytics pipeline for video and speech, and a browser-based web application to demonstrate it. We evaluated already existing solutions for gathering sentiment analytics, and integrated two of them into our analytics pipeline. The whole system was deployed in a virtualized container environment using Docker. Besides the pipeline and web application, we also designed and implemented some visualizations for the data that we gathered.

In the end we developed a working prototype, although deeper analysis and evaluation of the actual accuracy of its results needs to be performed. Human emotions are rather difficult to quantize. We found that the current APIs and libraries publicly available for performing sentiment analysis are already quite accurate and feature-rich, and we expect them to get even better.

Keywords:	programming, web development, analytics, learning analytics, distributed systems, sentiment analysis
Language:	English

Acknowledgements

I would like to thank my parents, and Teemu and Ari for the interesting thesis subject, and also my employer Reaktor for supporting me throughout the whole writing process.

Helsinki, March 17, 2019

Matti Jokitulppo

Contents

Chapter 1

Introduction

1.1 Problem statement

In recent years, with fast, low-latency internet becoming readily available for more people, the market for real-time video streaming has grown significantly. This increase in internet quality has been especially impactful for mobile applications, from large corporations like Apple with FaceTime, Google and Google Duo, to smaller up-and-coming companies like Snapchat and Instagram, whose products also offer real-time video streaming features. In fact, the networking hardware and telecommunications company Cisco predicts that live video will consist of 17% of all internet traffic by 2022, up from the 5% it did in 2017 [33]. Besides improvements in connectivity, there have also been large-scale efforts within the software industry to create an open-source standard for enabling real-time peer-to-peer audio and video communication, namely WebRTC. [17] WebRTC is meant for enabling real-time video calls for on both web browser and mobile applications, and does not require the end-user to install any third-party browser plugins or add-ons, unlike previous browser-based streaming protocols. Besides WebRTC, HLS (HTTP Live Streaming) has also been a major player, with online media streaming providers like Twitch giving the particular standard their backing[1].

Furthermore, with the advent of social media sites like Facebook, Twitter and Instagram, the demand for large-scale data analytics has also grown tremendously in the last few years. Companies and advertisers alike both want to better understand their users and their behavior to maximise their profits, treading occasionally using somewhat morally ambiguous tactics. This trend of collecting data has also been boosted by the availability of

[1] https://blog.twitch.tv/twitch-engineering-an-introduction-and-overview-a23917b71a25

cheap processing power and storage to analyze that data, due to breakthrough of cloud computing. This is thanks to services like Amazons AWS and Google Cloud Engine, where from one can rent capable virtual machines to run large-scale compute jobs on, which are then billed on an as-needed basis. Besides corporations collecting data on their users, individuals are also interested in tracking their well-being and life in general with fitness mobile applications, for example. [31]

Taking into account these two somewhat recent trends of the increase of live video streaming and the growing interest and commercial availability data science, and also the increase of readily available tooling for both, it makes sense to research what sort of useful metrics could potentially be extracted and analyzed from live video teleconferencing. The research questions this thesis will attempt to answer are, what kinds of emotion-related metrics can be extracted from conference-call type videos between two or more participants in real-time, what are the best open-source tools and methodologies for gathering them, and how to structure the data and application infrastructure such that it can be easily analyzed to provide useful insights at a later point in time. Within the thesis we will particularly focus on using this data to aid people in self-reflection, since it is often difficult to regulate emotions and be present during teleconferencing. We will also attempt to visualize, compare and contrast the results gathered from both an individual and the larger group, or discussion. Leinonen (citing Vygotsky) [21, 34] describes reflection as an individual internalized conversation, that aims to conceptualize internal mediated objects into something more concrete. This long-term process of social situations with other people and creating and invoking these intermediate objects is crucial to developing the skill of reflection, according to Vygotsky [34]. To achieve this formalization of reflection, a browser-based prototype was created, which attempts to gather and aggregate data from various sources on both an individual and a group level. Also, we will shortly go through some more in-depth ethical questions that were raised during the thesis process that were not looked into with much depth in other sections of the thesis.

1.2 Structure of the thesis

The rest of this thesis is organized as follows. Chapter 2 describes some of the background on why we feel creating such a sentiment analysis tool is necessary. Chapter 3 contains an evaluation of existing solutions and a high-level overview of the application architecture, planned features and the overall development cycle. After the fact, Chapter 4 describes a more throughout

description on the actual implementation, and choices and benchmarks made. Chapter 5 contains an evaluation of the final outcome of the prototype, its strengths and weaknesses and ideas for future development and research efforts. Finally, Chapter 6 contains a short post-mortem on the thesis and its findings.

Chapter 2

Background

2.1 Computer-supported cooperative work

Throughout history, humans have imagined technological advancements of the far future in a grandiose, some could say even bombastic manner. To name an example, Italian artist Luigi Russolo's 1913 manifesto "The Art of Noises" argues that the recent influx of new sounds and the changing soundscape of the city birthed by the ongoing industrial revolution at the time renders the whole known musical history of mankind hopelessly deprecated. Russolo speculated that the advent of electricity would allow the musicians of the future to "substitute for the limited variety of timbres that the orchestra possesses today the infinite variety of timbres in noises, reproduced with appropriate mechanisms" [15]. However, according to legend Russolo's new theories on the emerging nature of music itself did not fare very well in practice. Reportedly, his first concertos resulted in mass riots from the part of the audience.

Stepping out of the realm of science fiction, barring some obviously groundbreaking technological advancements, the effects of new inventions in our everyday lives are often quite subtle. For example, instead of the dystopian world-dominating artificial intelligence seen in 60s science fiction movies, in reality artificial intelligence ended up being much smaller in scope regarding the tasks a given program could solve. This all-encompassing human-like intelligence is generally called strong AI, and the more task-focused approach is called narrow, or weak AI. Examples of weak AI used in modern times include many recommendation systems that aim to cater content and advertisements catered for our individual viewing habits or determining a fair price and optimal route for an Uber trip. Even though the research field is decades old, artificial intelligence has gained a large boost in popularity in re-

cent years due to the commodization of cheap processing hardware. Most of AI tools we encounter are based on neural networks and similar classification based approaches, like the recommendation engines mentioned before.

In 1962, Douglas Engelbart proposed that the computer should not only be used for computations, but rather as a tool for augmenting human intelligence [8]. It is easy to see that his visions have since become a reality as computers become more and more ingrained in our daily lives in many ways. One slightly more recent way of conceptualizing the way groups of humans work together assisted by computers comes from the field of computer-supported cooperative work. Mayel-Patel defines CSCW systems as "complex distributed applications that incorporate a number of different tools working in concert to support goal- and task-driven collaborations that involve multiple sites and participants" [25]. In general terms, CSCW systems can be broadly categorized into four different kinds. Originally proposed by Rodden [29], Figure 2.1 contains a modernized version by Mayel-Patel et al. In the figure, the types of CSCW systems are categorized by both the latency between participants and the synchronicity of interactions. For example, messaging systems include emails, bulletin boards and news groups. Co-authoring systems are platforms for collaborative editing like Google Docs or Trello, and the conferencing category includes synchronous communication between remote participants. The system described in this thesis can be categorized as such.

2.2 Supporting reflection via analytics

According to Kahneman's popular psychology book Thinking, Fast and Slow [18], the whole of human cognitive perception can be divided into two separate systems working together, System 1 and System 2. System 1 is a legacy of our evolutionary past. It is in charge of reacting instinctively to external stimuli with little cognitive effort. Examples of such actions include driving a car on a straight road, or reacting with disgust upon being exposed to something unsightly.

System 2 is more deliberate, logical and requires much more of our focus and energy to operate. It deals with things such as individual long-term decision making and self-reflection. According to Kahneman's hypothesis, our personal thoughts and actions can vary wildly, depending on which System is in charge at the moment, so to say.

Within the context of a social interaction, be it between two people or in a group situation, face-to-face or not, our attention to the conversation at hand can often falter due to a variety of reasons, causing us to fall back to the

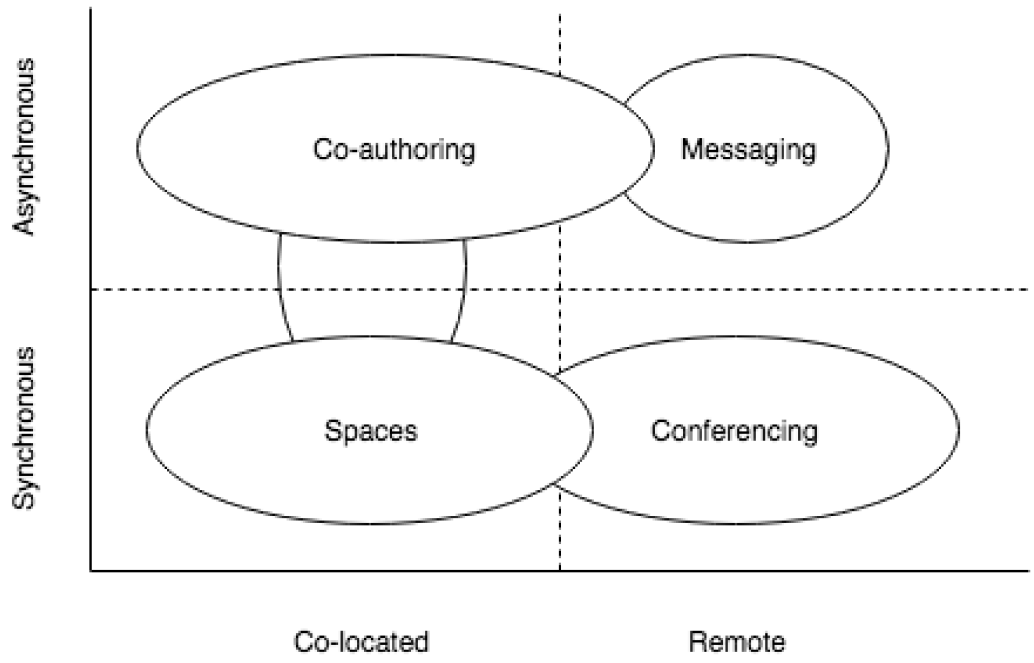


Figure 2.1: Classifications of types of computer-supported collaborative work

relatively low maintenance System 1. This is even more apparent when such interaction happens via teleconferencing, as by nature it is lacking the social presence of other people, which we have been accustomed to during hundreds of thousands of years of evolution. Besides, many other social cues, changes in expression and mood and such can be easily missed or misinterpreted. To help mediate this problem, our proposed solution is to build a system of tools that work in the background, invisible to the end-user, collecting relevant sentimental metrics to better facilitate self-reflection and learning post-conversation. According to Leinonen et al. [21], most existing research done on reflection utilizing computers has focused on the writing aspect, with substantially less research done on utilizing digital audio and video, so we feel such experimenting with such a system is necessary. With modern machine-learning and natural language processing techniques, we believe that a variety of useful metrics can be gathered from these online interactions which can then be further used to aid this goal of self-development. Furthermore, even though the results itself that can be measured might not be very accurate themselves as opposed to writing one's thoughts down by themselves, our hypothesis is that with continuous sessions the quantity of data will trump the quality, so to say. This is especially if analytics gathering can be made totally automatic and non-intrusive from the point-of-view of the end-user.

