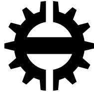

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

**TIMO KALLIOMÄKI**

# DESIGN AND PERFORMANCE EVALUATION OF A SOFTWARE PLATFORM FOR VIDEO ANALYSIS SERVICE

Master's thesis

Examiner: Associate Professor Petri Ihantola

The examiner and topic of the thesis were approved on 31 May 2017

# ABSTRACT

**TIMO KALLIOMÄKI**: Design and Performance Evaluation of a Software Platform for Video Analysis Service
Tampere University of Technology
Master of Science thesis, 50 pages
March 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Associate Professor Petri Ihantola

Keywords: software architecture, video analysis, web service, inter-process communication, virtualization

Video analysis is the programmatic observation of features in a video stream. This thesis designs a software platform which acts as a host for multiple video analyzer applications. The objectives are to allow effortless integration of analyzers such that dependencies between algorithms can be satisfied automatically, provide the analysis functionality over the internet as a service which can act as the engine for client applications, and do this integration in a manner which does not form a bottleneck for the analysis process. The research question is how to build a platform for integrating the analyzers in a way that makes integration easy and achieves good performance.

The thesis consists of gathering requirements for the system, a review of related literature, a description of the design and evaluation and discussion of the designed system from the viewpoints of functionality, performance and architecture. The specification devised for the system defines it at least initially as more of a service to be utilized by the backends of client applications than a scalable content delivery network -like system, and it is emphasized that integration of various heterogeneous analyzers must be easy. Previous literature describes video analysis systems also operating in the cloud, but only ones tailored for a specific purpose and involving only a single analyzer. To make integrating new analyzers easy, the system designed here features the main ideas of allowing analyzers to run in Docker containers and register themselves with the platform at runtime with the platform determining analysis execution order based on information declared at registration-time. For performance, memory is shared between the platform and analyzers to avoid redundant operations.

The platform provides good enough performance, not forming a bottleneck to the operation of the tested analyzer despite a loose approach to coupling, but tests with multiple analyzers operating concurrently would be needed to form a full understanding of the performance. The automatic resolution of dependencies based on requirements declared by analyzers is a novel way of allowing easy integration, and would likely be of use even in versions of the system developed vastly further. The REST API of the produced system is sufficient to facilitate the development of client applications. The stated goals are met, but actual implementation of client applications utilizing the platform would allow better assessment of the fitness of provided functionality. Tests of performance with more analyzers are needed, and if it proves to be lacking, there may be cause for replacing parts of the platform with ones utilizing computing resources more efficiently, or even designing a more tightly coupled analysis architecture operating as a single process.

# TIIVISTELMÄ

**TIMO KALLIOMÄKI**: Videoanalyysipalvelun ohjelmistoalustan suunnittelu ja suorituskyvyn arviointi
Tampereen teknillinen yliopisto
Diplomityö, 50 sivua
Maaliskuu 2018
Tietotekniikan DI-tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: apulaisprofessori Petri Ihantola

Avainsanat: ohjelmistoarkkitehtuuri, videoanalyysi, www-sovelluspalvelu, prosessien välinen kommunikaatio, virtualisointi

Videoanalyysi on kuvavirrassa esiintyvien kohteiden ohjelmallista havainnointia. Tässä diplomityössä suunnitellaan ohjelmisto toimimaan alustana useille videoanalyysisovelluksille. Tavoitteina ovat analyysisovellusten vaivaton integrointi siten, että algoritmien väliset riippuvuudet voidaan tyydyttää automaattisesti, analyysin tarjoaminen verkon yli palveluna joka voi toimia asiakassovellusten kehitystasoisena taustajärjestelmänä, ja integroinnin toteutus tavalla joka ei rajoita analyysiprosessin suoritusnopeutta. Tutkimuskysymys on, kuinka rakentaa alusta analysoijille tavalla, joka tekee integroinnista helppoa ja saavuttaa hyvän suorituskyvyn.

Työ koostuu järjestelmän vaatimusten keräämisestä, katsauksesta aiheeseen liittyvään kirjallisuuteen, suunnitellun järjestelmän kuvauksesta ja sen arvioinnista toiminnallisuuden, suorituskyvyn ja arkkitehtuurin näkökulmista. Järjestelmälle laadittu määrittely asemoi sen ainakin aluksi ennemmin asiakassovellusten taustajärjestelmien hyödynnettäväksi kuin skaalautuvaksi sisällönjakeluverkon kaltaiseksi järjestelmäksi, ja eri heterogeenisten analysoijien integroinnin helppouden vaatimusta korostetaan. Aiempi kirjallisuus kuvaa myös pilvessä toimivia videoanalyysijärjestelmiä, mutta vain tiettyyn tarkoitukseen räätälöityjä yhtä analysoijaa hyödyntäviä. Jotta uusien analysoijien integrointi olisi helppoa, suunniteltava järjestelmä perustuu analysoijien ajoon Docker-konteissa ja itserekisteröitymiseen alustaan ajonaikaisesti alustan määrittäessä analyysisuoritusjärjestyksen rekisteröityessä annettuun tietoon perustuen. Suorituskyvyn vuoksi jaetaan muistia alustan ja analysoijien välillä, jotta päällekkäiset toiminnot vältettäisiin.

Alusta tarjoaa tarpeeksi hyvän suorituskyvyn, eikä löyhästä kytkennästä huolimatta muodosta pullonkaulaa testatun analysoijan toimintaan, mutta täyden suorituskykykäsityksen saavuttamiseksi tarvittaisiin testejä useilla yhtäaikaisesti toimivilla analysoijilla. Algoritmien määrittämiin vaatimuksiin perustuva automaattinen riippuvuuksien selvitys on aiemmin käyttämätön tapa mahdollistaa helppo integrointi, ja olisi todennäköisesti hyödyllinen myös pitemmälle jatkokehitetyissä järjestelmän versioissa. Tuotetun järjestelmän REST-ohjelmointirajapinta on riittävä sallimaan asiakassovellusten kehitys. Määritellyt tavoitteet saavutettiin, mutta todellisten alustaa hyödyntävien asiakassovellusten toteutus sallisi paremman tarjotun toiminnallisuuden soveltuvuuden arvioinnin. Suorituskyvyn testaus useammilla analysoijilla on tarpeen, ja jos se osoittautuu puutteelliseksi, voi olla tarpeellista korvata järjestelmän osia suoritusresursseja tehokkaammin hyödyntävillä, tai jopa suunnitella tiukemmin kytketty yhtenä prosessina toimiva analyysiarkkitehtuuri.

# PREFACE

The work reported in this thesis was made possible by the funding of Tekes, as part of the 360 video intelligence project. The 360VI project is a 2016–2018 project bringing together TUT and various industry actors working on design and development of algorithms for analysis of 360-degree video and applications to utilize the results.

I carried out my work as a research assistant at the Laboratory of Pervasive Computing at TUT between 2017 and 2018. First of all, I would like to thank my supervisor and examiner, Associate Professor Petri Ihantola for both the direction I've received on prioritization of work as well as the invaluable advice on how to write a good thesis. Thanks to you, this work was always exciting. Research Assistant Wenyan Yang at the Laboratory of Signal Processing deserves special thanks for serving as my sounding board regarding the integration process, as well as for bearing with me when I time after time managed to make his code segfault. Shout out to the jolly folk of the F1 corridor and the neighboring areas for being fun colleagues and great lunch company.

This work would have been without purpose if not for all the various parties making algorithms or applications to utilize them. I would especially like to mention the people of Nokia Technologies who coordinated the group effort, and Santtu Pajukanta from Leonidas who educated me on container orchestration.

Whatever undertaking I ever may engage in, I would not be there to do it if it wasn't for the loving support of my parents I have always had the privilege of enjoying. Mom, Dad, thank you.

In Tampere, Finland, on 22 February 2018

Timo Kalliomäki

# CONTENTS

1. INTRODUCTION .......... 1
1.1 Video analysis and 360-degree videos .......... 1
1.2 The objectives of the thesis .......... 3
1.3 Overview of the thesis .......... 3
2. SYSTEM REQUIREMENTS .......... 5
2.1 Elicitation .......... 5
2.2 Use cases .......... 6
2.3 Environmental constraints .......... 7
2.4 Requirements specification .......... 8
2.4.1 External requirements .......... 8
2.4.2 Internal requirements .......... 10
3. BACKGROUND .......... 12
3.1 Video processing .......... 12
3.2 Program execution and memory management .......... 13
3.3 Software architecture .......... 16
3.4 Software systems and the web .......... 17
3.5 Image processing services .......... 18
4. THE DESIGN AND IMPLEMENTATION OF THE 360VI ANALYSIS SERVICE 20
4.1 Web interface and video database .......... 20
4.1.1 API utilization .......... 21
4.1.2 API implementation .......... 25
4.2 Video decoding and flow between platform and analyzers .......... 27
4.2.1 Dependency resolution .......... 27
4.2.2 Decoding and processes .......... 29
4.2.3 Process and data flow .......... 30
4.3 Video analyzers .......... 32
4.4 Software development process and deployment .......... 34
5. SYSTEM EVALUATION AND DISCUSSION .......... 37
5.1 Functionality .......... 37
5.2 Performance .......... 38
5.3 Architecture .......... 41
6. CONCLUSION .......... 44
REFERENCES .......... 46

# LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| API | application programming interface |
| ASIC | application-specific integrated circuit |
| AVC | MPEG-4 Part 10, Advanced Video Coding, a video compression standard |
| BGR | blue, green, red |
| CLI | command line interface |
| CPU | central processing unit |
| CUDA | Compute Unified Device Architecture, a platform for parallel computing by nVidia which allows utilization of GPUs for general-purpose computing |
| DASH | Dynamic Adaptive Streaming over HTTP, a technique for streaming multimedia over the internet |
| DDR | Double Data Rate, a type of memory used on computers |
| FPS | frames per second |
| GDDR | Graphics Double Data Rate, a type of memory used on GPUs |
| GPGPU | General-purpose computing on GPUs |
| GPU | graphics processing unit |
| HEVC | MPEG-H Part 2, High Efficiency Video Coding, a video compression standard |
| HTTP | Hypertext Transfer Protocol, a network communication standard |
| IPC | inter-process communication |
| JSON | JavaScript Object Notation, a structured format for storing and transmitting data |
| MPEG | Moving Picture Experts Group, a multimedia storage and transmission standardization group |
| MP4 | multimedia container defined by the MPEG-4 Part 14 standard |
| PCI-E | Peripheral Component Interconnect Express, a bus used in computers |
| REST | Representational State Transfer, a web service design approach |
| RAM | random access memory |
| SRT | SubRip Text, a format for timed text on multimedia |
| TS | Transport Stream, a media container format designed to be tolerant of transmission errors and allow starting playback even when starting receiving at an arbitrary point |
| URL | Uniform Resource Locator, a way of addressing web resources |
| VRAM | Video RAM, memory used on GPUs |
| YAML | YAML Ain't Markup Language, a structured format for storing and transmitting data |
| YUV | a color encoding system named for its components |
| 360VI | 360 Video Intelligence, a collaboration project involving various academic and private organizations doing R&D related to analysis of spherical video |

# 1. INTRODUCTION

The body of existing images and videos is growing ever larger. While computers have already transformed the way we process text, search for information from it and ask questions based on it, a similar change is ongoing with visual data. *Video analysis* refers to technologies for utilizing videos intelligently by software and machines which enable this. Recent hardware and software developments are making *spherical* or *360-degree* videos more common. This thesis presents a design for the underlying infrastructure for the various software components involved in performing video analysis on 360-degree videos and evaluates its performance.

## 1.1 Video analysis and 360-degree videos

*Computer vision* is a field of artificial intelligence aiming to allow computers to understand the contents of images and videos. Amongst topics of interest are detecting actors, contexts and other features present in visual data. With mathematical methods or more complex, learning models, computer vision methods take in images and output information about the contents of the imagery. For an extensive introduction to the subject, see e.g. Bigun [3]. These techniques enable applications to utilize visual materials in richer ways than simple storage and playback. For instance, we might want to follow the movement of a target through multiple video feeds, a task which is tedious to perform manually for a large amount of material, or present the user of a video player application with interactive options.

Traditionally, the images and videos have been rectangular, portraying one direction from the capture device at a time. Conversely, an observer on the scene may simply turn for another viewpoint. Surveillance systems answer this problem by having multiple cameras in different locations, and presenting users with multiple displays or the possibility to switch between feeds. This is, however, quite different from the way we view our surroundings in nature. 360-degree videos are a more novel solution. They consist of multiple, originally rectangular images recorded from the same viewpoint in different directions, combined to form a picture sphere. There is more information than in a regular video, and the user is free to choose the viewing direction at playback time. As equipment becomes more widespread, 360-degree videos are growing more common.

360-degree videos pose many technical challenges, such as adapting the computer vision methodologies to be compatible with spherical visual representations and developing new applications utilizing the results of analysis performed with the methodologies. Another question is how the users should interact with 360-degree video – instead of traditional

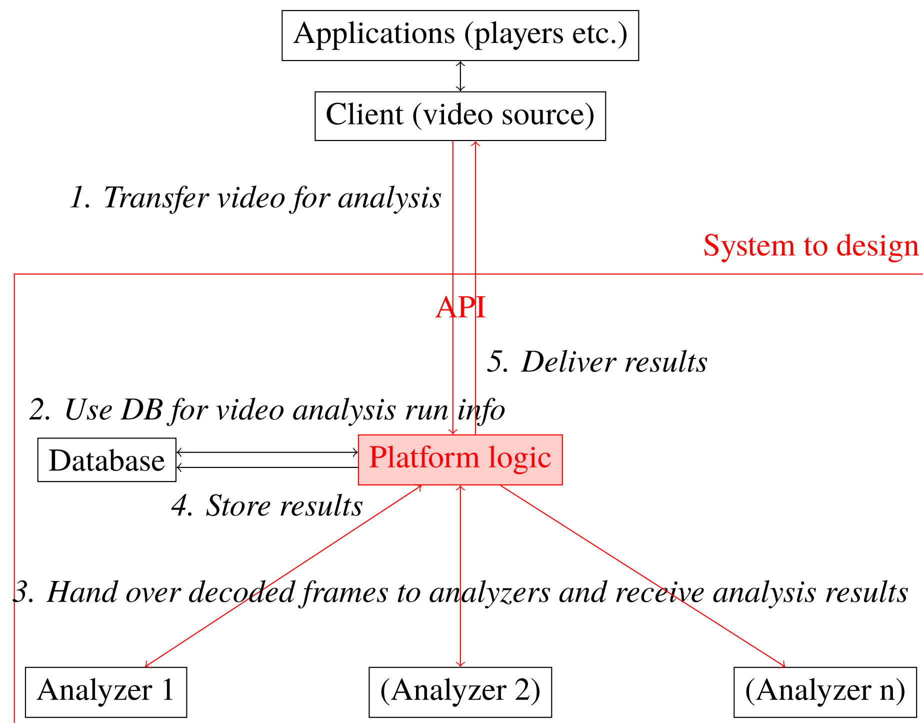


***Figure 1**. The role of the analysis platform in a video analysis workflow*

interfaces like screens and physical input devices, hardware like virtual reality headsets may be used.

A workflow consisting of recording video material, analyzing it with computer vision methodologies and distributing the material and analysis requires infrastructure support to bring the different tools together. For instance, the *algorithms* involved must be orchestrated properly. An algorithm is a formal method of performing some task, and in computer vision usually refers to a way of producing analysis results from video input in computer vision. One algorithm may require results from another algorithm to run, and this kind of execution chains need to be coordinated programmatically to achieve full automation. To make video analysis accessible to a wider audience, an analysis engine might be exposed to be used over the internet.

An *analysis service* is defined here as a software solution which takes in videos and analysis requests providing corresponding results. It integrates several *analyzers*, each being an implementation of a single video analysis algorithm. The artifact providing the service and the analysis infrastructure can also be called the *analysis platform*, as in the algorithm developers' point of view, the analyzers are integrated to the platform. The analyzers and platform form an integrated system with which various *clients* interact to obtain analysis results for some video data they have. Figure 1 depicts the role of the platform in the larger video analysis workflow. For instance, if a user of an application wants to follow a certain person in a video feed, the analysis process starts when the client application, running on e.g. a cell phone, sends a video to the service, requesting the object recognition analysis to be run. The *application programming interface*, API, of the analysis service

dictates the way the request and results are communicated. The platform utilizes a database for information needing to be stored, such as enabling the interaction to be divided to multiple steps, or playing back the same video at a later time. The images composing the videos and any required previous analysis results for each image are distributed by the platform to the analyzers in the correct sequence. In the person-tracking example, the required class-identifying object detection is run for each frame first, and then its results are handed with the frame to the identity-identifying object recognition. Once all analyzers have finished, the platform stores the final results in the database if appropriate, and sends them to the client application.

## 1.2 The objectives of the thesis

The research question of this thesis is **how to build a platform for integrating the analyzers in a way that makes integration easy and achieves good performance**. The reason for the formulation is that the analyzers are heterogeneous software components building upon different software stacks and operating in isolation, which contradicts ease of interoperability and efficient performance.

Apart from the software artifacts and documentation for integration, the output of this work also includes a preliminary analysis interface, as well as integration guidelines for algorithm developers. The video analysis algorithms integrated with the platform are outside the scope of the thesis. The presented platform is a part of the "360 video intelligence project", or 360VI, in which actors from academia and industry are collaborating on the task of 360-degree video analysis, producing algorithms and applications.

## 1.3 Overview of the thesis

The design process starts with understanding what needs to be produced. A video analysis system could be built with all analyzers being a part of a single, uniform software artifact, but the algorithms involved in the 360 video intelligence project are developed independently, and additional work is required for their interoperation. Analysis is complex both in the amount of the data involved and the calculations required [8], so the platform which integrates the algorithms needs to be designed taking performance into account. As there are different parties using analyzed videos, often running on hardware which in itself is not capable of performing analyses, the analysis platform should also be exposed to the outside world so that analysis can be offered as a service – that is, allow clients to remotely request analysis results for video. These needs are expanded upon and developed into specific requirements in Chapter 2. Once the requirements are known, similar systems are reviewed to seek existing solutions and lessons to learn for the design. The evaluation of existing work and the design process require an understanding of the principles and processes involved in video processing and software system construction, for which literature is reviewed. This groundwork for understanding how the requirements can be met in the design is covered in Chapter 3.

The preliminary stages are followed by the design process of the system. The analysis platform to design consists of an interface for analysis, analyzer integration and the result storage required for some use cases. It handles input videos, providing the different algorithms with the data in the correct order and organizes the results. Integrating the analyzers requires defining commonly agreed-upon methods of input and output, as well as handling the input dependencies between algorithms. The dependency resolution is performed without any central precomposed dependency definition. A web interface is exposed for the clients to request analysis execution from the service, requiring documentation as well. The design solutions are detailed in Chapter 4.

After the design phase, the designed system is then evaluated on how well it meets the previously set requirements. The suitability, performance and maintainability of the analysis platform are compared to the specification and discussed in Chapter 5. They are also contrasted with related work. Finally, the contributions and findings are summarized in Chapter 6, which also presents suggestions for future work.

# 2. SYSTEM REQUIREMENTS

As there was no detailed specification to start with, the first step in devising the system is to collect requirements. The requirements are a list of statements concerning the system which it must fulfill to meet the needs of the stakeholders. The stakeholder groups with the most direct involvement with the video analysis service system are *application developers* who build software for end users utilizing video analysis and *algorithm developers* who implement the individual analyses (cf. Alexander [1]). This chapter begins with covering what these groups need. The prevailing assumptions about the environment the system will be run in are also explicated. Based on these premises, requirements for the system are laid out.

## 2.1 Elicitation

The first step in system design is requirement elicitation: what must the system to design provide to its users and what are the technical needs constraining the solution. The analysis platform must act as an intermediary between applications utilizing video analysis and analyzers providing it. In the end, the added value of a video analysis system lies in utilization of analysis results by the end users. They interact with various applications, such as video players, which in turn request analysis from other software. The system being designed fulfills the role of providing analysis, even to applications running on less performant hardware.

This work started with technical design meetings with application developers involved in the 360VI project. The agenda was to discuss the needs for the system and identify the core usage cases to guide the design of the platform. Due to the fact that the purpose of the system is to provide a service to other software rather than being a product with intrinsic value, requirements were elicited from other developers rather than end users. Methodology such as interviews and observation in context was not considered necessary in this situation.

The premise for the design process was to develop a single system providing multiple types of video analysis. Several algorithms along with their characteristics and requirements regarding input and output were laid out in informal discussions: many of the algorithms are implemented on *graphics processing units* (GPUs), some on *central processing units* (CPUs), and existing implementations had various input requirements. The application developers were involved as well, giving input on what use cases the system might satisfy and what kind of an analysis result format would be the most suitable for utilization by client applications.

As the system is of experimental nature, the requirements were allowed to evolve long into the project. This was achieved via iterating from the abstract to the concrete, gathering feedback along the way. After the initial meetings, a high-scale draft of the system design was made and presented to the algorithm and application developers. This yielded mainly comments on prioritization: what features would be most useful in the short term, and what could be left for future implementation. The next stage were a draft API documentation for application developers and integration instructions for algorithm developers to learn which parts were considered suitable and which, if any, could be problematic. No feedback was received, so it was assumed the stakeholders considered the designs adequate for the needs known of at this point in time.

## 2.2 Use cases

The main kinds of analysis usage patterns that were identified to serve as the base for a requirement specification were

- “batch”, in which an existing large pool of video data is to be analyzed, with a one-time upload to the service and download of the results
- “stream”, in which an existing video stream (a third-party service acting as a video source) using the MPEG DASH protocol [13] is to be augmented with analysis results, and
- “live” (or near-live), in which video is to streamed to the analysis platform and the results received in full-duplex.

In addition, a query interface for existing results was discussed as potentially being of use. Queries to a database of video analysis results would enable use cases such as “find all videos which have cars.”

Common to all the identified cases is receiving an analysis output corresponding to the input video and utilizing the results in some way. The observations made about the video may be utilized in tandem with the original video itself, or perhaps used for simultaneous processing of larger amounts of videos. While humans could understand natural-language labels such as “car” or “face” in the results, intelligent applications require *ontologies*, controlled vocabularies which enable automatic reasoning [18]. When an analyzer output conforms to a known ontology, it becomes easier to utilize the results in combination with existing applications due to having a “common language.” A very simple example of the advantages of using an ontology could be to represent the classifications using translations, icons or some other indication useful to the users in the displayed view. A more advanced application might be a self-driving car making decisions based on its surroundings (a case which would likely warrant dedicated local hardware to ensure availability).

On the other side of the platform are the algorithm developers, who produce the analyzer software artifacts. An analyzer takes as its input *frames* – the still images comprising a video

– and possibly analysis results derived with other algorithms, and outputs new analysis results. The analyzer needs to be provided with a way to receive input and report output. No specification for an interface like previously exists, with each analyzer implementation having its own way of operating. A single interface applicable to all the different analyzers is needed to make integration possible.

Sometimes nothing else than an image is required by the analyzer, sometimes the results of a certain algorithm for that image are needed, and sometimes the results of a previous algorithm for multiple frames are needed. This makes the analyzer dependencies complex. Another complication is that while an algorithm can be stateless, always simply providing the same output for the same single input, this is not always the case. Algorithms may also analyze videos over a longer timespan with a certain frame affecting also the analysis of *previous* ones, which in practice could mean e.g. providing results only after every 32nd frame.

The particular algorithms considered for initial integration into the platform were *object detection*, which detects objects and infers their general classifications operating statelessly on a per-frame basis, and *context recognition*, which provides situational awareness regarding the video and requires the object recognition results of frames after the one to analyze. Other, less advanced algorithms to possibly be integrated in the future are *tracking* (linking together observations in discrete-time results), which needs the object recognition results up to the frame to analyze, and *activity recognition*, which requires context recognition results of frames after the one to analyze.

## 2.3 Environmental constraints

In any project, there are technical and organizational realities which rule out some solutions which could theoretically answer the needs. These limits must be taken into account in addition to the desired added value before laying down the detailed specifications for implementation.

To increase the chances of the system being utilized, it should be easily approachable. An interface must be specified for the application developers to communicate with the service, and this interface should allow integration with as wide a spectrum of applications as possible. The current norm in communication over the internet between applications by different developers are REST APIs (Representational State Transfer) [14] which are most often implemented using JSON (JavaScript Object Notation). In interoperability, popularity of a technology must be taken into account as one choice criterion. The large majority of APIs introduced recently are in JSON format, which practically makes the format an assumption for new ones [12]. A JSON API was chosen for communication between the video analysis platform and clients for this reason.