In attempts to design tools for aiding in reflection, one early categorization by Schön was to separate them into reflection-in-action and reflection-on-action [30]. Tools that utilize reflection-in-action aim to collect data influence people's behavior in real-time, while tools that utilize reflection-on-action draw on past experiences to aid in reflection. We feel that, due to the nature of the data collected, it would be best not to offer potentially inaccurate real-time feedback to the user, but rather to focus on visualizing the big picture post-session. Providing real-time feedback to the user could paradoxically cause the user to actually be even more distracted.

Furthermore, in recent years, the concept of self-tracking has become increasingly popular [24] among first-world people,with the advent of fitness trackers such as Fitbit, various sleep monitoring and productivity applications and Internet-of-Things enabled household products such as bathroom scales. There was even a movement started by two journalists in 2007 around the concept of quantified self, with its members attempting to incorporate monitoring technology into their daily lives in various ways to enhance their wellness and productivity, among other proposed benefits. One of the movement founders, Gary Wolf described quantified self as 'Self-knowledge through self-tracking'[1]. Indeed, there has been a noticeable increase in the concept of achieving your best self in western culture in recent years, and this core concept of self-improvement is an important motivation for self-tracking [22]. We feel that a tool for enabling self-reflection such as the one outlined in this thesis is also natural progression for this as of yet ongoing trend of interest in self-development via means of technology.

[1] `http://antephase.com/themacroscope`

Chapter 3

Methods and materials

3.1 Comparison of existing solutions for video sentiment analysis

3.1.1 SentiVid

SentiVid[1] is an existing academic solution for real-time sentiment analysis, originally developed by the The Intelligent Systems Group at the Technical University of Madrid. Their solution uses the Google Speech Recognition API available natively in the Chrome Browser to transcribe a video in real-time. The research group in question have also developed their own sentiment and emotion analysis service called SEAS[2] (Sentiment and Emotion Analysis Services), which is then used to analyze the sentiments gathered from the submitted video. To gather and represent sentiment analysis results, they use their own data schema by the name of Marl [35]. For representing emotions, they also have their own in-house solution by the name of Onyx [32].

One thing of note is that in SentiVid the video streaming itself is not real-time, but it uses user-defined pre-uploaded videos as the source material, which are then transcribed in the browser and sent to their own API for analysis. Furthermore, their proposed solution only uses audio data, and it has no facial recognition, or similar video-related features. Their custom in-house data schemas are also somewhat cumbersome to re-use from a subjective point of view, and their API is also not very well documented and difficult to set up. However, their real-time speech-to-text works rather well based on preliminary testing.

[1] `https://github.com/gsi-upm/video-sentiment-analysis`
[2] `https://github.com/gsi-upm/SEAS`

3.1.2 Google Cloud Video Intelligence

Google's offering for real-time video analysis is a bundle of cloud-based tools packaged under the name of Google Cloud Video Intelligence[3]. Besides plain sentiment labeling, their service also offers a wealth of other features, such as video transcription, advanced metadata labeling, detecting scene changes and even identifying potentially pornographic content. Their offered use-cases include generating meta-data for video search, building recommendation engines or displaying contextual advertisements.

While their API is well-documented with libraries available for many different programming languages, much like the aforementioned SentiVid their proposed solution does not deal with live video content. There is also no information available on how the underlying system actually works. It stands to reason this is strictly due to business reasons. While their tool suite is largeá, their sentiment analysis functionality is also somewhat lacking, being limited to simple labelling.

3.1.3 Jargon.ai

Jargon.ai[4] is a teleconferencing product by New York-based start-up of the same name. Their product is very close to what will be implemented in this thesis, offering a variety of sentiment analytics based on things like facial expressions and heart-beat measurements extracted from the video footage. Besides offering real-time analytics, they also offer after-session follow-up reports with AI generated highlights. How this advertised AI works exactly is not documented on their website or mentioned in their developer documentation, unfortunately.

3.1.4 Summary of existing solutions

During our initial research we could not find a tool for sentiment analysis for assisting in self-reflection that fit all our needs. SentiVid only used audio captured from the video as the sole source of data, and did not support live video. Google Cloud Video Intelligence had a wealth of features, but lackluster support regarding sentiment analysis, and also did not support live video. Of all the solutions evaluated Jargon.ai was by far the most promising, aggregating data from multiple sources and even generating reports for aiding in self-reflection. However, Jargon.ai is proprietary software hosted and maintained by a private entity. Therefore, no guarantees can be made

[3] `https://cloud.google.com/video-intelligence/`
[4] `https://www.jargon.ai/`

Table 3.1: Comparison of existing sentiment analysis software

	Open source	Audio	Video	Live collecting	Reports
SentiVid	Yes	Yes	No	No	No
Google Cloud Video	No	Yes	Yes	No	No
Jargon.ai	No	Yes	Yes	Yes	Yes

on what the data will be used for, nor do they disclose how their pipeline actually processes, evaluates and stores data. Table 3.1 contains a summary of the features of the existing software we evaluated.

All in all, in our opinion there is a clear need for an open-source, scalable, transparent tool for gathering sentiment analysis that can be easily self-hosted. In the next chapter we will describe and document how we implemented such a tool, and finally we will describe and evaluate our findings and results during the development process.

Besides actual consumer-oriented software, not much existing research can be found on real-time sentiment analysis for videos. Previous research is mainly focused around YouTube and other video sharing services, such as L. Kaushik's approach, focusing on extracting audio from YouTube videos, running it through a speech-to-text service and then extracting sentiment data from the results [19]. Additionally, there exists a sizeable amount of work for predicting the popularity of online videos via sentiment analysis [11, 37]. However, we will focus strictly on human behavior and emotions within the scope of video teleconferencing in this thesis.

3.2 Prototype development process

Historically, software was developed in a very linear manner. For example, first one would gather requirements, then write a comprehensive design document, and then finally implement, then verify the system that was built. This software development paradigm is now called the "Waterfall model", dating back as far as 1956 [3]. However, in the turn of the century many professionals in the software industry took note of the fact was that software and software projects were getting increasingly demanding and complex. The traditional waterfall model, by nature, has problems coping with changes to the system, since it requires all design is done up-front before the actual implementation. Also, if defects in the system are found in the very last parts of the development process, it may very well be that there will not be enough time or budget left to fix them in a proper manner, since each stage is usually strictly time-slotted and budgeted for.

In 2001, as a solution to these problems a group of seventeen software developers joined together and co-wrote The Agile Manifesto [2], which lays out twelve principles to combat the inherent problems found in Waterfall. Among them are principles such as "Welcome changing requirements, even in late development" and "Working software is the primary measure of progress". In the initial stages of the development process, we decided to utilize principles of Lean Software Development, as laid out by Tom and Mary Poppendieck in their book "Lean Software Development: An Agile Toolkit" [27]. Lean Software Development is an extension built on the principles laid out in the Agile Manifesto, emphasizing fast delivery and iterative improvement over up-front decisions. We felt that a lean software development process would be a natural fit for the actual implementation phase, as in the very beginning we did not have much of an idea what the final end-result would entail, what it would look like, what sort of analytics data we could reliably gather and such. The bulk of the actual design and development work was also done by one person with the data visualization being a notable exception, so it was natural that there was no real need for cumbersome task management, progress tracking or other such tooling.

3.2.1 Functional and non-functional requirements

As outlined by Fulton [13], a functional requirement defines a function or module of a system, software or otherwise where a function is described as a specification of behavior between outputs and inputs. In layman's terms, functional requirements describe what a system does. Meanwhile, to help in further fleshing out functional requirements one can use non-functional requirements, which place additional constraints on the system which cannot be expressed as functionality, strictly speaking. Examples include needs such as performance, quality-of-service and security needs. According to Adams [1], functional requirements specify the application architecture, while non-functional requirements specify the technical architecture of the system.

Some core technical and non-technical requirements we came up with before starting to implement the analytics prototype can be found in Table 3.2. The actual functional requirements we gathered before implementing the prototype contain the bare minimum to create a Minimum Viable Product for the analytics software. An MVP is considered to be a product that has just enough features to satisfy the core functionality of the product. According to Ries [28], MVPs are an excellent way of implementing software in the world of start-ups, since they allow companies to get their products to market fast, and also allow them to quickly prototype and evaluate new things without committing too much time and other resources. The functional requirements

Table 3.2: System requirements

Functional requirements	**Non-functional requirements**
The system will collect analytics from live video	The system should work well on the latest browsers
The system will collect analytics from live audio	The system should be easily horizontally scalable and well-designed, implemented using the microservice paradigm
The system will persist the data collected	The analytics collecting should happen in the background, not interrupting the user
The system will show simple visualizations of the data	The analytics collected should be as anonymous as possible

for our prototype may seem trivial, but it is important to write them out before work can begin, so that the objective of the thesis is always clear throughout the development process.

The non-functional requirements collected and refined in the beginning were somewhat more in-depth than the functional ones. For our project, they concern mainly the technical architecture, pertaining to scalability and non-intrusiveness, which we will examine in-depth in Chapter 3.3. Since we wanted the analytics collection during the conference to be as opaque to the users as possible, we also defined a requirement toensure that user data would be kept as anonymized as possible from the very start, due to various ethical reason. We will go over how well we accomplished these requirements in Chapter 5, and also look at what features the MVP might be potentially expanded with.

3.3 High-level software architecture design

Traditionally, software has been designed and implemented as monoliths, usually written in a single programming language, running under a single process that handles all application functionality, such as business logic, rendering the user interface and interacting with the chosen data store. Nicola et al. defines a monolithich application as a "a software application composed of modules that are not independent of the application to which they belong". [6]

While there is nothing inherently wrong with monolithic applications, if care is not taken over time they tend to become tightly coupled and a given

module becomes difficult to develop and change without causing issues in other software modules that depend on it. To combat these issues, a new design pattern called microservices has become increasingly popular lately. In a microservice architecture, a system is composed of a group of loosely coupled services that can be developed and scaled separately. These services generally do not share any common resources, but rather they interact with each other through some other means, message passing for example. Obviously, this induces additional latency for module communication, since in a monolithic software communication between modules generally happens in-memory via method calls or shared variables, for example. Additionally, microservice-based systems require a more sophisticated deployment pipeline and versioning system than monolithic ones. Typically, monolithic software is scaled vertically, but microservice-based services are scaled horizontally. Scaling horizontally means adding more nodes to a system, while scaling vertically means adding more resources to a single node [26]. It could be thought that microservices are a natural partner for lean software development, since both focus on breaking things down into smaller pieces. Microservices are also a natural fit iterative development, since the software can be implemented in parallel, in small cycles.

For the implementation part of this thesis, we decided to adapt a microservice-oriented architecture based around the publish-subscribe pattern [12]. Looking over at our initial requirements, it is clear that we have one or more sources of analytics data (video, audio and so on), and one or more consumers to analyze this data. These consumers would then pass their results onward to be collected and persisted somewhere. In our initial designs, a messaging queue solution of some sort would act as the mediator between the consumers and the producers, allowing them to function, scale and be updated separately, as long as the agreed upon messaging format would stay the same.