The various algorithms are developed independently by developers in various organizations. This leads to there being no shared codebase, and even differing assumptions about the

Linux distribution the analyzers are executed on. Therefore, a tightly integrated design for the platform, with various analyses taking place within a single process, was not considered. Another consequence is that the interface between algorithms and the platform should be as simple as possible, since constantly changing implementation details of different algorithms in tandem is not feasible.

The algorithms may also depend on each other in ways described in the previous section. This forms implicit dependency graphs, complicated by the fact that there are different modes of dependencies. Since the development approach of “make the platform first, integrate algorithms afterwards” means the platform cannot take the role of a central repository of information regarding the algorithms, there is no prior knowledge of the dependencies when implementing the platform. Consequently, the platform must be pliable to introduction of new analyzers with algorithm dependencies and cannot have a hardwired definition of the execution sequences.

The computer vision operations performed by the algorithms are very resource intensive [8]. The internal parallelism and highly specific computing units of GPUs (cf. [36]) make them suitable for certain computer vision operations. Thus, many analyzers running these tasks are developed to run on a GPU, or at least execute some operations on one, and GPU access must be provided to them.

Some online services have millions of users, and scaling to this kind of scenarios is a nontrivial task involving e.g. coordination of duplicated resources. However, the scope of this thesis is limited to a research-and-development-tier system, so scaling to a large number of requests is not included in the goals to meet. Only the speed at which one or a few can be served is a priority. This means that the internal workings of the platform must be efficient enough for the pass-through time of a single video to be low, but performance when several multiple analyses are requested at the same time will not be considered in depth.

## 2.4 Requirements specification

Based on the desires and constraints above, the specifications which the system must fulfill were formulated and listed. They are grouped into two groups corresponding to the main two involved shareholder groups of application and algorithm developers, respectively. The “external” requirements prescribe how analyses are requested and presented, while the “internal” ones relate to how data and control flow inside the platform, between the analyzers (cf. [2]).

### 2.4.1 External requirements

The external requirements concern the functionality provided to application developers. They define how a client, for example a video player on a cell phone, interacts with the

analysis service: operation sequence, division of responsibilities, and input and output formats.

**Support videos in MP4 [9] container format** The system takes in video files or streams. A video is digitally represented in the format of a *container*, which is a multimedia file with one or more interleaved *tracks*, which may be video, audio or subtitles. MP4 was found to be the most commonly utilized container format on the ecosystems utilized by the application developers.

**An API suitable for offline cases** The interfacing entity posts a whole video file and receives the video augmented with the analysis results. Additionally, the API allows the interfacer to explicitly specify whether the result should instead be delivered as a stand-alone JSON file without the media originating from the client rather than as a track in a re-built MP4 container.

**An API suitable for MPEG-DASH streams** The interfacing entity posts a link to a DASH stream. The analysis service downloads and analyzes the segments of the stream and provides a new DASH manifest with both the original media segments and the analysis results. Since the segments form one "logical video", an analysis result may be affected by multiple segments.

**A flexible data format** Different algorithms provide different results. There are some information fields which are applicable to results from more than one algorithm, and the data format must allow representing information with the same semantics similarly.

- A result usually has an associated timestamp, but it may concern the whole video. If applicable, the result must have a timestamp which unambiguously identifies the relevant position in the video.
- A result may be localized to a certain rectangular area on the video, or it may concern the whole frame. If applicable, the result must have a location specification consisting of a coordinate pair, width and height of the bounding box.
- A result may have a score, indicating how confident the algorithm is about the result. This score is on a scale from zero to one.
- The result may be a class from a controlled vocabulary. The result must indicate the vocabulary in addition to the class.

Additionally, the algorithms may be updated. A result set must identify both the algorithm which produced it and its version.

**A URL for analyzed videos** After the platform has accepted a video for input, it must provide a *uniform resource locator*, URL, for that video. This URL can be used to get retrieve results at a later time.

**Single-algorithm specification** The interfacing entity must be able to specify a desired algorithm without knowing the dependencies of that algorithm. The algorithm to run is given in the form of a simple string containing the algorithm name, and it is the

responsibility of the platform to also execute the required dependencies in the correct order.

Another way to see these external requirements is as a promise of functionality provided. The specification can therefore be compared by application developers against their needs to evaluate the usefulness of the analysis service to them. Conversely, newly arisen needs should be considered when updating this specification.

### 2.4.2 Internal requirements

The internal requirements for the platform are related to the analyzers. They specify the interaction of the software components composing the video analysis service and establish some parameters regarding the quality of the implementation.

**Integration** The platform must support integration of analyzers running in any modern Linux environment. Each algorithm developer may choose a distribution and libraries to utilize. Adding an analyzer must be possible without development efforts on the platform.

**Communication** The passing of data between the platform and analyzers must happen at a high throughput. Disk writes are too expensive: processing one single 8-bit color 4K video at 30 frames per second needs a throughput of 712 MB/s, when most SSDs can provide around 500 MB/s.

**Dependency resolution** The platform must resolve the algorithm dependencies needed to determine the correct order to run all required analyses. It must support the following modes of interdependencies

- single-frame: algorithm needs the output of another algorithm for the currently processed frames
- whole-video: algorithm needs the output of another algorithm for all frames of the video

The tracking algorithm is already implemented with an internal result buffer, otherwise a “cumulative” dependency mode might make sense to support processing time series. Context recognition is for now supported with the whole-video dependency, but it might be more efficient to define a dependency mode with requirement “windows”, e.g. “please provide the object recognition results 16 frames prior and 16 frames after the current frame to analyze”.

**Frame providing** The platform must hand over frames of the videos in the correct order. When an analyzer is given the task of analyzing a frame, it must be able to assume availability of all information which has been listed as necessary for the operation of the algorithm.

**Easy deployment** Both the platforms and the analyzers integrated into it must be effortlessly installable in a new environment. This should be possible without involved

configuration or knowledge of the software requirements of each algorithm on the deployer's part.

The external requirements were subject to changes in the wider ecosystem. These internal requirements, on the other hand, are likely to evolve only based on the needs of the parties directly involved in the development of the system.

# 3. BACKGROUND

To understand how to fulfill the specification devised from the requirements gathered in Chapter 2, this chapter reviews earlier literature. The main material handled is video, so the basics of digital imagery and video are visited, and since video data is large, the data handling capacity and techniques of computers are explored. Existing theory of software architectures and literature on previous cloud systems are reviewed to base the architecture of the system under design upon. Finally, existing cloud vision systems are reviewed in particular depth, since they are the ones with the largest potential to learn from when designing a new one.

## 3.1 Video processing

The most typical representation of video in digital processing is as a series of still image frames. This *raw* representation, while simple, makes the data requirements for any non-trivial length of video far too large even for modern computers. Therefore, digital video is nearly always stored and transferred in an *encoded* form. The encoding process consists of *spatial compression*, which treats individual frames applying still image compression methods such as color space quantization and redundancy removal, and *temporal compression*, which expresses some frames as a reference to another frame plus the difference between the two. This is a lossy process; some visual information present in the original images is lost. The videos are only *decoded* into raw visual data when they are to be played back or otherwise used. This reduces the storage and bandwidth requirements, but increases the need for processing power, as both encoding and decoding are expensive operations. Dedicated *application-specific integrated circuits* (ASICs) for video decoding and encoding exist, and the hardware designed for executing specific operations provides more efficient operation than general-purpose computing units. [15, p. 111–146]

Videos, like still images, can be expressed in several color formats. Monitors and other display devices typically receive their input as a combination of numeric values for the red, green and blue color components. However, since the human eye is more sensitive to differences in lightness than in hue, it is more efficient to allocate more bandwidth to the former than the latter. This technique is called *chroma subsampling*, and it introduces an additional layer of complexity into video processing: conversions between transfer and display color space. The YUV system is the most common encoding. Similarly to playback, computer vision algorithms also usually require either grayscale or even-components representations, so a color representation conversion is necessary before analysis. This conversion, although much less complex than image encoding and decoding, is also often done on hardware specifically built for the purpose. [7, p. 212]

Even when using an even-components representation, there are variants differing in the order in which color components are laid out: for each pixel, there are separate red, green and blue values, which may be ordered differently. RGB is the most common ordering, and BGR the second. Regardless of how the color components are defined, there may be different *color depths*, which signify the number of bytes used for each channel in a pixel. The most typical color depth is 8 bits per color, or 24 bits per pixel, while 10bpc and 12bpc are emerging as solutions for systems where greater color precision is required. [7, p. 161–164]

After compression, videos are typically stored in a *container*. A container file format defines how to divide several concurrent elementary streams of audio or video into small *packets* and interleave, or *multiplex* the packets such that bits of information presented at the same time are located close to each other in the resulting stream. This increases complexity of producing media files and playing them, but is needed to achieve synchronized transmission. The container also provides a way to transmit metadata associated with each track or the whole file. [15, p. 31–38]

A computer vision algorithm will need only the demultiplexed images from the packets of a single video stream. One container-related complexity to consider when designing systems which transfer videos is that in MP4 containers [9] the so-called `moov` atom, video metadata required for decoding, is often placed at the *end* of the file, meaning decoding cannot start until the `moov` at end of the file becomes available. The `moov` can be relocated, an operation sometimes referred to as fast start preparation, but it is far more commonly placed at the end because that simplifies the video authoring workflow.

The decoding of video is a nontrivial process to orchestrate. One naïve approach might be to first decode the whole video and then start displaying or processing it. There are two problems with this approach: because video data is large, it may not fit into the system RAM (random access memory) at once, and it may be desired to start working on a video input before all data has been received – or indeed, even recorded. Since decoding and processing/display are a producer and consumer which may proceed at different speeds, one may need to pause decoding to wait until further encoded input is available or to prevent filling up buffers on the playback side. [38, 30]

## 3.2 Program execution and memory management

When particularly low latencies or large throughput are required from a data processing system, it is useful to remember the physical qualities of computers. The CPU of a computer has an extremely fast on-die cache, but the caches are very small in capacity due to cost and physical limitations. The CPU receives the instructions and data to process from other components via a fast interconnect, such as the Intel QuickPath, connected to the motherboard. Among possible fast sources and targets of data are the system RAM, typically attached to a DDR (Double Data Rate) bus with a capacity of 17 GB/s (version 4), and the

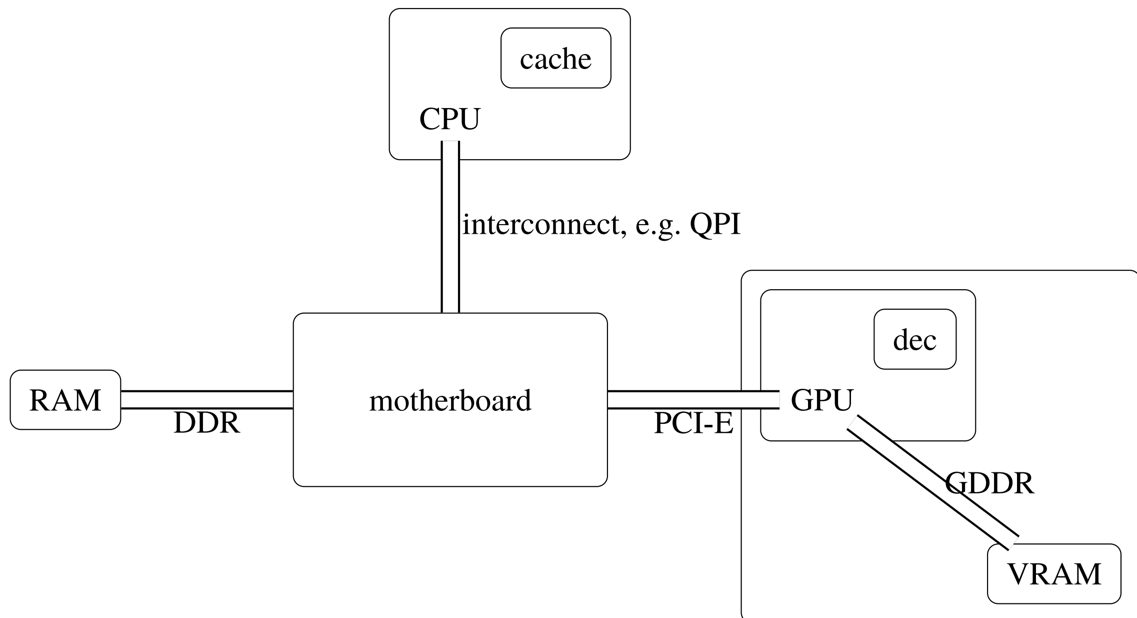


***Figure 2**. Processing architecture of modern computers*

GPU, typically attached to a PCI-E bus (Peripheral Component Interconnect Express) with a capacity of 16 GB/s (version 3 x16). The graphics processing module consists of the GPU itself and Video RAM (VRAM) memory attached to it using the GDDR (Graphics Double Data Rate) bus with a capacity of 56 GB/s (version 5X). Figure 2 illustrates the components and connections mentioned. Also relevant to note is the dedicated chip for video encoding and decoding present on most modern GPUs. There are also CPUs with integrated graphics processors, but the performance of IGPs is far lower than that of most powerful dedicated chips. (cf. [5, Chap. 4, 6–7])

The throughput capacities of the various buses in a computer can be contrasted with the bandwidth requirements for video. 0.15 GB/s for very commonplace 24 frames per second 8-bit 1080p is very much smaller than any of those capacities, with multiple parallel live-speed operations being possible. On the other hand, the 2.8 GB/s for 30FPS 10-bit 8K, which could become commonplace in the not-too distant future, is already a considerable chunk of the PCI-E and DDR capacity for just one video stream. The throughput of persistent storage is in the hundreds of megabytes per second, so it is not feasible to use it for raw video data.