A high-level architecture diagram for the application can be seen in Figure 3.1.To summarize the planned architecture, first the user opens the web application in their browser, and initiates a teleconference. During the call video frames and speech are streamed from the user browser into a WebSocket proxy (The publisher), which validates and formats the data and publishes it into a message queue. When new data is written to the queue, the Consumers ingest them, perform a specified sentiment analysis task on the data set, and write it to another message queue when they are finished. Finally, the Writer reads this data from the queue and persists the results into a data store. Each of these components are an individually scalable and deployable microservice, loosely coupled together using message queues to asynchronously write and read data. Each microservice functions in a

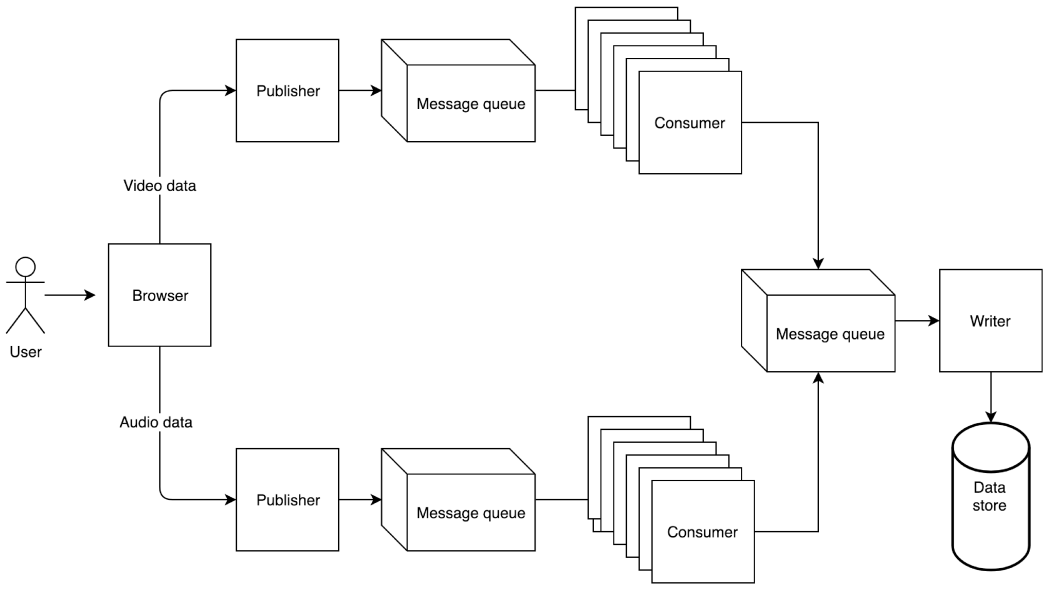


Figure 3.1: Analytics pipeline architecture

deterministic, referentially transparent manner, meaning that a given input will always result in the same output, which makes testing and development quite a lot easier.

3.4 Real-time audio and video streaming

3.4.1 WebSockets: Stateful protocol for real-time communication

The WebSocket protocol [10] is an IETF-standardized protocol for full-duplex communication in the browser, over a single HTTP connection. When utilizing HTTP, a new TCP connection must be opened and closed for each HTTP request. Additionally, these connections are completely stateless, making it impossible to distinguish user sessions with plain HTTP. This can trivially be solved by using HTTP cookies or some other way to persist user sessions, but doing so still carries the overhead of opening separate TCP connections for each request. Furthermore, even with cookies the server has no way of pushing data to the client, but instead the server must continuously poll the client for possible updates.

When using WebSockets, none of these restrictions apply. Both the client and server are free to send and receive data as they wish over a single TCP connection. The initial handshake for initiating a WebSocket session works

over plain HTTP. First, the client sends a HTTP GET request with an Upgrade header sent to server, seen in Listing 3.1. The server then agrees to a switch to the WebSocket protocol by sending back a response with Upgrade header, seen in Listing 3.2. Once this handshake is complete, the same TCP connection used for the initial HTTP handshake is re-used for the full-duplex WebSocket stream. Referring back to Figure 3.1, in the implementation, WebSockets are heavily used to transmit data such as the transliterated text and video frames to the backend for processing.

Listing 3.1: WebSocket request frame

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

Listing 3.2: WebSocket response frame

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Wed, 16 Oct 2013 10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

3.4.2 WebRTC

WebRTC [17] is an open standard for enabling real-time audio and video streaming between two or more clients without requiring the usage of an intermediary server. At the time of writing, WebRTC is supported natively by all the major browser vendors. WebRTC is meant mostly for browsers and web applications, but different implementations for native applications also exist.

The three core standardized abstractions offered by the WebRTC standard are MediaStreams, which offer a way of accessing the camera and microphone of a device, RTCPeerConnection which represents a persistent connection between the local device and the remote device, and RTCDataChannel which in turn represents a bi-directional duplex stream for transmitting arbitrary data in a peer-to-peer fashion. Besides video conferencing applications WebRTC and RTCDataChannels have been used for many other novel purposes, such as securely transmitting files between clients[5].

[5] `https://webtorrent.io/`

One important real-life use-case that the WebRTC standard and its implementations have so far not agreed yet is how signaling should be performed. Signaling in this context means the transfer of the necessary meta-data to enable two internet-connected devices to connect to each other. While this might seem simple, with the massive scale of the modern internet, clients can no longer be expected to directly reference each other via IP addresses due to firewalls and NATs. To solve this, a separate standard called Interactive Connectivity Establishment (ICE) is used. ICE has three different methods for attempting to connect clients. Besides attempting a direct IP-to-IP mapping, which is often the most efficient and secure way, STUN and TURN servers are generally the most often used solution.

In short, STUN (Session Traversal Utilities for NAT) servers allow a connected client to query for their public IP address behind a firewall. The STUN response will allow the WebRTC application to figure out its own public IP address, enabling it to offer this address to the remote client such that a connection can be formed.

In case the forming of the connection still fails after attempting to use STUN, the next alternative is utilizing TURN (Traversal Using Relays around NAT). With TURN, instead of attempting to connect to each other, the clients use a preconfigured central server to route data to each other. This has naturally some downsides, such as requiring quite a bit of computational resources from the server.

For our architecture, a third-party vendor by the name of Twilio was used[6]. Twilio offers an abstracted JavaScript library built around the basic WebRTC protocol, and also offers their own self-hosted TURN/STUN for more robust connectivity. Twilio was chosen among other competitors because it had a very robust free plan, and we already had previous experience in using it. Additionally, similar to lean start-ups, using a platform-as-a-service for forming the actual video calls between users allowed us to focus on the actual core innovations, rather than trying to implement this part ourselves.

3.4.3 Analytics aggregation pipeline

The actual pipeline for performing analysis on text and image data streamed from the front-end is structured in a scalable, event-driven way using RabbitMQ[7]. RabbitMQ is a fast, open-source message queue and broker with wide industry usage. As mentioned before, the front-end sends data such as

[6] `https://www.twilio.com/video`

[7] `https://www.rabbitmq.com/`

images or text to an initial brokering server, which then publishes this data to dedicated message queues. These message queues can be subscribed to by an arbitrary amount of clients, who then read the data and proceed to perform various operations, such as running natural language processing or image analysis on them.

This architecture allows us to easily add and experiment with new analysis software, since it is simply a matter of implementing a new microservice around a tool and using the existing RabbitMQ client library to subscribe to the right queue. Other advantage of the architecture in place is having the WebSocket-to-queue publisher be generic, in so far that it does not care about the data it handles, but simply passes it onwards to a configurable queue. Indeed, in the prototype the same code handles both video and textual data. One downside of this approach is that there is no validation of data received from the sockets, before it is published for the clients to analyze.

The WebSocket broker was written in the Go programming language. Go is a statically typed programming language originally developed at Google with relatively few keywords to its syntax. Go is used in many large-scale industry projects, such as the virtualization program Docker, and the container orchestration framework Kubernetes.

3.5 Image processing service

For the prototype, Google Cloud's image processing API called Google Cloud Vision was used for receiving and analysing video frames received from the browser. Other sentiment APIs, such as Microsoft's Emotion API[8] or Amazon Rekognition[9] were previewed, but Google's offering had the best free pricing tier and more robust features, which was import at the start, since we did not know in the beginning what kind of data would we end up using. Another option would have been to use our own custom machine-learning based solution, but that would have required a significant amount of pre-research, development and testing, which would have been out of scope for this thesis. The service offers a sizeable amount of information for image-based data, such as landmark detection and content labeling for search and analytical purposes, and facial sentiment analysis based on pre-trained machine learning models[10]. A different approach would also have been to use our own custom solution. The pros and cons of the evaluated tools can be seen

[8] `https://azure.microsoft.com/en-us/services/cognitive-services/emotion/`

[9] `https://aws.amazon.com/rekognition`

[10] `https://cloud.google.com/vision/`

Table 3.3: Comparison of facial recognition tools evaluated

	Pros	**Cons**
Emotion API	REST API Different emotions (6 different ones)	Closed source Lackluster free tiers
Cloud Vision	Cheap Good developer documentation Many features REST API Different emotions (6 different ones)	Closed source
Rekognition	Live tracking across multiple frames Different emotions (7 different ones)	Deeply vendor-locked into AWS No REST API Lackluster documentation Closed source
Custom model	No vendor lock-in Theoretically fully customizable 100% Open-source Affordable	Hard to find enough training data Need intricate knowledge of machine-learning

in Table 3.3. Within the scope of this thesis, mostly the facial sentiment analytics part of the API was used. As we will demonstrate, our infrastructure can easily scale to support multiple sources of analytics data, so we could implement other sentiment APIs with relatively little effort. Furthermore, benchmarking closed-source sentiment APIs by comparing their actual measured results is difficult, since emotions are quite subjective, so we did not spend much time comparing results given by Google Vision with its competitors.

A simple middleware program to read results from the RabbitMQ queue, gather insights from the Google Cloud Vision API, and send them onwards to another queue for further processing was implemented. From a given image, four different sentimental statistics of note are available. These are the likelihood of the person or people in the image displaying anger, joy, sorrow and surprise. These results are clamped to five different constants, ranging from a very unlikely display of emotion, to very likely. The reason these values are clamped to a certain numerical range, such that client application functionality will not ideally change if they re-train their model. The image processing service was also written in Go, since it had an official library from Google for interacting with their facial recognition API. In Figure 3.1 the image processing Google Cloud Vision-based microservice would be one of the consumers reading video-based data, analyzing it and passing the results onwards.

Table 3.4: Comparison of natural language processing tools evaluated

	Pros	**Cons**
Natural Language Understanding	Good sentiment support Well documented	Closed source
Cloud Natural Comprehend	Wealth of features Well documented	Closed source Still in beta Lackluster emotion
Comprehend	Good documentation	Closed source Lackluster emotion support

3.6 Language processing service

The second microservice for processing text-based data based on user speech is very similar in structure to the image processing service. The service simply reads data from a preconfigured message queue, analyzes it and writes the results to another queue. For processing language, IBM's natural language processing API called Watson NLP was used[11]. Watson NLP was chosen due to its generous free plan and wealth of features. Many of the other natural language processing libraries used did not support in-depth sentiment analysis, but rather a more simplistic positive/negative sentence classification, which rendered them unusable for our application. Their API was quite easy to get up and running. For the purpose of this thesis, we only needed the sentiment analysis functionality, however. The API takes in a phrase or group of sentences, and generates results based on that. The API then picks certain keywords from the input, and scores them according to five basic emotions. Similiar to the image recognition libraries, Google [12] and Amazon [13] also have their own offerings in this section, in the forms of Google Cloud Natural Language and Amazon Comprehend. A comparison of these options can be seen in Table 3.4.

The service was programmed in the Python programming language, using the official Watson API. This is one of the upsides of our microservice design, since it allows us to use the most optimal language for a given software component. In Figure 3.1 the Watson NLP API is one of the consumers reading text-based data measured from the front-end.

One design constraint placed by the Watson API is that sending results immediately as they are measured in the browser client leads to suboptimal results, since the API is idempodent, meaning that identical sentences always

[11] `https://cloud.ibm.com/apidocs/natural-language-understanding`
[12] `https://cloud.google.com/natural-language/`
[13] `https://aws.amazon.com/comprehend/`

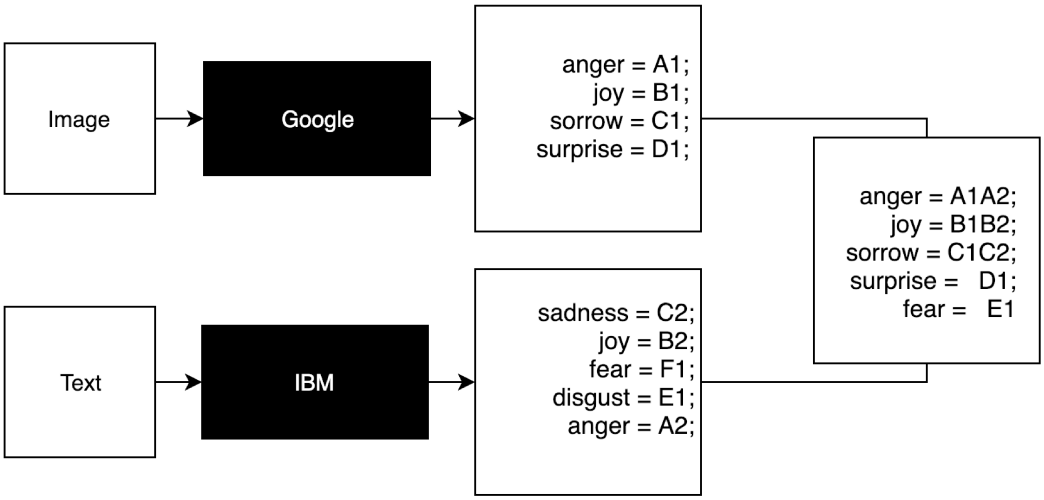


Figure 3.2: Diagram of our analytics pipeline

lead to the same measurements. This is because the API has no streaming support and does not persist knowledge on the actual conversation taking place. This was solved in the browser application by only flushing the data every five sentences, which seemed to work well after initial testing.

A rough diagram of our complete analytics pipeline can be seen in Figure 3.2. Due to the different measurements we receive from Google and IBM and their closed-source nature, we have to mix the two results together in the end somewhat. We deemed this to be an acceptable compromise, since quantifying human emotions is about abstracting things anyways, so adding one additional layer of abstraction more is an acceptable trade-off for the time being.

3.7 Data serialization format

All of the third-party external APIs mentioned so far use the JavaScript Object Notation (JSON) format as their standard data transfer format. As the name implies, JSON is a subset of JavaScript's standard object notation format, but standard libraries for handling JSON exist for most of the current programming languages out there, such as Go[14] and Python[15]. JSON is currently standardized by IETF under RFC 8259 [4].

The two main data structures that form the basis for JSON are unordered collections of key-value pairs represented by pairs of curly braces, and lists,

[14] `https://golang.org/pkg/encoding/json/`
[15] `https://docs.python.org/2/library/json.htmL`

Table 3.5: Data serialization benchmarks

	SBE	**Protobuf**	**MsgPack**	**FlatBuffers**	**JSON**
Code generation	Yes	Yes	No	Yes	No
Strong typing	Yes	Yes	No	Yes	No
RPC capabilities	Yes	Yes	No	No	No
Data size (bytes)	123	111	205	292	264
Encoding time	5.535 ms	5.437 ms	5.575 ms	5.553 ms	5.895 ms

represented by pairs of brackets. These two data structures can then contain three other data types, numbers, strings, booleans and also null values. An example of a JSON structure representing a person and their address and phone number data can be seen in Appendix A.1.

However, while JSON is easy to parse and read, it is also somewhat inefficient in terms of data transfer, being a text-based, human readable format. For example, in our application if we wanted to use JSON as the de facto data representation format, we would have to serialize and deserialize the data objects into UTF-8 encoded strings and then converting them into series of bytes before pushing them to the work queue. Then we would have to parse them again within the reader services. It is easy to see that using JSON leads to a non-neglible amount of wasted memory and computational power spent in this case.

While alternatives such as Binary JSON (BSON), as used by the MongoDB[16] exist, they also have the downside of being untyped and schema-free. In our particular use-case, it would be nice to guarantee the data integrity throughout the whole pipeline. Thankfully, a multitude of space-efficient, strongly-typed alternatives to JSON exist, such as MessagePack[17], Simple Binary Encoding[18] FlatBuffers[19] and Protobuf[20] exist. A comparison between these four data serialization formats can be seen in Table 3.5, along with ordinary JSON for comparison. All the benchmark results were generated using Go. For the reference JSON deserialization, the standard library JSON package was used[21]. Examples of data formats for Protobuf, Simple Binary Encoding and Flatbuffers can be found in Appendices A.2, A.3 and A.4.

From the results gathered, Protobuf generated the smallest final size for

[16] `http://bsonspec.org/`
[17] `https://msgpack.org/`
[18] `https://real-logic.github.io/simple-binary-encoding/`
[19] `https://google.github.io/flatbuffers/`
[20] `https://developers.google.com/protocol-buffers/`
[21] `https://golang.org/pkg/encoding/json/`

the example data structure given as input, with Simple Binary Encoding ended up generating the second-smallest results. Of note is that the data output by FlatBuffers was larger than the original data. This is most likely because FlatBuffers is more concerned with keeping the number of memory allocations done during serialization/deserialization as small as possible, and also to enable accessing data without deserialization at all, according to their website.

One important distinction is that the example data used to generate the results contains a sizeable amount of string data fields. Unlike numbers, these are somewhat more involved to compress. For numerical fields, one can easily perform bit-packing operations to store multiple values in a single byte, and then unroll them on decompression since the schema is known ahead of time, but string data usually requires more complex compression algorithms.

All the strongly-typed data serialization formats tested require their own tooling that is used to generate functions to serialize and then deserialize efficiently and safely, based on predetermined schema. Maintaining these files cause some extra cognitive overhead during the development process. Of note is that Simple Binary Encoding does not support variable-length data inside nested composite types currently,[22] which was very unfortunate for our use-case, since our data ended up being quite nested.

3.8 Data writer service

3.8.1 Overview of service

After the various analytics microservices have finished their work and written their results into a results queue, they are then read and processed by the writer service. Like the other microservices outlined within this thesis, the writer service is rather simple in it's functionality. It simply polls a predetermined set of message queues, listening for new messages. When a new measurement is generated the service parses the message and converts it into a format that can then be inserted into PostgreSQL. The writer service is also written in Go.

Besides just writing data, the service is also in charge of managing the database in other means, mainly related to schema migrations. In the realm of relational databases, migrations are incremental, reversible changes to the data schema. Utilizing migrations allows developers to adapt their data schema as requirements evolve and priorities shift during the development process, making them useful when developing software in an agile manner,

[22] `https://github.com/real-logic/simple-binary-encoding/issues/11`

where the final data format has not been totally designed up-front, in the very beginning of the software development cycle.

3.8.2 Choosing the right data store

Traditionally, relational databases have been the de facto solution for persisting data in a consistent, available manner. In relational databases, data is organized into tables based on predefined schemas with one or more columns of different attributes. These tables then contain rows of records, which can then be read, inserted, updated, deleted and otherwise manipulated using different query languages, most often SQL (Structured Query Language). Besides simple data operations, relational databases also offer a host of other useful data integrity features, such as constraints for the sake of data validation, or transactions for keeping the system state consistent at all times, even in the case of outages during updates concerning multiple queries or distributed databases. Relational databases are generally scaled vertically, which essentially means that the underlying hardware is updated by adding more RAM, hardware, CPU cores and such. Vertical scaling is easier to manage but is more costly than horizontal scaling due to the cheaper price of commodity hardware when bough in bulk. Generally, relational databases follow the rules of the mathematical discipline of relational algebra, which specifies the rules various tables can be joined together to form a end-result for a given query.

However, in recent times, with the growth of so-called Big Data, non-relational databases have also gained popularity. It has been noted that many useful features offered by traditional relational such as always keeping the data consistent and guaranteeing data integrity among multiple distributed database located in different clusters are difficult to scale for massive amounts of data and users. Therefore in recent years a wealth of alternative non-relational data store alternatives, often called NoSQL (short for Not Only SQL) have surfaced. NoSQL databases generally scale horizontally, meaning that the database load is distributed evenly across multiple machines by data sharding, replication or some other means to even the load. NoSQL databases can be categorized into four different kinds.

Key-value databases are the most simple NoSQL data stores, conceptually. Data is stored using simple hash tables, without any schema, enabling fast look-up based on those keys. Data can be trivially distributed among multiple nodes in a database cluster by getting the modulo of the key by the amount of nodes in the cluster and using it to determine which node in the cluster the data should be stored. Examples of such key-value databases are Redis and Amazon DynamoDB.

While traditional relational databases store data by row on disk, column-oriented databases instead opt to store them by column. Generally for running large queries these disk read times are the major bottle-neck, so storing data by columns rather than by rows allows for much better performance on certain kinds of queries that only need to access the columns of a given set of records, when computing the sum of all values of records for example. Columnar data storage also allows the data to be stored in a compressed format, since the data value contents will be uniform according to a given schema. Examples of columnar data stores are Cassandra and Riak.

Currently perhaps the most popular kind of NoSQL database are the various document-based databases out there. As the name might imply, data is stored in schema-free documents, which can be represented as serialized XML or JSON, to name a few. Document-based databases generally do not have strict schemas in place, and there is nothing to differentiate the data from the format like in traditional relational database systems, where tables and rows are conceptually completely different. This lack of a strict schema allows for fast turnout during development as the user does not have to spend time on schema maintenance, but it also has the downside of no guarantees of data format consistency that can be guaranteed with traditional relational database. Document-based databases store data internally as key-value pairs which allows for the same sharding benefits as when using plain key-value stores, but usually also offer a query language for performing more complex operations. Often used examples of document-based databases are out there are MongoDB and CouchDB.

The fourth and most rarely seen type of NoSQL database are graph databases such as Neo4j. While the underlying storage mechanism for graph databases varies by implementation, generally the interface exposed to the end-user abstract data as networks of graphs made of vertices and edges, instead of tables or collections. Graph databases are used for very specific purposes, such as modeling social media relationships or storing sections of the internet.

Ultimately, for our application we chose to use the open-source relational database PostgreSQL[23]. PostgreSQL had all the features we needed and more, with support for multi-table transactions, good performance and developer documentation and a wealth of data types. The other serious contender for the data store was the document-oriented MongoDB, but ultimately most of the data we used was very relational, so it made sense to use a relational database to that end. The various analytics data we aim to gather is strictly tied to the user from a data modelling point-of-view, and the users are then

[23] `https://www.postgresql.org/`

parts of a group, or session.

For improved performance, streaming master-slave replication was set up for the PostgreSQL cluster. In such a set-up, data writes are only allowed for the master database, and the changes are then replicated onto any number of slave nodes, which handle all of the reads. This is a commonly used deployment strategy because often in web applications reading data is much more common than writing data, so it makes sense to even out the read load between multiple nodes. Furthermore, generally read queries are more complex than write queries, since they might include expensive calculations and cartesian joins between multiple tables, while often writing data just includes inserting new rows into a table. Since we are mostly concerned with mostly collecting data within the scope of this thesis, this assumption does not completely apply, but naturally were this a real-world application we would want to extensively analyze the datasets we have collected in one way or another.