As long as the data to be processed by a program remains in the random access memory address space of a single process, reading it can be considered very fast. While complicated processor-level caching is necessary to achieve this fastness and software may be designed for optimal cache behavior, the effects of cache hit optimization can be considered negligible in comparison to the effects of sharing data between multiple processes. In the latter case, either data must be copied from one address space to another, or the memory needs to be shared. While typically IPC (*inter-process communication*) is done by the former approach, this imposes a performance penalty for each copy. In particular, in a multiprocessor system,

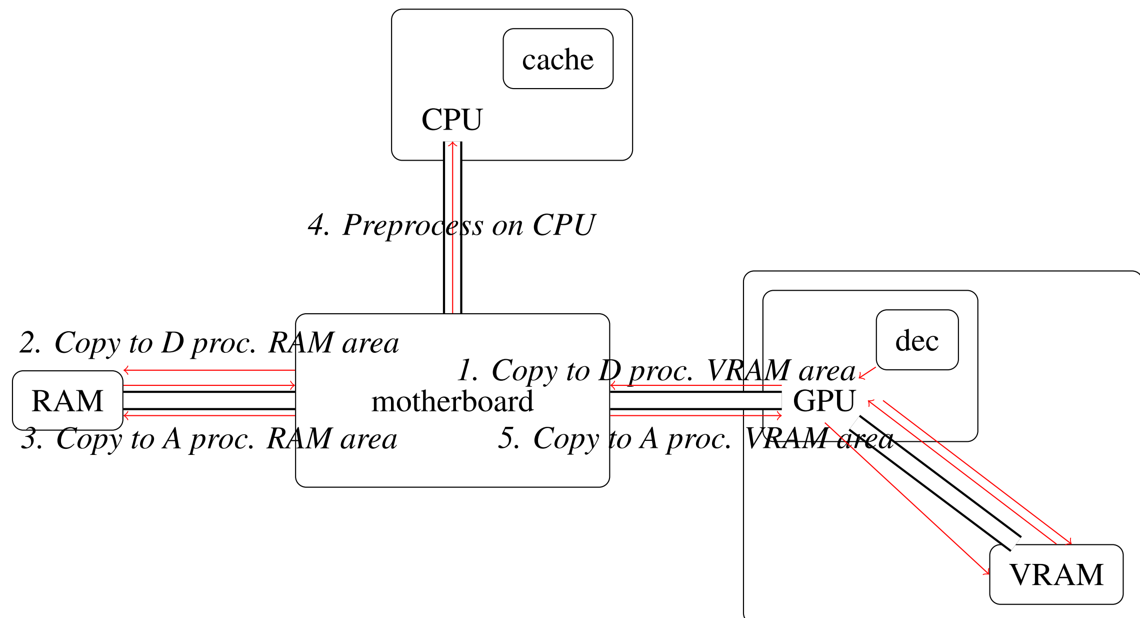


***Figure 3**. Memory copies of raw video data when decoding and analysis are in separate processes*

each processor may have its own physically distinct memory, with accessing other areas of memory being slower. Shared-memory implementations for IPC exist, sometimes even providing a message-passing abstraction, but they complicate the software architecture and require more work to implement. [4, 19]

The hardware components involved in the memory utilization design are not limited to the CPU and RAM. As many computer vision tasks are performed on GPUs, the cost of transferring data through the PCI-Express bus and back must be considered. Since the bus is slower than RAM, with the bandwidth of the whole PCI-E 3.0 `x16` bus (15.8 GBps) roughly equal to the bandwidth of a single DDR4 memory module (17 GBps), memory copies from the CPU to the GPU address space are even more expensive than inside or between RAM modules. The performance impact of time spent doing copies between the central and graphics processing subsystems varies by application and its data access patterns, but can often be considerable [17].

The performance of heterogeneous CPU-GPU computing has been widely studied from a low-level perspective, usually within a single process and often with only one algorithm at a time, and it has been observed that some applications benefit greatly from simultaneous usage of CPU and GPU, but data movements need to be carefully planned [39, 44, 27, 34]. Figure 3 illustrates how raw video data may be copied multiple times on computer hardware if decoding and image processing occur in different processes. Before video is decoded on the GPU, it is in a compact, encoded form. A specially developed application may perform operations on the video data on the GPU, but most typically the frames are copied to decoder process RAM if any processing is to be done. Unless shared-memory IPC is used, another copy of the video data is made inside RAM to provide the analyzer process

with the data. The analyzer process may perform some preliminary operations for which the data must be read by the CPU; for the main analysis tasks the GPU is used and thus a copy to VRAM is needed. This kind of round trips from and to the GPU, and possibly also on the RAM side, are obviously inefficient. They, however, may be tolerable due to the large bandwidth of the fast buses: the time taken for even a couple copies of one frame is still quite small, possibly a negligible percentage of the time required for analysis.

If there are multiple analysis tasks running on a GPU in different processes, the number of copies in naïve implementations raises even further. GPUs have their own VRAM, the management of which is not entirely the same as main system memory. While there exist tested solutions for inter-process communication like on the general processing side, off-the shelf solutions for GPU IPC are far less mature than corresponding CPU ones making implementing software utilizing GPU IPC tedious [41]. Elimination of the RAM-side copies is easier to implement than GPU IPC. The recently introduced heterogeneous processors fulfilling both the CPU and GPU roles eliminate over-the-bus memory copy overheads by using unified memory spaces [21]. This may simplify software designs in the future, but the heterogeneous processors available today are mostly low-power solutions rather than high-capacity ones. This means that most practical systems heavily utilizing graphics processing are still built with dedicated GPUs with their own memory.

## 3.3 Software architecture

The simplest computer program is developed at once, the only unknown before running it being what input it will receive. In more complex systems, the possibility of easy expansion might be desired. How to extend software systems with components which are not known about in advance is not a new question: *run-time registration* is a classic pattern in software architectures, allowing a framework to be defined without prior knowledge of component implementations which will be available. Once a compatible component has been produced, it can announce its presence to a register in the framework, from which consumers of the framework can query the components available for utilization. The pattern is often demonstrated with classes, but can also be used on higher-level structures such as application plugins or even independent systems. In the latter case there may be the added complexity of knowing when a registered system becomes unavailable, especially if this can happen unpredictably. [16, p. 120–121]

There are many ways to organize the interaction of software components, the term itself having different meanings in different contexts. The particular topic of how to organize processing of data without a great degree of interactivity or high-level logic has been studied widely, with proposed solutions ranging from low-level ones like compile-time schedulers [47] to high-level ones like implicitly-declared flows [43]. One approach suitable for multi-step data processing flows is the *pipeline* pattern, described e.g. by Mattson [33]. A software pipeline is analogous to an assembly line and is useful for both conceptualization and performance when there is a sequence of information on which multiple operations need

to be performed. Even if a single operation is not parallelizable, parallelism of processing units may be taken advantage of on large inputs by having different units work on different parts of the sequence. Pipelines lend themselves well to situations where different units are best suited for different tasks, allowing all processing units to be in operation most of the time. A pipeline is a high-level view to parallelism, and an execution stage may be internally parallel as well if the stage can be parallelized between subunits.

## 3.4 Software systems and the web

A traditional way of allowing different software platforms to run on the same physical machine, and to provide isolation for security reasons, is *virtualization*. In virtualization, a software running on the operating system of physical *host* computer provides an abstraction layer emulating hardware, allowing multiple *guest* operating systems to act as if they were running on their own hosts. This allows more flexible and efficient utilization of a single machine. Drawbacks of virtualization include the abstracted hardware reducing performance and each guest operating system requiring its own, large software image. A more recent development are software *containers*, which instead of a full virtual computer only define sandboxes inside which different operating systems can run on the same computer, all interfacing almost regularly with hardware. Performance impact is negligible and container images can be produced in stacks, allowing e.g. the same operating system image to be utilized in multiple application images. [37]

While GPUs may be virtualized when used in the cloud [22], application container systems treat the GPU like the CPU, exposing it as-is to individual containers. This means that the GPU resource is shared the same way as if between different processes on the same operating system, so like with CPUs, the addition of the operating-system-level virtualization does not cause a notable performance penalty. On the other hand, while CPU instruction sets such as x86 are highly standard and ubiquitous; there are competing general-purpose GPU programming languages. Another practical complication is specific GPU drivers being required, making even application-level virtualization of *general-purpose computing on graphics processing units*, GPGPU, require more involved setup than that of software running on CPUs [35].

A crucial part of building a system to be interfaced with by other systems is defining an appropriate interface. On the web, a popular approach is making "RESTful" (Representational State Transfer) APIs, as described by Fielding [14]. The approach prescribes using standard HTTP (Hypertext Transfer Protocol) methods on *resources*. A resource is a data entity which can be listed, added, modified or deleted. For instance, instead of defining an operation called `incrementField`, the client submits a new representation of a resource with an incremented value in one field. There is no notion of sessions: all state is contained in representations of the resource, making individual interactions stateless. Pautasso *et al.* consider RESTful APIs easily approachable due to their uniform interface, which is both easy to understand and usable with very simple tooling. Identified disadvantages include

the difficulty of how to represent specific operations as basic interactions on resources, as well as guaranteeing the quality-of-service being challenging [40].

An emerging trend in software organization are *microservice architectures*. Villamizar *et al.* characterize microservices as more of a philosophy than a strict pattern to follow: having systems composed of distinct parts developed and deployed separately, and using separate persistence, can make complex architectures more easily understandable and manageable [45]. Depending on the implementation, microservices can also make applications scale better, allowing hardware upgrades to target the parts of systems needing the most extra capacity instead of duplicating all components, leading to redundancy. The core idea is that each component service has its own responsibility, and apart from predefined simple interfaces (which should be flexible enough that changes to each need not be made in tandem), changes can be made to one service independent of the others. A typical example of a microservice architecture, according to Villamizar *et al.*, might be a web service built with multiple smaller services, each of which provides some closely related group of functionalities through a REST API and has its dedicated database or other method for the persistence of data. Microservice architectures can be compared to the *interface segregation* principle often discussed in the field of object-oriented programming, as both call for narrow interfaces dedicated to certain activities with as little reason to change as possible.

## 3.5 Image processing services

Virtualization, RESTful APIs and services are approaches typically used in building cloud systems. The particular task of video analysis may not be the most typical example, but usage of cloud infrastructure has been found to have the potential to increase performance and robustness of analysis workflows [31]. Multiple studies like Wu *et al.* [46] and Zhang *et al.* [49] have proposed container-based virtualization for usage in computer vision systems with image analysis algorithms. Identified benefits over hypervisor-based virtualization include smaller performance overhead, run-time resource reallocation and live system updates. The most common system used in previous literature is Docker, which Linux containers on Linux and more recently Windows hosts. These studies discuss resource allocation algorithms but not the methods and performance effects of sharing input data between computing resources. Furthermore, they only cover the case of a single computer vision algorithm, or a single algorithm and tracking, so the issues of data sharing performance and interfaces are not discussed.