One interesting alternative that would have potentially worked quite well for our application would have been to use InfluxDB or a similar time-series database. Integrating InfluxDB into the current pipeline would be an interesting experiment, since one of the main use cases is storing real-time analytics, among other metrics. InfluxDB also has a large variety of premade dashboards for data visualization, which could be fruitful to look into. However, we still ended up using PostgreSQL nonetheless.

3.8.3 Modelling the data schema

Due to the groundwork laid out in the data serialization design phase in Chapter 3.7, converting the agreed upon Protobuf files into SQL tables was straightforward. Each model in Protobuf maps nearly one-to-one into a table in postgres. For example, the table mapping to a Protobuf model for a Google facial recognition results can be seen in Listing 3.3.

The only major design problem mapping Protobuf data models to SQL tables was deciding how to model the relationship between users, rooms, and measurements. Were this an actual application and not just a prototype, more granular access control and authentication for users and rooms would be needed, but for the purpose of this thesis we decided to include both the users and rooms as embedded structures in the measurement objects. Furthermore, to assist in aggregating and identifying user data, each username is unique within the whole database. In PostgreSQL, users and rooms are included as separate tables quite simple in their structure, containing only the name and identifier. Naturally, more design-work on the data schema should be done, if the prototype were made into a production-ready application.

Listing 3.3: SQL CREATE statement for google results table

```
CREATE TABLE google_results (
    id bigserial primary key,
    created_at TIMESTAMPTZ,
    detection_confidence DECIMAL(4) NOT NULL,
    blurred DECIMAL(4) NOT NULL,
    joy DECIMAL(4) NOT NULL,
    sorrow DECIMAL(4) NOT NULL,
    surprise DECIMAL(4) NOT NULL,
    image bytea NOT NULL,
    user_id integer REFERENCES users(id) NOT NULL
);
```

3.9 Deploying the application

To orchestrate the various different microservices and their dependancies, the virtualization platform Docker and Docker Compose were used. Docker allows for running applications and processes inside light-weight virtualized containers that are isolated from each other and the host systems unless otherwise explicitly specified, and Docker Compose allows users to define how they interact with each other in terms of networking and various other shared resources. To define the contents of these containers the Docker user must write Docker images, which contains step-by-step instructions on how the required environment needed for the application to run are provisioned. The user can either download pre-built images from the internet, or build them themselves. Additionally, many cloud providers such as Amazon Web Services allows users to upload their own Docker images and run them, with additional features such as monitoring tools, load-balancing and autoscaling of containers based on the current load. However, for the prototype we implemented we just used the Docker Compose tool running on a rented private virtual server. In our architecture diagram seen in Figure 3.1, all of the separate services are deployed inside their own Docker containers, with all their interfaces and dependancies on each other specified out in the Docker Compose configuration file.

An example of a Docker image for the image recognition service implemented can be found in Listing 3.4. Within the Dockerfile we simply download the required third-party dependancies using the go get command of the Go compiler, and then we build the binary. The last line specifies that upon starting the container, the entry-point of the application is the binary we

Listing 3.4: Dockerfile for image recognition service

```
FROM golang:1.10

ADD . /go/src
WORKDIR /go/src
RUN go get ./...
RUN go build -o main ./src/main.go
CMD ["/go/src/main"]
```

built on the previous line.

Docker Compose is a tool for managing dependancies between multiple containers, along with their configurations. A snippet from the deployment manifest for the entire microservice cluster can be found in Listing 3.5. In the example, three different services are specified. The RabbitMQ service is pulled from the official Docker registry upon starting the application. The Docker registry is a sort of content delivery system, where developers are free to upload their own base images for other people to use as a basis for their own. This allows a faster turnaround for developers, since they can spend less time setting up their application environment, since each container built using Docker an image will alway be the same.

The two WebSocket broker images configured in Listing 3.5 use the same base image that is built locally. The brokers, called image_publish and text_publish are linked to the RabbitMQ container using the links declaration. By default, Compose automatically sets up a private network between the containers, allowing them to access each other via hostnames. Compose also allows users to pass arbitrary environment variables to the containers, which are used here to customize the type of data that the WebSocket brokers handle. One more important thing that should be specified explicitly when using Compose is the ports that are mapped from within the cluster to the host machine. Without the two ports declarations under the brokers, they could not actually access the outside world from within the container. When using Docker Compose, the whole microservice-based application can be deployed easily using just one command, both when developing the application and also when deploying it to production.

Listing 3.5: Docker compose configuration example

```
version: '3'
services:
  rabbitmq:
    image: "rabbitmq:latest"
    ports:
      - 5672

  image_publish:
    restart: on-failure:10
    image: "publish:latest"
    ports:
      - "8080:8080"
    links:
      - rabbitmq
    environment:
      - RABBITMQ_ADDRESS=amqp://rabbitmq:5672
      - APP_HOST=:8080

  text_publish:
    restart: on-failure:10
    image: "publish:latest"
    ports:
      - "8084:8084"
    links:
      - rabbitmq
    environment:
      - RABBITMQ_EXHANGE=text
      - RABBITMQ_ADDRESS=amqp://rabbitmq:5672
      - RABBITMQ_CONTENTTYPE=text/plain
      - APP_HOST=:8084
```

Chapter 4

Implementation

4.1 Browser client

4.1.1 Application structure

Being a prototype with the actual data pipeline consisting of the bulk of the research work, the actual browser application structure is relatively simple, consisting of only two views. Upon first visiting the application, the user is shown a form where they can input their name and the teleconference room they wish to partake in. When the user clicks the login button, an authentication request is made to a backend server, which generates a cryptographically signed token for the given username, allowing the user to join the conversation and move onto the actual teleconferencing view. In the teleconferencing view, the user can converse with others as can be expected of such an application, while the sentiment data gathering is performed in the background. A screen capture of the teleconferencing view can be seen in Figure 4.1. In accordance with agile prototyping, the first versions of the prototype did not contain any authentication screen or styling, since it was not strictly necessary at the very beginning.

4.1.2 React and TypeScript

The web application was written in the TypeScript programming language, utilizing the React framework. TypeScript is a strongly typed superset of the JavaScript programming language, adding many additional features such as a substructural type system from modern statically typed languages to JavaScript[1]. TypeScript compiles down to regular JavaScript, allowing it to

[1] `https://www.typescriptlang.org/`

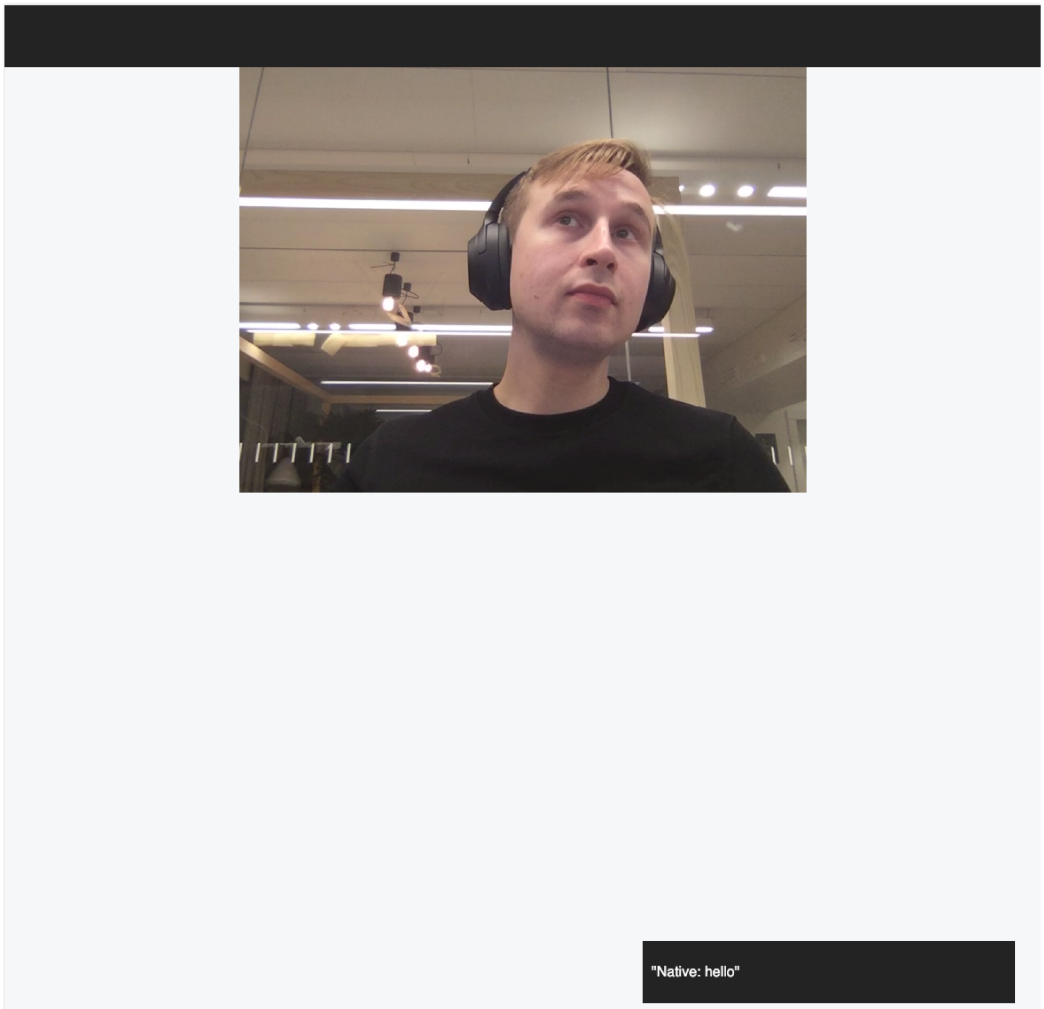


Figure 4.1: Video teleconferencing view with one participant

be run in all browsers.

React is a declarative component-based framework for creating user interfaces. In React, user-written components must always expose a single render method, which determines the HTML the component should render based on its state. Besides plain HTML, React components can also render other nested React components. Components have internal, private state, but they can also be passed on properties from their parent components, enabling more granular composition and re-use. Whenever a state change is detected, React uses a sophisticated diffing algorithm to compute the new DOM model from the state mutations[2]. This makes it easy and transparent for the programmer to always keep the user interface synchronized with the application state.

An example React component written in plain JavaScript that renders a single HTML span tag, the content of which is dynamically computed based on the name attribute passed onto the component from its parent can be seen in Listing 4.1. The same exact component, but written in TypeScript can be seen in Listing 4.2. Of note is that the JavaScript version will accept anything as the name attribute, since JavaScript is a weakly typed language, but the TypeScript compiler will raise an error if the name attribute is not explicitly passed in as a string. While this particular example is trivial to verify, it is easy to see why TypeScript might help avoid program errors in more complex cases components, with lots of nested data and properties passed around.