Other approaches for computer vision system architecture include MapReduce [32], Apache Spark [28] and Apache Kafka [25]. These software architectures are more monolithic than container-based ones, requiring processing software to be specifically developed to integrate to the systems. The studies also omit the details of placing video decoding and algorithm interoperation in the workflow, emphasizing efficient implementation of a single algorithm or the scalability of the system for a large number of inputs by generous allocation

of processing resources. No study was found that treated analyzers as distinct components with interdependencies.

Traditionally, service-oriented architectures have been largely based on the usage of CPUs while computer vision typically requires GPU resources. More recently, cloud services providing also powerful graphics processing capabilities are emerging. They can be used in multiple ways, the most relevant of which here is is the operational systems layer. This means lower-level tasks for which GPUs lend themselves to particularly well. The hardware used is most typically either traditional, dedicated GPUs or more modern GPGPU units which are largely the same as GPUs with regards to architecture, but do not feature display functionality. Hybrid processors are not used often in the cloud. A challenge that remains is that GPUs are less standardized than CPUs: while the same software can run on Intel and AMD processors due to the common x86 instruction set, the same is not true for nVidia and AMD GPUs. [6]

# 4. THE DESIGN AND IMPLEMENTATION OF THE 360VI ANALYSIS SERVICE

To integrate the various analyzers into a single system accessible over the internet as described in Chapter 2, an integration system with a REST API, input handling, frame extraction, algorithm dependency resolution, and resource management was built. This chapter describes and explains the design choices from the algorithm integration, service usage and internal points of view. The ordering and visualizations of the chapter are adapted from Kruchten [26], who describes a model for software architecture descriptions consisting of

- a logical view, which outlines the main concepts involved in the system and their associations,
- a process view, showing the flow of execution and process lifecycles,
- a development view, depicting how the software is organized into various separately-modifiable sections,
- a physical view, displaying the placement of processes on physical units and the communications between the units,
- and scenarios, archetypal use cases which serve as a starting point and validation for designs.

Presented first is the "outsider's" logical view, in conjunction with the API exposed to clients. This is followed by the platform-internal process and development views, which describe the platform-orchestrated process flow and platform/analyzer component distribution, respectively. Finally, the placement of the designed and utilized software artifacts is shown in the physical view alongside with some discussion of deployment of the whole system. The "plus one" of scenarios has largely been covered in Section 2.2.

The produced artifacts – source code and documentation – are open source. They will be available at https://bitbucket.org/tkalliom/360vi-platform in August 2018, when the funder-mandated embargo is over.

## 4.1 Web interface and video database

This section covers the design of the API, placing emphasis on the application developer's point of view. The design for the video analysis API is based on the requirements in Section 2.4.1 and aims to be a typical, idiomatic REST API. To further make the API approachable for application developers (see Section 2.3), some inspiration was drawn from the YouTube

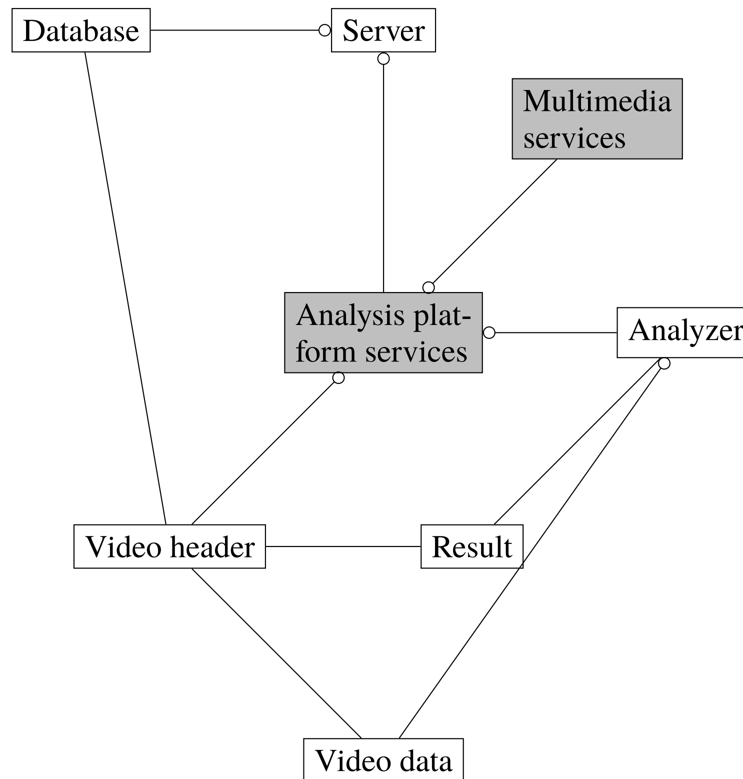


***Figure 4**. Logical view of the analysis service. Simple lines indicate association, circled lines indicate usage. Shaded elements are libraries without a lifecycle of their own or saved state.*

video API [48], as it is fairly well known. The main incorporated idea is the header/data division in video upload.

The client first connects to the analysis service server to create a *video header* containing metadata about a video, most notably the desired algorithms to run. The service stores the header in a database. Data is then uploaded to correspond to the header, and the stored header determines which analyses to run. The server lets platform logic decode the data using stock multimedia libraries and call analyzers to augment the video with results, also stored in a database. The results are then sent to the client. Figure 4, the logical view of the architecture, provides a summary of these higher-level concepts involved in the system and their association and usage relationships. Application developers interfacing with the analysis service need to understand this model, while objects on a lower level than the ones portrayed should not be necessary to learn to utilize the API successfully. An API reference more detailed than the one given in this section is included in the implemented application in a standard format.

### 4.1.1 API utilization

A good documentation is crucial to make the entry threshold for the usage of a software component or service low. In order to make the documentation familiar and thus accessible to as many developers as possible, the widely used Swagger API specification (see [29])

was chosen. With Swagger, the API developer writes a formal API description in the YAML markup language using Swagger syntax, and then a JSON API documentation is exposed to potential users of the API. The documentation can include human-readable notes and is typically rendered into a human-readable layout using tooling, but the definition being formal brings advantages such as being able to run test API calls from the documentation webpage. Version 3 of the specification, now titled OpenAPI, was released after the documentation in this project was made.

The usage of the video analysis service begins by choosing which analyses to run on a video. Typically, an application would know the names of certain algorithms it can utilize, but the current availability of various algorithms can be retrieved from the `algorithms` endpoint, which reports the name, version and dependencies of each algorithm. Most of the time the application developers will not need to know the dependencies, but the listing is a quick and convenient way of confirming the names to use. While the algorithms do have platform-assigned IDs, name-based usage means the request does not need to be different when sent to different instances of the analysis service. Furthermore, the ID can change with minor updates to the algorithm. This way analysis results can have links which allow finding out exactly which version of an algorithm version produced the results, but applications do not have to keep track of the ID changes. The information could be used e.g. to know when to re-run the same analysis for improved results.

Analyzing a video starts by uploading a video *header*. The main function of the header is to specify which algorithm runs are desired; this is done by giving a simple list of algorithm names. An alternative might be to specify the algorithms using URLs, but this would require defining a prefix namespace and cause more work to application developers, and no practical benefits were identified. The video header is only metadata; for instance, the file format and codecs of the video to be uploaded remain unknown to the service at this phase. The motivation for the separate header stage is to support chunked uploads, where the video data is uploaded in multiple stages. This provides more flexibility, as e.g. a network failure will only affect one part of the video, and the client is also allowed to pause the upload without causing request timeouts. The two-phase upload API design solution was inspired by the YouTube API. Other approaches might be to send the video as encoded form data or in a multi-part request, but these methods are not suited for chunked uploads, and the former is also highly inefficient for large binary blobs such as videos. An example of a request with a video header is given as Listing 1a.

The service responds to a video header upload with a URL it assigns to the video to be uploaded. The assigned identifier will also enable retrieving the analysis results later from the `analyses` endpoint. The header upload response payload will also include the complete resolved algorithm dependency tree, with URLs specifying also the current algorithm version. This information will likely not be necessary for everyday use cases, but was chosen to be returned for informational and debugging purposes. It could hypothetically be used for visualization, e.g. generating a graph of the constituent algorithms for an application.

```
1  POST /videos HTTP/1.1
2  Content-Type: application/x-www-form-urlencoded
3  Content-Length: 57
4
5  analysis_algorithms=context_recog&analysis_algorithms=object_recog
6
7  POST </videos URL indicated by previous response>
8  Content-Type: video/mp4
9  Content-Length: 5120
10 Accept: application/json
11
12 <binary data>
```

***Listing 1***. *Examples of requests to a) upload a video header (lines 1–5) and b) upload video data (lines 7–12)*

Once a header exists in the analysis service, the client can start uploading the actual video data. An example of a video data upload request is given as Listing 1b. The current input formats supported are either a single MP4 or TS (Transport Stream) file, or multiple ones forming one “logical” video. For TS files, the analysis service is able to start processing the file before the upload is finished; the same is not true for MP4 files due to the possibility of the `moov` atom being at the end of the file. For performance reasons, it could make sense to mandate fast start optimization for MP4 inputs, so processing could always start while uploading is in progress. To indicate a chunked upload, the client can set the `Chunk-Number` HTTP header in the request. In this case, an empty request body will serve as the end-of-file marker.

The upload functionality supports standard HTTP content negotiation [23]. The analysis client can place an HTTP `Accept` header in the upload request to indicate whether it analysis results in-band with the video (MP4) or out-of-band (JSON) are desired. This header is given in the data upload stage, as an `Accept: video/mp4` header does not make sense before the server has any multimedia data for the video to include in its responses.

The response to a file upload depends on the uploaded file. If the analysis server is already able to run all analyses for the file, the client will immediately receive the analysis results in the response. If further client action is required – for instance, after uploading a single chunk of a video when analyzers requiring the whole video are run – the response will be empty. Section 4.2.3 covers these possible process flows in greater detail. The analysis results can be retrieved from the service once analysis is complete using the URL assigned to the video.

A sample response for retrieving a video header is given in Listing 2. While far smaller than the videos, analysis results can still be nontrivial in size, so they are not included in the body of the video response. Instead, the client can follow each of the references in `data.relationships.analysisResults.data` to make a request for the

```
{ "data": {
    "id":    "5981ce26b29a1453d5170219",
    "type":  "videos",
    "attributes": {
      "name":       "My␣Video␣30A",      // optional
      "status":     "done",              // or receiving, processing
      "dashSource": "http://example.com/mv30a.mpd" //optional
    },
    "relationships": {
      "analysisResults": {
       "data":  [
         { "id":   "5983023f495bbb5607b30609",
           "type": "analyses",
           "algorithmName": "recognition"
         }, { "id":   "59831ec0232be819c816b3fc",
           "type": "analyses",
           "algorithmName": "context"
       } ],
       "links": {"related": "/videos/599420d68b72a400189029ac/results"}
    } },
    "links": {"self": "/videos/599420d68b72a400189029ac"}
} }
```

***Listing 2***. *A sample video object in the analysis service*

desired analysis run. An example of an analysis run response obtained this way is given in Listing 3.

Rather than using an entirely custom format – which would be bound to be unfamiliar to application developers – the response format is adapted from the JSON API specification [24] but not fully conformant due to the lack of well-established JavaScript tooling. A fully HATEOAS [14, Ch. 5] API could be interesting for enabling applications that dynamically adapt to available courses of action, especially since artificial intelligence is already involved in the problem domain. However, no specific use cases were identified, so the API was designed as a more "simple" JSON API with the assumption being that applications are built following the API specification.

To summarize, the endpoints of the client-facing video analysis API described in this section are:

**`GET /algorithms`** retrieves the listing of available algorithms (there is also a corresponding *post* operation for algorithm registration, but that is service-internal rather than part of the API for application developers)

**`POST /videos`** creates a new video resource, which does not yet have any data, but the analyses to run are specified

**`POST /videos/{videoId}`** uploads the actual video data to run the previously specified analyses on; either in a single request or in multiple chunks as specified by the `Chunk-Number` header

```
1  { "data": {
2      "id":    "5983023f495bbb5607b30609",
3      "type":  "analyses",
4      "attributes": {
5        "results": [
6         { "timestamp": 0,
7           "score":     0.8,
8           "rect":      {"x": 0, "y": 0, "width": 350, "height": 220},
9           "classification": {"name": "rgn_vocab", "class": "bicycle"},
10        },
11        {...}, ...
12     ] },
13     "relationships": {
14       "video": {
15        "links": {"related": "/videos/599420d68b72a400189029ac"},
16        "data": {"type": "videos", "id": "599420d68b72a400189029ac"}
17       },
18       "algorithm": {
19        "links": {"related": "/algorithms/5981ca28be3091448eaa76c5"},
20        "data": {"id": "5981ca28be3091448eaa76c5", "type": "algorithms"}
21     } },
22     "links": {"self": "/analyses/5983023f495bbb5607b30609"},
23 } }
```

***Listing 3***. *Sample of analysis results in the analysis service*

**`GET /videos/{videoId}`** retrieves the video resource containing the algorithm listing, analysis status (pending or finished) as well as a link to each finished analysis

**`GET /analyses{analysisId}`** retrieves the results of a single analysis (for instance, “results of object recognition on video 1001” and “results of context recognition on video 1001” have distinct IDs)

Querying functionalities were discussed but ultimately not implemented, as it was unclear how exactly the responsibilities would be divided between the platform, other services and client applications. It does seem obvious that if a large repository of analysis results is accumulated, some kind of system for querying and statistically analyzing them would be of interest.

### 4.1.2 API implementation

JavaScript is the language *de rigueur* in web development at the moment. On the server side, it is usually executed in the Node.js runtime[1]. Shortcomings of the language include lack of static analysis; it is possible to run mistakes which in other languages could have been identified as soon as the code is written. For this reason, TypeScript[2], a syntactical superset of JavaScript was chosen. The addition of typings makes development easier as

[1]https://nodejs.org

[2]https://www.typescriptlang.org

immediate feedback is received for mistakes such as function parameters in the wrong order.

The implementations of REST APIs often contain many similar parts such as checking whether the input from the user is in the expected format, giving a helpful error message if it isn't – and documenting all this. The `swagger-node` module[1] for Node.js was chosen due to it allowing the same declarative definition to serve as the base for both documentation and implementation, reducing so-called "boilerplate" code. This streamlines the development workflow of the platform, as any implemented API endpoints are always documented. The module is comparatively obscure (download numbers three orders of magnitude lesser than those of the underlying Express.js), but while a cause for caution for systems to be maintained for years, was not deemed reason enough to implement the corresponding functionality by hand. This kind of framework choice is often down to a software engineer's intuition and word of mouth, and any rigorous comparisons of benefits and implications were not found for the purposes of this project.

In the research-and-development phase, the service is designed to be hosted in one place for experimentation by various actors. Security and authentication functionality are left out consciously, and would obviously need to be taken into account when developing a production-grade system. Some possible future approaches for the development and usage of the platform would be:

- Making a proper analysis-as-a-service platform, which would feature authentication, authorization and possibly usage limits. This kind of "outsourcing" would lower the threshold for new application developers.
- A more "strict" microservice approach, where only the analysis is performed by the system and any persistence and querying done by interfacing systems. According to the microservice principles, isolating the core functionality makes the service "pluggable", increasing reusability. The currently offered capabilities in the adjacent areas might not be sufficient for any particular real product; they exist only to have something in place for experimentation.
- Allowing the utilizing organizations to host the system in-house. This would enable using the platform as part of an internal microservice architecture, improving performance and making customization easier.

All the improvements listed above could be characterized as "everyday" software engineering, in that practically usable results could likely be achieved by developers without extensive studies.

[1] https://github.com/swagger-api/swagger-node

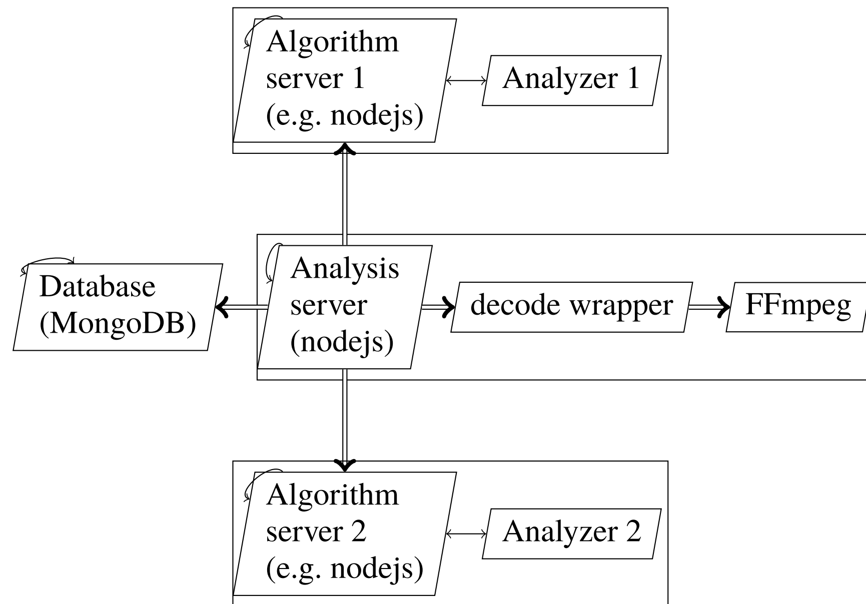


***Figure 5***. *Process view of the analysis service handling an analysis request. Self-loop indicates a process with a lifecycle spanning multiple video analyses.*

## 4.2 Video decoding and flow between platform and analyzers

The designed system has as its main actor components the platform engine, a database, and each integrated analyzer. Figure 5 highlights how the analysis server has the most central role, and the other components provide functionality to it and are called when requests are being served. The platform serves as the entry point for video, communicating with algorithm servers and a database, which run as services. When necessary, the analysis server starts decode processes and the algorithm servers start analyzer processes. The lifecycle of the analysis server processes is detailed in this section, and that of the analyzers below in Section 4.3.

### 4.2.1 Dependency resolution

When a video header is added, the required analysis order is determined based on dependencies of each involved algorithm. A directed, acyclic graph is required by the platform – feedback loops would greatly complicate the dynamic organization, but at the moment, no analysis flows involving those have been identified. Depending on what kind of dependencies the algorithm declare as needed, there may be situations where algorithm B requires results from algorithm A for frame $m > n$ in order to produce results for frame $n$, meaning that several passes over the whole video may be required. As a better understanding of the variety of algorithm dependency modes is gained, it may be possible to eliminate the need for multiple passes. This could be done e.g. by declaring limited frame dependency windows: if the most that analyzers need is a *local* forward-availability of results, the platform could keep a set amount of raw frames in memory at once, until all analyzers have finished with that window.

```
1  Sort(A)
2   P ← [[]]
3
4   C ← [a ∈ A : a.dep = ∅]
5   push(P, C)
6
7   while ∃a ∈ A : a ∉ P
8    do while C ←[a ∈ A : a ∉ P
9                    ∧∀d ∈ a.dep : d.alg ∈ P
10                             ∧(d.kind ≠ 'wholeVid' ∨d.alg ∉ P[P.length])
11               ]
12        do concat(P[P.length], C)
13      end
14      if C ← [a ∈ A : a ∉ P ∧ ∀d ∈ a.dep : d.alg ∈ P]
15        then push(P, C)
16      end
17   end
18
19   return P
```

***Listing 4.*** *Algorithm dependency resolution*

The algorithm used to determine the order in which analyses must run, given in Listing 4 is essentially a depth-first topological sort [11, p. 549–551]. To determine the order in which analyses must run, the algorithm dependencies are sorted topologically. Topological sorting is an operation which can be performed on directed, acyclical graphs: producing a vertex ordering such that for every edge $uv$, $u$ precedes $v$. Sorting of tasks based on their interdependencies is a classical application. The algorithm which was used (Listing 4) is largely the same as the depth-first topological sort given by Cormen [11], but with the added concept of "passes": going through the video must begin again when a "higher-order" dependency is encountered, so the algorithm keeps together algorithms which can be run in a single pass in order to minimize the number of passes.

The algorithm `Sort` sorts the given set of algorithms $A$. Since the graph must be complete, a preprocessing step in which all dependency links are followed and added to the same data structure must be performed. The first thing to be added to the result ordering $P$ on lines 4–5 is the set of "trivial" algorithms; ones which have no dependencies and can thus be run on a frame at any time. Note how $P$ is an array of arrays: a pass is represented as an inner array, and inside a single pass, the ordering is as defined simply by the dependencies. The loop on lines 7–14 simply iterates as long as there are algorithms not yet added to $P$. The first inner loop adds to the last *existing* pass algorithms whose all dependencies are satisfied algorithms run up until a single frame in the current pass. However, it does not add any algorithms depending on algorithms in the current pass and requiring the output for all frames. When such algorithms are encountered, a new pass is started for those on lines 14–16, and on the next iteration of the outer loop, the new pass is used.

### 4.2.2 Decoding and processes

The platform achieves video decoding with FFmpeg multimedia processing toolkit[1] with the command line `ffmpeg -copyts -debug_ts -i pipe:0 -f image2pipe -vcodec rawvideo -pix_fmt bgr24 -vsync passthrough`. The various flags mean

1. `-copyts -debug_ts`: do not normalize timestamps (so remuxing is later possible) and output timestamp information (so it can be used in result timestamps). The `-debug_ts` flag is not optimal in that its output format is not guaranteed between versions, bug the alternative `-vf showinfo` is very slow due to it calculating frame checksums.
2. `-i pipe:0`: get the data to demux and decode from `stdin` (the server process pipes the data to the FFmpeg process)
3. `-f image2pipe -vcodec rawvideo`: output raw image bytes rather than encoding or using any higher-level container format
4. `-pix_fmt bgr24`: sets the output color space, chosen due to the commonly used OpenCV library preferring this
5. `-vsync passthrough`: do not drop or duplicate frames to maintain exact frame rate

Since video decoding is a relatively complex operation, the system is designed to minimize the amount of times it has to be performed: the platform decodes the video, after which each analyzer receives raw bitmaps as input. Scaling to large input volumes may require developing a tailored decoder implementation using the libraries such as `libavformat` and `libavcodec` on which FFmpeg is based on. Currently, FFmpeg is simply run in its own thread as a CLI (command line interface) application, meaning that there is no granular execution control. It must be noted that this easy-to-implement multiple-process design means that before being placed in the shared memory area, frames have been written to two address spaces in the RAM; decoder and decode wrapper (see Figure 3). In the case where dependencies necessitate multiple passes over the video, decoding is done once for each pass – uncompressed image data is so large it is not feasible to preserve all frames between passes.

A simple Node.js server typically runs in one thread, with several CPU cores being utilized by launching separate instances of the server to serve multiple requests. It was noted, however, that the time to process even one request became very large: when one process was simultaneously transferring bytes from the FFmpeg pipe to files, as well as reading analysis results and maintaining a large result set, the single Node.js process became CPU-bound. This is largely due to the fact that garbage collection is used in Node.js.

[1]https://www.ffmpeg.org/

In lower-level programming, each programmer is responsible for allocating and freeing memory, end e.g. FFmpeg reserves a fixed amount of memory sufficient for a few frames, recycling the memory areas as older frames are no longer needed. The automatic memory management of Node.js, on the other hand, likely results in new memory allocations happening throughout the process, as there is no mechanism to specify the needed amount of memory. The effect becomes apparent when the raw image data is run from FFmpeg via the Node.js process before finally writing the data in frame-sized chunks to files in `tmpfs`. The performance problem was solved by using two processes to serve one request; the code handling decoding was isolated into its own "decode wrapper" process handling only starting FFmpeg and splitting the output to files. This arrangement removed the bottleneck for analyzing a single video at a time.

Another practical finding made during development was that contrary to the intuitively used guideline "the less time spent waiting, the faster the progress" led to suboptimal performance in the case of accepting data from the FFmpeg process. Reading data every time there was e.g. 64KB of it available on the pipe took over two times as long as waiting for a whole frame to be present on the pipe before reading and writing it to the `tmpfs`, the latter case limited only by the FFmpeg decode speed. This is likely a combination of the aforementioned memory allocation as well as system call overhead.

A simple improvement which could be made would be to control the decoder process priority based on the speed of the analyzers. The tested analyzers kept up well with the decoder, but another analyzer or a load with more concurrent requests might result in the `tmpfs` filling up. This could be prevented by not allowing the decoder to run too far ahead of analysis: halt the decoding when there is a buffer of, say, 100 unanalyzed frames and wake it up when it falls between 50.