[2] `https://reactjs.org/docs/reconciliation.html`

Listing 4.1: Example React component in JavaScript

```
class HelloComponent extends React.Component {
  render() {
    return <span>Hello {this.props.name}</span>
  }
}

<MyComponent name="Bob" />; // <span>Hello Bob</span>
<MyComponent name={0} />; // <span> Hello 0 </span>
<MyComponent />; // <span> Hello undefined </span>
```

Listing 4.2: Example React component in TypeScript

```
interface Props {
  name: string;
}

class HelloComponent extends React.Component<Props, {}> {
  render() {
    return <span>Hello {this.props.name}</span>
  }
}

<MyComponent name="Bob" />; // <span>Hello Bob</span>
<MyComponent name={0} />; // Error at compile-time
```

One important thing of note is that the TypeScript compiler is only as useful as the typings provided to it. Many third-party libraries do not have proper type definitions, and since TypeScript's type-checking is only performed at compile-time, without being provided a type definition file for a given module or library the compiler cannot infer the shape of the data correctly. Quite often a web application such as the one described here have to be integrated with many third-party APIs. Since TypeScript itself can only guarantee run-time type safety, we can not leverage it to type-check data fetched from these external sources at run-time. To solve this, a third party library by the name of io-ts[3] was used. Io-ts enhances TypeScript with a runtime type system, since ordinary TypeScript only type-checks during compile time.

[3] `https://github.com/gcanti/io-ts`

4.1.3 Browser analytics collecting

During the teleconference, the user camera stream is tied to a standard HTML5 [9] video element. To capture video frames in order to run various facial recognition, every five seconds the current frame in the video element is captured and rendered to as a HTML5 canvas element, which is a low-level element used for drawing 2D graphic primitives. This canvas is then serialized, containing the current frame as a JPEG-encoded binary blob, which is sent to the analytics pipeline via a WebSocket connection, as was previously described.

To transliterate speech into a textual representation such that sentiment analysis can be performed, two different alternatives were implemented and then compared. The experimental Web Speech API [14] was the first thing we tried. This API is currently only supported on the latest versions of Google Chrome, but it offers a convenient and somewhat accurate way of transforming speech to text. The other alternative was IBMs commercial alternative, called Watson Speech to Text[4]. The API surface is identical to the proposed Web Speech API, IBM's API just uses their own proprietary backend for transliterating speech into text.

In order to perform a comparison between the native Web Speech API of the browser, and IBM's speech-to-text offering, a list of sentences called the Harvard sentences were used [16]. The Harvard sentences list is a collection of phrases originally meant to ease the standardized testing of voice transmitting systems. A YouTube video of a female voice reading a list of the aforementioned sentences was downloaded, and fed both into the Web Speech API via a simulated microphone, and sent straight to IBM's backend server. Surprisingly, the output was very similar for both the Web Speech API and IBM's Text to Speech API, so we ended up using the native, standardized and freely available Web Speech API, so we ended up using the native, standardized and freely available Web Speech API.

In the first iterations of the analytics pipeline, there were only two backend microservices, one handling text data and one images. Both services handled everything from receiving the data to analyzing it, and sending it back to the client. This was done to speed up the development workflow in the early phases, since we had not yet decided on how we would want to store the data, and this way we could visualize immediate feedback to the user based on the data received from the sentiment APIs, which aided in debugging the system.

Before starting to design the visualizations and the final backend archi-

[4] `https://www.ibm.com/watson/services/speech-to-text/`

Table 4.1: Comparison of sentiment measurements

Google	**Watson**	**Summarized emotion**
Joy	Joy	Joy
Sorrow	Sadness	Sadness
Anger	Anger	Anger
Surpise	N/A	Surprise
N/A	Disgust	Disgust
N/A	Fear	Fear

tecture, one important design decision that had to be made beforehand was whether or not to combine data from the two different sources we had. As mentioned earlier, we received four different emotions from Google's facial recognition API, and five different emotions from Watson's NLP API. Since there is no visibility on how the services used generate these results, some design decision had to be made regarding whether or not the results from the different services could be compared together. In the end, it was decided that we would mix the results together for the sake of the visualization, since quantizing emotions into numbers is a large enough abstraction as is, so it stands that most likely the end-user would not care about such nuances. The APIs and their corresponding sentiment labels can be seen in table 4.1.

4.2 Visualization of sentiment data

4.2.1 Planning

Once the pipeline for gathering the sentiment data had been implemented, we designed and implemented a simple single-page visualization, which the user of the application could use for self-reflection after a session. Besides visualizing the user's own emotions, we also decided to visualize the emotions of the group, or room of people at a given time side-by-side. This would allow the user to compare and contrast their experienced emotions with the ones displayed by others, leading to new opportunities for reflection. Of couse, care had to be taken to display the data in such a way as not to compromise a single individual's privacy, at least to a certain extent. By happenstance, the emotions we gathered also coincide with Ekman's theory of six basic emotions [7].

After initial design meetings, a rough sketch of the visualization was created. It was decided that the data would be visualized as three different separate sections, divided into two columns. The leftmost column would

visualize the individual's own emotions, and the rightmost would show the entire group's emotions. During the design process, we tried to carefully consider the form the feedback would be given, as to provide the most neutral and subjective view of the emotional status, with minimal bias towards some emotions over others.

In Figure 4.2 you can see the final implementation of the visualizations. The topmost visualization is a radar, or spider chart for the average of all the emotions experienced during the session. The radar chart has six different axis, quantifying the six different emotions that were laid out earlier. Next up, we envisioned there would be a word cloud of all the expressions used during the session for both the individual and the user, with the size of the text pertaining to the impact the expression had on the conversation. Below the word cloud are the emotions visualized as a Marimekko chart. A Marimekko chart is a type of bar chart, where each bar is of a variable width. In our case the horizontal axis is time, and the vertical axis is a column of all the emotions averaged from a certain time span. According to our initial estimates this would be a useful way of visualizing data, since it would be not likely that a single measurement would contain large values for many emotions at once. It is unlikely for an individual to express large amounts of both anger and joy at the same time, for example.

4.2.2 Implementation

Based on these initial designs, a prototype web-based visualization was quickly created using HTML, CSS and JavaScript. For most of the visualisations, the Javascript framework D3 was used[5]. Besides D3 the functional programming library Ramda[6] was heavily used, since most of the actual business logic of the visualization had to do with transforming immutable data from one format into another, which functional programming paradigms are well suited for. Along with the frontend, a simple REST API for serving the data was also implemented. It simply queries the main database that is written by the writer process on demand and sends it back to the client to be visualized. Both the implementation and design process were quite iterative throughout. The first visualizations were simple line graphs, generated using random data points, before the actual data was ready to be visualized. In the end, we did not go through with line graphs, since the end results were somewhat difficult to read when we attempted to visualize multiple emotions.

One other small but important design choice that had to be made during

[5] `https://d3js.org/`
[6] `https://ramdajs.com/`

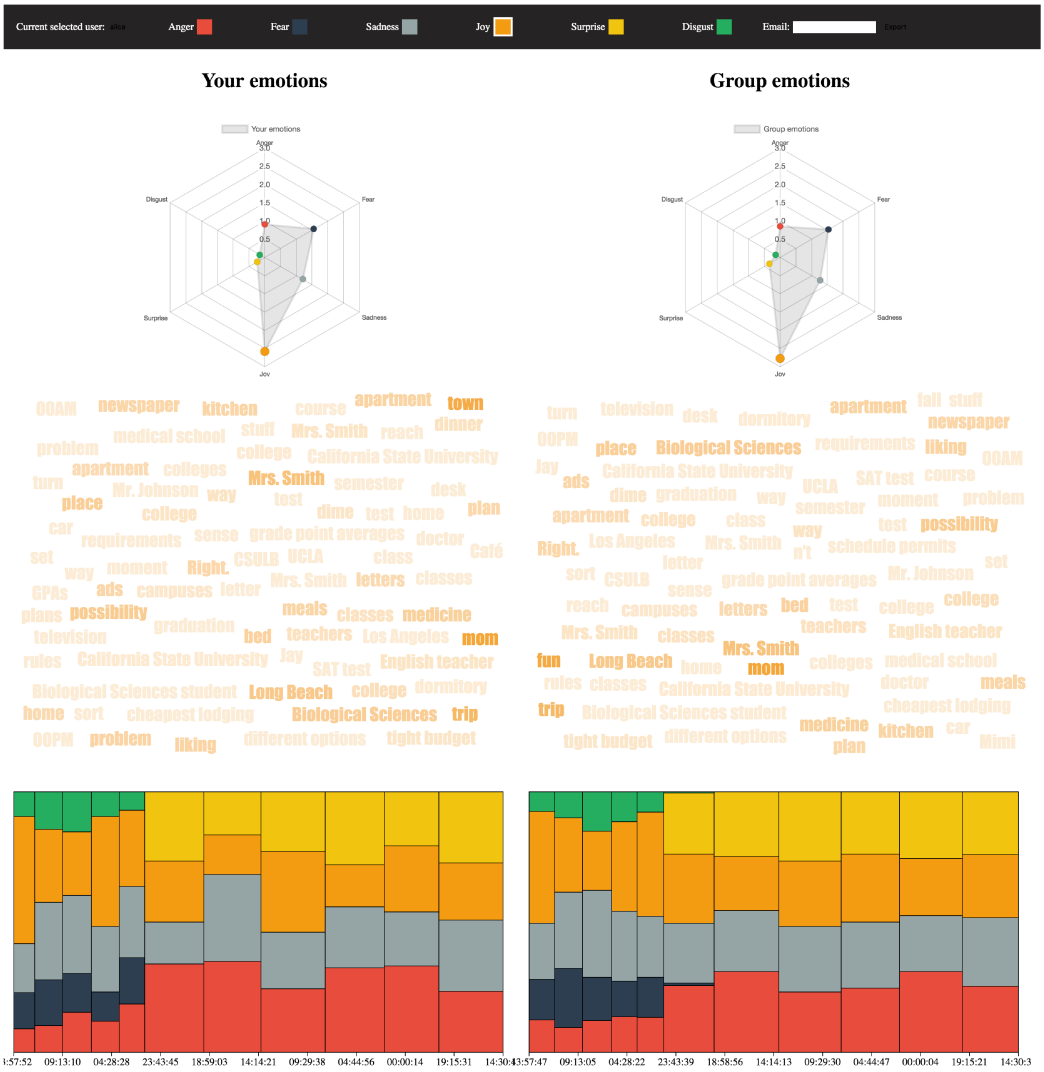


Figure 4.2: Screenshot of the final visualization

the actual implementation that was not immediately apparent in the design phase was to choose the colors for the various emotions. Six different colors were chosen [20]. The color chosen for anger was red, the color chosen for fear was a deep midnight blue, sadness was a dark gray, joy was a bright orange, surprise was yellow and disgust was green. The colors themselves were chosen from the service Flat UI Colors[7], that offers various color palettes that match well together.

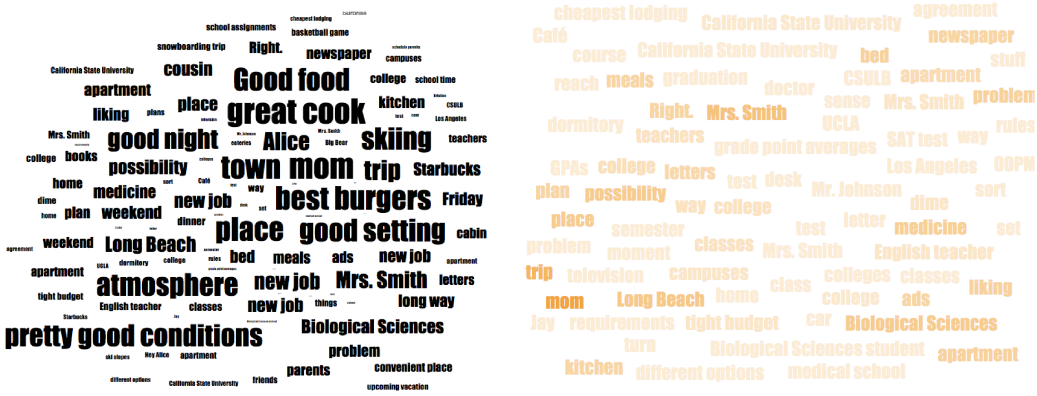
The only major new addition from the design was the header on the top of the page. The header contains a list of colored buttons which can be used to choose which emotions are highlighted for each of the visualizations. In the very first implementations, it was possible to control separately which emotions were emphasized for the individual and the user, but this feature was ultimately discarded due to the complexity of the user interface. Additionally, the header contains a dropdown that can be used to select the currently active user, for debugging purposes. Additionally, some visual style changes for print were applied with additional CSS, to make the page layout flow in a more structured manner.

The radar chart implementation ended up being quite similar to what was originally planned. The charting library Chart.js[8] had a plug-and-play radar chart that was relatively easy to implement. The only new addition to the mock-up was emphasizing the currently selected emotion by making the dot larger.

The word cloud ended up going under a couple of different revisions. From the beginning, it was clear that it would be quite useful to visualize the various words used in the session grouped by the emotions elicited. In the first, monocolored revision, the size of the words in the cloud were used to indicate the emotional impact the word had on the conversation. However, traditionally when utilizing word clouds the size of the word directly corresponds to the amount of times the word was used. Therefore, in the end it was decided that to avoid confusion the words would be kept as the same size, but the opacity of the color would be used to indicate its impact. The initial black-and-white word cloud design can be seen in Figure 4.3a and the new one can be seen in Figure 4.3b. The data fed to the word clouds seen in the figures are based on English teaching material where two people converse with each other.

[7] `https://flatuicolors.com/`

[8] `https://www.chartjs.org/docs/latest/charts/radar.html`



(a) Initial word cloud design (b) Final word cloud design

Table 4.2: Lines of code per programming language for prototype

Files	Language	Lines of Code
20	JavaScript	6680
33	TypeScript	1139
10	Go	686
18	SQL	415
3	HTML	396
2	Python	361
8	Sass	127
1	YAML	79
1	Protocol Buffers	44
8	Bourne Shell	41
5	Dockerfile	33

4.3 Overview of implementation

Table 4.2 contains an overview of the programming languages used in the final microservices. Most of the code was written in JavaScript and TypeScript, since these were used for both the frontend web application, and various simple authorization services for generating API tokens. Go was the third most used programming language, with most of the backend being written in it. SQL was the fourth, with the data migration statements taking up a sizeable amount. This data was generated using the CLOC tool[9]. All of the code was stored in GitHub, with git being the version control system chosen.

[9] `http://cloc.sourceforge.net/`

Chapter 5

Discussion

5.1 Discussion and evaluation

Overall, the structure of the demo application we implemented is functional based on end-user testing, and the microservice pattern described in Chapter 3.3 ended up working well for us. Of course, there is always a certain degree of uncertainty when attempting to quantify human emotion, and in the current system, we feel that the two largest factors of uncertainty are the transcribing of spoken word to a textual format as described in Chapter 4.1.3, and the overall transcribing of emotions from text and image data. Unfortunately, we could not find any public documented sources on how exactly Google and IBM actually perform sentiment analysis. While obscuring this might make sense for them from a business perspective, it makes evaluating the validity of the system difficult. All in all, the APIs currently publicly available for image and speech analysis are quite good, although they still have problems detecting subtle nuances in speech, like jokes or sarcasm. It is to be expected that as demand and interest in machine learning and automation rises, readily available for them will become even better. It would be interesting to investigate into potential methods to evaluate the actual accuracy of the data that is measured by the tool. Links to the publicly available source code repositories for all the software described in this thesis can be found in Appendix A.1.

For the backend pipeline, from a purely technical point-of-view, it would be benefitial to implement centralized logging, since in the demo applicaton there is nothing like that available, simply sending things to terminal standard output. We would recommend looking into Kibana and Logstash. Protobuf ended up being great for us, being quite easy to use and allowed us

to easily guarantee type safety throughout the various microservices. PostgreSQL also worked quite adequately for our needs.

Reflecting on the functional and non-functional requirements we gathered in 3.2, we can see that we managed to fulfill all of the initially planned functional requirements to at least some degree. The system successfully gathers data from live video and audio, and persists them to our database. The visualizations were also implemented to a satisfactory degree, based on some small-scale testing with users, although there are still some improvements that could be made. As for the non-functional requirements, some improvements could be made. One of our requirements was that the system should work well on all the latest browsers, but due to time contraints we only tested it properly on Google Chrome. From a technical point of view, all the features should work on browsers such as Firefox or Safari, with the notable exception of the Web Speech API. However, we tested the Watson Speech-to-text implementation during the development process, and found out that it also works reasonably accurately, leading to more or less the same results as the native browser transcribing. Watson's implementation also works on all the latest browsers that support the WebSocket protocol. Other browser features used for the web application and visualization were quite standard, such as the new async/await syntax[1] and they could easily be made to support all the latest browsers from the major vendors if we so wanted to. Utilizing an agile iterative process during the development ended up working well for us, speaking from a purely subjective point of view.

As was outlined in Chapter 2, the original reason for developing the prototype was to create a computer-based tool to facilitate reflection based on video and audio data, without requiring any feedback from the user. This is one other area that the current prototype needs to be improved upon. In order to properly evaluate and benchmark this, we would need to perform quite extensive testing with users, since reflection is a long-term process. Furthermore, we could not reliably evaluate how factual the actual sentiment measurements were, which was one of our original research questions. The data we receive from Watson and Google seem to make sense from a purely subjective point-of-view. As was stated previously, quantifying human emotions is quite difficult, and only so much accurate data can be gleamed from web camera screenshots and speech-to-text transliterations. This is the one point where we feel that the prototype, and henceforth this thesis, falls short. Were this project to be continued, performing such a study would be critical.

For the anonymization requirement, we feel that we accomplished it some-

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function`

what well. As was pointed out in Chapter 4, we currently just collect the users username, not much else, with the rest of the data being just the sentiment data that is necessary for us. However, the demo application is quite trivial, since it has no authentication nor authorization, so not much actual design effort had to be put into it. Were this an actual application used by real people, much more effort should be made into the ethical and potentially legal aspects of the project.

Reflecting back onto the original research questions laid out in Chapter 1.1, we can see that there are quite a variety of emotions that can be detected, as was discovered in Chapter 3.5 and 3.6. However, the APIs outlined here often fail to capture subtle microexpressions, and all but Amazon's facial recognition API has the built-in functionality for tracking emotions across longer timespans, which leads obviously to better results if the source is a live video stream. As for the other technology choiches, like previously said, the technology stack we chose from the front-end framework to the backend architecture is solid throughout, although there will always be room for improvement. Furthermore, our original research question specifically specified open-source should be looked into. Surprisingly, it turns out, based on the research laid out in previous chapters, that most of the best sentinment analysis tools from a purely technical point-of-view were closed source and ran and hosted by large companies like Google or Amazon.

5.2 Future developments

For future development of the project, it would be interesting to integrate more tools for generating insights into the analytics pipeline, since the current design is quite composable. Also, it would be very interesting to generate a large amount of real-life data, and attempt to train a machine-learning models based on that data to potentially allow individuals to predict their emotional reactions beforehand. Furthermore, it would be worthwhile to look into other ways of gathering sentiment analytics besides images and content of speech. For example, the pitch, tone and rate of speech could possibly be fruitful sources of data. There are some interesting new developments in the field of facial recognition by Pietikäinen et al. on novel methods of detecting microexpressions from video footage [36], which would be quite useful for our needs and purposes. The recognition systems we evaluated could only handle obvious changes in facial expressions, not subtle, unconscious ones. Additionally, Pietikäinen et al. has some interesting research on detecting heartbeats from minute fluctuations in RGB video footage [5], which would provide useful to integrate into our system. Besides that, it would be very

interesting to integrate eye tracking functionality into the actual application, to determine which participants the end-user looks at while feeling certain emotions. Additionally, actions performed by the user in-browser could also be tracked and analyzed, such as the time the user spends outside of the tab that application is running in, or where the mouse cursor of the user is located at. Furthermore, more care should be taken into how the measurements in the visualization are calculated, as the current method favours facial expressions over speech somewhat, due to the better availability of data, as in large conference calls people tend to listen more than to talk.

It would also be intriguing to refactor the core logic of the browser analytics collection into a separate library, which would potentially allow people to integrate the pipeline into their own existing video-based applications. This might even make for an interesting business venture or startup idea.

Looking at the actual analytics, in our current implementation we only look at the emotions experienced by an individual, and the emotions experienced by the group. Besides this, it would also be interesting to analyze and visualize how the words and mannerisms of an individual affects other people. We feel that this would be very helpful for self-reflection, as you could potentially see how the way you act affects the moods of other people.

One interesting aspect that was not looked at in-depth in this thesis were the obvious ethical problems that might arise from attempting to quantify and analyze such deeply human things as emotions on a large scale. Besides the evident privacy problems that had to be taken into account during the development of the data visualization part of the thesis, there is also great potential for abuse of a tool like the one described in this thesis. Indeed, as of writing the Chinese government is working on a big-data analysis powered social credit system mass surveillance system to rate its citizens and their reputation. It is easy to see how a non-intrusive and easy-to-use video-based tool such as the one demonstrated here might be employed by a government or similar agency to gather intelligence on its citizens or other parties. Besides government surveillance, the tool, or similar technology could also potentially be abused by private companies. However, it is important to keep in mind that when it comes to private companies there is really no way to know if they are not already using such technologies. A compelling angle of approach for future development would be look into how tools based on machine-learning such as the one developed should be best applied from a purely ethical point of view, in a fair and unbiased manner.

By using free social media platforms such as Facebook or Instagram, people have implicitly agreed that their data would be used for advertising, for example, but this data has only concerned factual points as far as we know. For a substantial majority of cases people do not make their purchasing de-

cisions based on logic but their emotions, it is not difficult to envision that advertisers would be very keen on having a deeper insight into the current emotional state of mind of a consumer for extremely targeted advertising. Recently, it came to light that Facebook had already sold a massive amount of private personal data to advertising agencies[2]. There has been a notable increase of interest in personal privacy by the public with the advent of the EU General Data Protection Regulation, and it stands to reason that people would consider their emotional state-of-mind to be even more private than strict facts. In 2014, social scientist Shoshana Zuboff predicted the rise of so-called surveillance capitalism [38] in the near future. According to her, data collection and procession for targeted advertising and personalization will soon become one of the core tenets of all large corporations.

Quantifying complex phenomena such as health, productivity and emotions are inherently reductivistic with the current available technology, and one should be very careful when attempting to do so. For instance, a person's current health is not a based on a formula based on your heartrate and weight, and we should be extremely careful to avoid abstracting them too much, less we lose some crucial elements in the process. An apt label used for the phenomena where people become fixated on this gamified, simplified view is called "data fetishism" [31]. Besides arguments against reducing phenomena to simplistic numbers, self-reflection tools like the one outlined in this thesis have also been reported to replace less easily quantifiable metrics. One instance of such is reported by Lepton [23], who argues that women who use mobile applications for tracking their menstrual cycle believe the feedback given by the application more than their own bodies. Unfortunately, we did not have much time to dedicate to evaluating and designing this while implementing the application and visualization, but we feel doing so would be extremely crucial, less we influence people in the wrong manner.