### 4.2.3 Process and data flow

The process flow when an analysis request is being served varies slightly based on the use case and input format. First of all, the MP4 format has a `moov` header in the container which is required for decoding to start, and it is often placed at the end of the file. Because of this, in the case of MP4 video the compressed video is first received in its entirety before the decoding and analysis processes start. On the other hand, processing of TS inputs can occur in parallel to transfer from the client. If it was required for the clients to only send fast-start optimized MP4s, the "wait for the `moov`" path could be entirely eliminated. Allowing data transfer and video processing to occur concurrently would decrease the total time required for a single request, especially when the upload and analysis speeds are similar.

When processing of an analysis request starts, information on the requested algorithms and the corresponding dependency chain are loaded from the database by the platform. As the raw image data file for each frame becomes available for analysis, the platform starts

sending analysis requests (HTTP) for the analyzers. For the sake of utilizing parallelicity, multiple analyses may start in parallel for the same frame, and multiple frames may be processed simultaneously. Waiting is necessary when one algorithms requires output from another; in this case, analysis is requested from the latter algorithm only when the former has reported its analysis as complete by giving a success-indicating response to the analysis request.

After each frame is decoded by the platform, it is placed in a `tmpfs` file system. A `tmpfs` is a system which allows utilization of RAM using operations similar to file I/O [42], avoiding most of the complexity of lower-level IPC shared memory solutions. This design achieves the desired result of heterogeneous analyzers operating as different processes, while at the same time only writing each of the large frames to system memory only once. However, analyzers utilizing GPUs still need to copy the frame, potentially resulting in multiple transfers of the same data over the PCI-E bus (see Figure 3). The color space BGR24 was selected for usage as it was found to be the most commonly preferred format for analysis toolkits. The platform, as well as each of the analyzers, are placed in their own Docker container to answer the need of providing different software distributions to different analyzers to base upon. Therefore, a suitable RAM-backed file system path must be explicitly shared between the containers using "volume mounting", as by default containers share no data.

While a zero-copy memory sharing approach is used for data flow, control flows via HTTP interfaces, which is the more idiomatic way of communication for containers in a Docker container network. The platform exposes a `/algorithms` endpoint for algorithms to register their presence and dependencies at runtime. One limitation of the system as implemented is that there is no unregistration; in the event that an algorithm must be made temporarily or permanently unavailable, the algorithm database of the platform needs to updated manually. The removal of an analyzer is a rare enough operation that making the process more sophisticated by e.g. availability monitoring was not considered worth the development effort. Each analyzer is expected to expose a simple `/analyze` endpoint through which the platform gives the frame location, size and other information necessary for the analyzer to perform requested analyses.

Based on content negotiation, the analysis results will be delivered to the client either as a standalone JSON or inside a media container. For in-band analysis results, an SRT (SubRip Text format) subtitle track is used due to its simplicity. An SRT implementation with FFmpeg is very easy, as remuxing is both a basic use case of the program and does not require considerable processing power. MPEG-4 part 12 metadata tracks or part 14 scene descriptors might provide greater timing accuracy, but so far a clear need for these more sophisticated in-band metadata technologies has not been identified. Out-of-band results are more efficient when the client does not need multiplexed results or can do the multiplexing themselves, as the server does not need to send the video data the client already has.

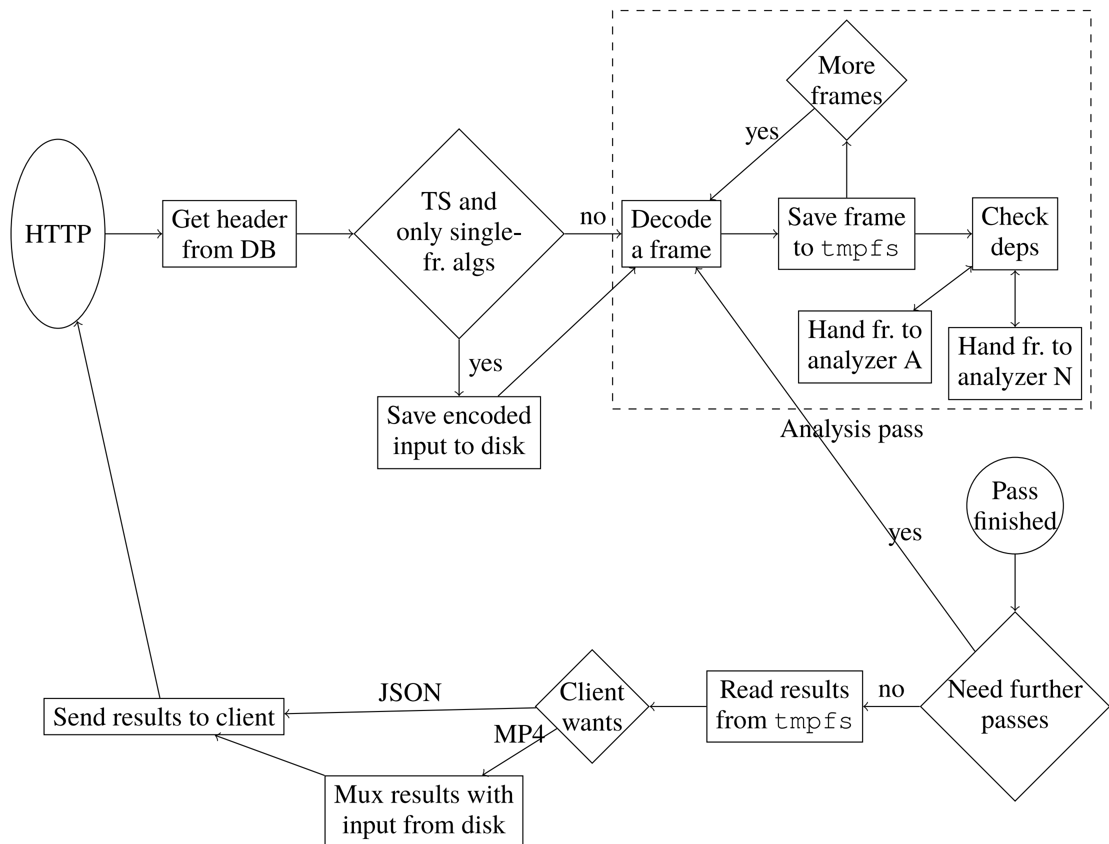


***Figure 6**. Data and control flow on the analysis pipeline when serving a single analysis request*

Figure 6 summarizes the data and control flows which have been described in this section, highlighting especially the decisions caused by input format, content negotiation and algorithm dependencies. Note that performing of analysis requests is not displayed in full detail: based on dependencies, multiple analysis requests may be made at once for a single frame, while some requests are made only previous are fulfilled. The check of whether there are pending analysis results in the current pass has also been omitted for simplicity of visualization.

## 4.3 Video analyzers

From the platform point of view, each analyzer is its own application taking an input and producing an output. The system architecture may be thought of as a kind of collection of microservices as on each end of simple pre-defined interface are independently developed components fulfilling one duty in relative isolation. Some interface-related technical requirements for the analyzers are still necessary, e.g. the HTTP interfaces and the loading of input frames with `mmap` (a system call faster than file I/O operations), but the internal architecture for analyzers is not specified. This means that algorithm developers are free to use different versions of various libraries or require different compiler versions, for example.

***Table 1**. Development view of the system: the layers indicating dependencies and the responsible entity for each component. *Algorithm developers can implement their own algorithm server, but the sample one supplied with the platform will likely cover most needs.*

| Layer | Off-the-shelf | Platform | Algorithm |
|---|---|---|---|
| Client-facing | swagger-node<br>Express.js | | |
| Service logic, registration | | engine<br>dep. resolution | algorithm server* |
| HTTP, filesystem access | Node.js | | |
| Persistence | MongoDB | | |
| Media helpers | | SRT<br>decode wrapper | |
| Media muxing/coding | FFmpeg | | |
| Analysis | | | analyzer |
| Analysis toolkit | e.g. OpenCV | | |
| Containerization | Docker | | |
| Operating system | GNU/Linux | | |

The division of systems into various programs and their associations is depicted in a development view, as described by Kruchten [26]. The relationship between platform and algorithm components, as well as the role of the supplied sample HTTP server wrappers, is illustrated in Table 1. Each component only depends on components on equal or lower levels, which makes the architecture less tightly coupled. It also shows which components of the analysis system are "off-the-shelf"; that is, not developed in this project or any of the associated ones under the 360VI umbrella.

When an algorithm application starts, it registers its presence with the platform. The information contained in the registration message to the platform consists of the following parts:

- the name of the algorithm, used for making analysis requests
- the version number of the algorithm, which should be incremented by the algorithm developer when there are changes after which its functionality can be considered different
- (optional) list of algorithm dependencies, with a minimum version number, as well as the type of the dependency (e.g. "current frame") – a maximum version number would be symmetrical, but in an R&D system it is assumed versions are kept up-to-date and the minimum version is sufficient as a "sanity check"
- (optional) port and host which the platform should use for analysis requests – by default, 80 and registration source host are assumed

The registration is done with an HTTP POST request to the platform, and this endpoint is described in the same platform Swagger documentation as the client-facing ones. Should

the address of the analyzer change, e.g. due to updating the software by replacing the container, registration must be performed again. An alternative approach might be to require that the name of an analyzer container match the name of the algorithm, in which case the Docker-idiomatic finding of services by name could be utilized. This would not remove the need for explicit registration, as the names of the present algorithms are not known in advance by the platform.

An analyzer is expected to act as a service, being ready to accept a request for the analysis of an image at any time. As each frame (and any dependencies as specified by the analyzer) becomes available, the platform issues an analysis request to the analyzer. How the process lifecycle is managed internally – a single process for all videos, or separate processes for each – is up to the algorithm developers, but it must be possible to handle multiple videos given in interleaved requests. The provided sample algorithm server starts a new process for each video to analyze. If an analyzer works more efficiently as a "server" and can process multiple videos concurrently, a simple configuration option to support this setup could be added to the sample algorithm server. In practice, this option would determine whether to spawn a new analyzer process when an analysis request with a new video ID is encountered, or to issue the analysis commands for all requests to the standard input of the same, singular process.

Analysis requests are sent via an HTTP endpoint as form data (`application/x-www-form-urlencoded`). Each analysis request contains, apart from the path of the frame data file, also a video identifier which may be used to differentiate interleaved videos. Other information in the analysis requests are frame size, needed to correctly interpret the data as a bitmap of a certain size, and frame number and timestamp, to allow the analyzer to produce result objects in the final format instead of leaving this to the platform. A sample Node.js server application providing a simple HTTP API-to-commandline adapter is provided with the platform and can be used once the analyzer is built to handle standard input in the specified format. Algorithm developers may also opt to use their own implementations of the interface if the analyzers have special requirements regarding e.g initialization and process lifecycles.

## 4.4 Software development process and deployment

As a result of the containerization, there are not many restrictions placed on each algorithm developer's workflow. For instance, not everyone needs to have access to each other's code repositories. Further, the developer of one algorithm need not necessarily know of other algorithms with utilize its output. Deploying an instance of the platform can be done either by fetching the latest code for the desired analyzers from their respective repositories and building the Docker images locally, or by using pre-built images provided by the algorithm developers, possibly on the Docker Hub[1] repository of images. Usage

[1]https://hub.docker.com/

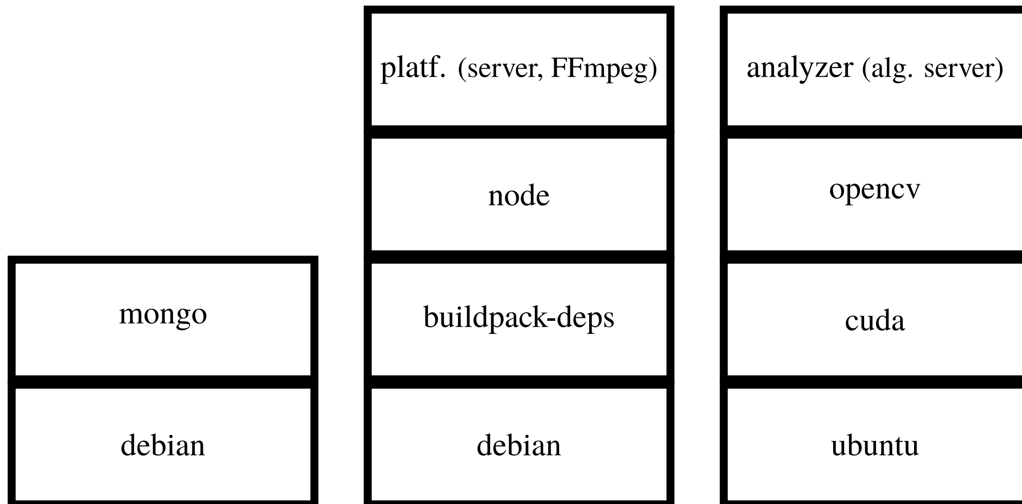


***Figure 7***. *Physical view of the system, indicating the single host on which it is designed to run as well as various containers. The analyzer on the right is only an example of a possible stack (and does not refer to any one particular OpenCV image).*

of the repository simplifies deployment of systems. However, if security is a concern to the analyzer implementer, the image files can also be distributed fr deployment through internal channels.