[2]https://www.cnbc.com/2018/03/22/facebook-data-scandal-commerzbank-and-mozilla-pull-advertising.html

Chapter 6

Conclusions

Initially, the aim for this thesis was to to evaluate the tooling and technologies currently available for gathering sentiment analysis, both commercial and open-source, and to utilize these libraries to create a prototype of a tool for assisting people in self-reflection during teleconferencing. To achieve this, a browser-based video call application integrated into a sophisticated, scalable analytics pipeline was designed and implemented, as was described in detail in Chapters 3.3 and 4. The analytics pipeline was implemented using a microservice model, and developed using agile methodologies (Chapter 3.2). A browser-based web visualization was developed to aid people in exploring, reflecting and improving upon on their emotional state in their day-to-day lives, Ultimately, we found that the available tooling now for facial analysis and natural language processing among other tools is already quite robust and usable, although open-source tooling is a bit lacking. All in all, there is a definite level of abstraction between the emotions an individual truly feels and the data that a computer program can easily quantify and calculate. All things considered, we fully expect these results to improve over time as technology advances and machine-learning tools become more and more sophisticated and ingrained into our daily lives. How far-reaching these changes will be remains to be seen, however.

References

[1] Adams, K. *Nonfunctional requirements in systems analysis and design.* Springer, Cham, 2015.

[2] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. Manifesto for agile software development, 2001.

[3] Benington, H. D. Production of large computer programs. *Annals of the History of Computing 5*, 4 (Oct 1983), 350–361.

[4] Bray, T. The javascript object notation (JSON) data interchange format. STD 90, RFC Editor, December 2017.

[5] Chen, J., Chang, Z., Qiu, Q., Li, X., Sapiro, G., Bronstein, A. M., and Pietikäinen, M. Realsense = real heart rate: Illumination invariant heart rate estimation from videos. In *IPTA* (2016), IEEE, pp. 1–6.

[6] Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering.* Springer, 2017, pp. 195–216.

[7] Ekman, P. An argument for basic emotions. *Cognition and Emotion* (1992), 169–200.

[8] Engelbart, D. C., and English, W. K. A research center for augmenting human intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (New York, NY, USA, 1968), AFIPS '68 (Fall, part I), ACM, pp. 395–410.

[9] Faulkner, S., Pfeiffer, S., Navara, E. D., Berjon, R., Hickson, I., O'Connor, T., and Leithead, T. HTML5. W3C recommendation, W3C, Oct. 2014. http://www.w3.org/TR/2014/REC-html5-20141028/.

[10] Fette, I., and Melnikov, A. The websocket protocol. RFC 6455, RFC Editor, December 2011. `http://www.rfc-editor.org/rfc/rfc6455.txt`.

[11] Fontanini, G., Bertini, M., and Del Bimbo, A. Web video popularity prediction using sentiment and content visual features. In *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval* (New York, NY, USA, 2016), ICMR '16, ACM, pp. 289–292.

[12] Freeman, E., Freeman, E., Sierra, K., and Bates, B. *Head First Design Patterns*. O'Reilly, 2004.

[13] Fulton, R. *Airborne Electronic Hardware Design Assurance : a Practitioner's Guide to RTCA/DO-254*. CRC Press, City, 2017.

[14] Glen Shires, H. W. Web speech api specification, October 2012.

[15] Grigar, D. Audio culture: Readings in modern music edited by christoph cox and daniel warnercontinuum press, new york, ny, 2004. 472 pp. trade, paper. isbn: 0-8264-1614-4; isbn: 0-8264-1615-2. *Leonardo 38*, 4 (2005), 357–358.

[16] IEEE recommended pratice for speech quality measurements. *IEEE Transactions on Audio and Electroacoustics 17*, 3 (September 1969), 225–246.

[17] Hickson, I. Webrtc 1.0: Real-time communication between browsers, September 2018.

[18] Kahneman, D. *Thinking, fast and slow*. Farrar, Straus and Giroux, New York, 2011.

[19] Kaushik, L., Sangwan, A., and Hansen, J. H. L. Automatic sentiment extraction from youtube videos. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding* (Dec 2013), pp. 239–244.

[20] Koponen, J. *Tieto näkyväksi : informaatiomuotoilun perusteet*. Aaltoyliopisto, Helsinki, 2016.

[21] Leinonen, T., Keune, A., Veermans, M., and Toikkanen, T. Mobile apps for reflection in learning: A design research in k-12 education. *British Journal of Educational Technology, 47(1)* (2016), 184–202.

[22] Li, I., Dey, A. K., and Forlizzi, J. Understanding my data, myself: Supporting self-reflection with ubicomp technologies. In *Proceedings of the 13th International Conference on Ubiquitous Computing* (New York, NY, USA, 2011), UbiComp '11, ACM, pp. 405–414.

[23] Lupton, D. Quantified sex: a critical analysis of sexual and reproductive self-tracking using apps. *Culture, Health & Sexuality 17*, 4 (2015), 440–453. PMID: 24917459.

[24] Lupton, D. *You are Your Data: Self-Tracking Practices and Concepts of Data*. Springer Fachmedien Wiesbaden, Wiesbaden, 2016, pp. 61–79.

[25] Mayer-Patel, K. *MediaSynch Issues for Computer-Supported Cooperative Work*. Springer International Publishing, Cham, 2018, pp. 191–208.

[26] Michael, M. M., Moreira, J. E., Shiloach, D., and Wisniewski, R. W. Scale-up x scale-out: A case study using nutch/lucene. In *IPDPS* (2007), IEEE, pp. 1–8.

[27] Poppendieck, M., and Poppendieck, T. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, Boston, MA, 2003.

[28] Ries, E. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. Crown Business, 2011.

[29] Rodden, T. A survey of CSCW systems. *Interacting with Computers 3* (1992), 319–353.

[30] Schon, D. A., and DeSanctis, V. The reflective practitioner: How professionals think in action. *The Journal of Continuing Higher Education 34*, 3 (1986), 29–30.

[31] Sharon, T., and Zandbergen, D. From data fetishism to quantifying selves: Self-tracking practices and the other values of data. *New Media & Society 19* (2017), 1695–1709.

[32] Sánchez-Rada, J. F. Onyx ontology specification. 2015.

[33] VNI, C. Cisco Visual Networking Index: Forecast and Trends, 2017–2022. Tech. rep., Cisco Systems, Inc, 11 2018.

[34] Vygotskii, L. S. *Mind in society : the development of higher psychological processes / L. S. Vygotsky ; edited by Michael Cole ... [et al.].* Harvard University Press Cambridge, 1930-1934/1978.

[35] Westerski, A. Marl ontology specification. 2011.

[36] X., L., X., H., A., M., X., H., T., P., G., Z., and M., P. Towards reading hidden emotions: A comparative study of spontaneous micro-expression spotting. and recognition methods. *IEEE Transactions on Affective Computing, in press (available online)* (2017).

[37] Yoon, S., and Pavlovic, V. Sentiment flow for video interestingness prediction. In *Proceedings of the 1st ACM International Workshop on Human Centered Event Understanding from Multimedia* (New York, NY, USA, 2014), HuEvent '14, ACM, pp. 29–34.

[38] Zuboff, S. Big other: Surveillance capitalism and the prospects of an information civilization. *Journal of Information Technology 30* (03 2015).

Appendix A

Appendix

A.1 Links to source code

LaTeX source for this thesis	https://github.com/melonmanchan/dippa
Twilio token service	https://github.com/melonmanchan/dippa-auth
Watson token service	https://github.com/melonmanchan/dippa-watsonauth
Websocket publishing service	https://github.com/melonmanchan/dippa-publish
Protobuf model definitions	https://github.com/melonmanchan/dippa-proto
Google face recognition service	https://github.com/melonmanchan/dippa-facerec
Watson NLP service	https://github.com/melonmanchan/dippa-nlp
Docker configuration files	https://github.com/melonmanchan/dippa-docker
Browser application	https://github.com/melonmanchan/dippa-front
Data writer service	https://github.com/melonmanchan/dippa-writer
Data visualisation	https://github.com/melonmanchan/dippa-vis

A.2 Data serialization format examples

Listing A.1: Example of JSON

```
{
  "firstname": "john",
  "lastname": "smith",
  "age": 25,
  "address": {
    "streetaddress": "21 2nd street",
    "city": "new york",
    "state": "ny",
    "postalcode": "10021"
  },
  "phonenumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

Listing A.2: Example of Protocol Buffer

```
syntax = "proto3";

package tutorial;

message Person {

  message Address {
     string streetaddress = 1;
     string city = 2;
     string state = 3;
     string postalcode = 4;
  }

  message PhoneNumber {
     string type = 1;
     string number = 2;
  }

  message Gender {
     string type = 1;
  }

   string firstname = 1;
   string lastname = 2;
   double age = 3;
   repeated PhoneNumber phonenumber =  4;
   Address address = 5;
   Gender gender = 6;
}
```

Listing A.3: Example of Simple Binary Encoding

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<sbe:messageSchema xmlns:sbe="http://fixprotocol.io/2016/sbe"
                   package="baseline"
                   id="1"
                   version="0"
                   semanticVersion="5.2"
                   description="Example schema"
                   byteOrder="littleEndian">
    <types>
        <composite name="messageHeader">
            <type name="blockLength" primitiveType="uint16"/>
            <type name="templateId" primitiveType="uint16"/>
            <type name="schemaId" primitiveType="uint16"/>
            <type name="version" primitiveType="uint16"/>
        </composite>
        <composite name="groupSizeEncoding">
            <type name="blockLength" primitiveType="uint16"/>
            <type name="numInGroup" primitiveType="uint16"/>
        </composite>
        <composite name="varStringEncoding">
            <type name="length" primitiveType="uint32" maxValue="1073741824"/>
            <type name="varData" primitiveType="uint8" length="0"
             characterEncoding="UTF-8"/>
        </composite>
        <composite name="varDataEncoding">
            <type name="length" primitiveType="uint32" maxValue="1073741824"/>
            <type name="varData" primitiveType="uint8" length="0"/>
        </composite>
    </types>
    <types>
        <enum name="Gender" encodingType="char">
            <validValue name="male">M</validValue>
            <validValue name="female">F</validValue>
            <validValue name="unknown">U</validValue>
        </enum>
    </types>
    <sbe:message name="Person" id="1" description="Description of a person">
        <field name="Age" id="1" type="uint16"/>
        <field name="gender" id="3" type="Gender"/>

        <group name="phonenumber" id="4" dimensionType="groupSizeEncoding">
            <data name="numberType" id="5" type="varStringEncoding"/>
            <data name="number" id="6" type="varStringEncoding"/>
        </group>

        <data name="firstname" id="7" type="varStringEncoding"/>
        <data name="lastname" id="8" type="varStringEncoding"/>
        <data name="streetaddress" id="9" type="varStringEncoding"/>
        <data name="city" id="10" type="varStringEncoding"/>
        <data name="state" id="11" type="varStringEncoding"/>
        <data name="postalcode" id="12" type="varStringEncoding"/>
    </sbe:message>
</sbe:messageSchema>
```

Listing A.4: Example of FlatBuffers

```
namespace Tutorial;

table Person {
  age:short;
  firstname:string;
  lastname:string;
  address:Address;
  phonenumber:[PhoneNumber];
  gender: Gender;
}

table Address {
  streetaddress:string;
  city:string;
  state:string;
  postalcode:string;
}

table PhoneNumber {
  type:string;
  number:string;
}

table Gender {
  type:string;
}

root_type Person;
```