Placement of the platform component and each analyzer in Docker containers was chosen as the approach to use due to the differing needs of each analyzer. Each algorithm developer composes a Dockerfile for integration of the analyzer to the platform. The Dockerfile indicates the base image which serves as the starting point when the analyzer image is being composed, and specifies the necessary operations such as installation of packages and copying of files to achieve a well-defined outcome. An instance of MongoDB provided by an off-the-shelf Docker Hub library image runs in one container, the REST API and platform logic designed here in another, and each of the analyzers built by algorithm developers in its own.

The physical view as described by Kruchten places emphasis on placement on physical nodes, highlighting options for redundancy and scaling of the system. The system under design is simply deployed on a single host, but the physical view in Figure 7 shows the placement of the various software processes on containers as well as image stacks. While collections of “Dockerized” services may often be distributed to several physical hosts, the 360 video analysis service is designed for single-machine usage due to data locality. This way, neither multiple instances of video decoding nor large amounts of network traffic are necessitated.

A rather simple improvement to make if the platform must be scaled up to production-level volumes is implementing load balancing using video IDs. Although analysis of each video is sensible to perform on a single physical machine, different videos could easily be routed to be handled by different hosts. Being very much processing power-bound, the system

should be easier to scale than many web applications where database accesses are often a bottleneck.

One practical problem of using Docker is that tapping into a hardware GPU is somewhat more complicated than using only a CPU. Many analyzers use nVidia CUDA (Compute Unified Unified Device Architecture) processing, and this places requirements on not only containers, but also the Docker host. The easiest solution found was the usage of an nVidia-provided driver-agnostic CUDA image as the base for analyzer containers, and launching the analyzers using the `nvidia-docker`[1] Docker command wrapper which automates the mapping of GPU device files into the containers.

[1]https://github.com/NVIDIA/nvidia-docker

# 5. SYSTEM EVALUATION AND DISCUSSION

The analysis service that was designed is now evaluated in this chapter, which presents the observations made about the implemented system and discusses the outcomes of the thesis. Findings are contrasted with expectations based on the theoretical context. Based on the research question in Chapter 1, the relevant questions now are:

1. Does the platform design introduce a performance penalty compared to "bare" execution of algorithms?
2. Is integrating algorithms with the platform easy?

After briefly contrasting the functionality as implemented with the specification and earlier systems, the first question is examined in its own section, after which the second question and the architecture in general are discussed in the last one.

## 5.1 Functionality

The basis for evaluating the functionality of a system is determining whether it fulfills the requirements given to it. This section briefly revisits the requirements laid out in Section 4.1.2 and reflects upon them.

**Support videos in the MP4 [9] container format** implemented, provided by the FFmpeg multimedia toolkit. The current implementation always waits until the whole MP4 has been received; it might be useful to detect the special case where the `moov` atom is at the beginning of the file and decoding can start immediately.

**An API suitable for offline cases** implemented. As specified, the format of returned results can be either JSON (out-band) or MP4 (in-band), based on HTTP content negotiation.

**An API suitable for MPEG-DASH streams** not implemented. DASH streams were given a lower priority and the implementation project ended before there was time to realize them. However, the design does take into account the possibility of DASH operation, so implementation should not require great changes to what has already been done.

**A flexible data format** little feedback has been received from algorithm developers since the design stage, so the suitability of the format is difficult to evaluate. One remark made by an application developer was that timings in the format are too frame-oriented, and some applications would benefit from a format which defines for each observation its start and end times. However, the currently integrated analyzers do not as-is provide such information, and it would likely require some design work with the algorithm developers.

**A URL for analyzed videos** implemented. Results are stored in a database and can be retrieved afterwards. Each analysis result set is assigned a URL as well.

**Single-algorithm specification** implemented. It is sufficient for clients to specify one algorithm.

The intended purpose of the service was to provide a backend for applications using analyzed video. This goal was met to a certain extent. What makes evaluation somewhat challenging is that applications relying on the service not exist yet; the observations are mostly based on informal trials by application developers. The lack of discourse with the application developers means the actual fitness of the service has not yet been tested in actual setups. While the feedback received from trials by application developers has been positive, there has not been much of it.

The goal of acting as a platform for various algorithms was realized, and dependencies were resolved. While e.g. Wu *et al.* [46] and Lei *et al.* [28] had previously run video analysis in the cloud, the former using Docker specifically, the literature review found little work on the topic of interoperation of distinct algorithms and none on how to orchestrate such in a practical system. Thus, defining interfaces for analyzers to implement and declaratively forming a complete analysis system from various parts was a novel technique and possibly the most remarkable contribution of this thesis. However, the number of analyzers integrated so far is low enough that it is not yet possible to come to a final conclusion regarding the fitness of the solution.

## 5.2 Performance

A system is only as strong as its weakest component, so even when some part of it is theoretically not optimal, it may well be practically good enough and not worth improving upon. The scope of this performance evaluation is the implementation of the platform, not that of the individual algorithms, so a design which does not slow down the existing parts can be deemed adequate. Thus, it is important to keep the performance findings in context. This section reviews the performance of the system versus "bare" analyzers using two 22-second spherical video clips with equirectangular projection, a 1920x690 Advanced Video Coding (AVC) one ("small") and a 3840x1920 High Efficiency Video Coding (HEVC) one ("large"). The test system is a server with Intel Xeon E5-2630 v4 processors (10 cores each, 2.2 GHz), two nVidia GeForce GTX 1080 Ti GPUs (1.48GHz, 11 GB) and 128 GB of RAM.

First, to provide some perspective to the numbers, Table 2 shows the time taken to merely decode the two video clips. As any video needs to be decoded in order to be analyzed, the decoding time provides a theoretical lower bound for the lead time of a video analysis process. The tests were run with FFmpeg, which is known to be very efficient in most cases. "Decode" refers to piping frames to `/dev/null`, i.e. "throwing away" the raw images, and "copyback" to piping them to a `tmpfs`. Each version of the tests was

***Table 2**. Decoding times of the video samples with FFmpeg 3.4.2*

| -hwaccel | Target | -pix_fmt conversion | Input | Avg (s) | Stdev (s) |
|---|---|---|---|---|---|
| (CPU) | /dev/null | – | small | 1.22 | 0.05 |
| (CPU) | /dev/null | – | large | 4.87 | 0.11 |
| (CPU) | /dev/null | bgr24 | small | 3.06 | 0.18 |
| (CPU) | /dev/null | bgr24 | large | 12.10 | 0.15 |
| (CPU) | tmpfs | – | small | 1.69 | 0.00 |
| (CPU) | tmpfs | – | large | 6.51 | 0.19 |
| (CPU) | tmpfs | bgr24 | small | 4.44 | 0.22 |
| (CPU) | tmpfs | bgr24 | large | 17.68 | 0.26 |
| cuvid | /dev/null | – | small | 1.27 | 0.05 |
| cuvid | /dev/null | – | large | 4.80 | 0.13 |
| cuvid | /dev/null | bgr24 | small | 2.97 | 0.07 |
| cuvid | /dev/null | bgr24 | large | 11.95 | 0.09 |
| cuvid | tmpfs | – | small | 1.69 | 0.02 |
| cuvid | tmpfs | – | large | 6.39 | 0.17 |
| cuvid | tmpfs | bgr24 | small | 4.56 | 0.07 |
| cuvid | tmpfs | bgr24 | large | 17.34 | 0.64 |

run both on CPU only as well as using GPU acceleration. FFmpeg version 3.4.2 with the command `ffmpeg [-hwaccel cuvid] -copyts -i input.mp4 -f image2pipe -vcodec rawvideo [-pix_fmt bgr24] -vsync passthrough - > (/dev/null | /tmp/frames.dat)` was used. Tests were run three times and the average and standard deviation of runtime are given.

Even when decoding on CPU and never transferring data over the PCI-E bus, the extra copy of frames from the decoder process to shared memory clearly slows down the execution. This result suggests that a "theoretically optimal" system would have to handle memory allocation manually, in order to have the frames in shared memory from the start. Also presented are the decoding times including YUV->BGR color space conversion, as the algorithms require input without chroma subsampling. It is clear that the conversion slows down decoding significantly. Perhaps somewhat unexpectedly the CPU approximately matches the GPU-integrated decoder. Although dedicated ASICs are more efficient at their specific task, a general-purpose CPU will perform like the ASIC if large enough. The FFmpeg HEVC decoder scales well, utilizing 16 of the 20 CPU cores with. Since the decoding can performance-wise be done on either the CPU or the GPU, and different analyzers use various combinations of CPU and GPU time, a more thorough look into the performance of the whole system with the various combinations might be warranted.

The design has the raw frame data written three times in the system memory: in the FFmpeg space, server decode wrapper space and the shared `tmpfs`. Testing only the server and decode wrapper writing frames to `tmpfs` in isolation, without analyzers running, the runtime for the 4K clip was 18 seconds, so the performance impact of the extra copy seems negligible for 4K resolution at least. However, when running the system at scale with multiple requests served concurrently, even memory bandwidth may be at more of

***Table 3**. Analysis times of video clips using the service versus the analyzer alone*

| Run as | Input | Avg (s) | Stdev (s) |
|---|---|---|---|
| standalone | small | 21.8 | 0.5 |
| standalone | large | 21.0 | 0.3 |
| integrated | small | 20.6 | 0.3 |
| integrated | large | 21.8 | 0.5 |

a premium, so the utility of a custom decoder implementation cannot be ruled out. A more sophisticated test setup might automatically load the service with multiple requests to establish the performance impact of the memory copies at scale, but this kind of testing was not performed as the stated goal for the research-and-development phase was to serve singular requests within a short time.

Actual system performance was tested with the yolo360 analyzer. Since the system designed is a platform for integrating multiple analyzers in a pipeline, tests with multiple analyzers would be more enlightening, but no other analyzers were available for testing yet. Table 3 compares the analysis times using the platform ("integrated") versus using the analyzer in isolation ("standalone"). The requests to the platform were sent from the same physical host, so effects of network bandwidth are negligible. The platform as implemented was using the CPU for decoding. The standalone test shows the performance when the frames are simply handed to the analyzer at once, instead of it waiting for each individual request from the platform.

At least for this particular analyzer, the platform manages to match the analysis performance without slowing down the process: the limiting factor of running analysis service is the analyzer, not the platform. While it may not have been immediately obvious the platform can provide data fast enough for the analyzer not to starve, it is not surprising that the purely CPU-based platform does not slow down the analyzer, which performs most of its heavy operations on a GPU. However, the picture could change from the one seen here when there are multiple analyzers sharing the PCI-E bus, which is currently utilized inefficiently. Obviously, multiple analyzers would also mean less CPU and GPU time for each of them, but this is an insurmountable fact, not a property of the platform. Even a single analyzer faster than yolo360 might provide some insight to the platform performance; for instance, a hypothetical algorithm that does not need color space unpacking could be fast enough to make the platform design a bottleneck (although on the other hand, that would also decrease the amount of data).

During the development, the yolo360 analyzer was at one point heavily CPU-bound. According to the developer of the yolo360 analyzer, the CPU load was caused by resizing each image to the detector input size. Although yolo360 was later optimized to be far more efficient with scaling, the question of image sizes was raised. At the moment, a single copy of a frame is provided to each analyzer in the input size. There are different analyzers with different reactions to input image size in both runtime and quality of results, and this

should be explored to find the optimal approach to integration. One option might be to to "negotiate" one or a few frame sizes and have the platform provide these, possibly reducing duplicate work by analyzers. This introduces the need of keeping track of required and preferred input sizes – likely with a mechanism similar to announcing algorithm dependencies. The GPU could possibly improve performance of image resizing, whether done by the platform or individual analyzers.

## 5.3 Architecture

Evaluation of architecture is somewhat more difficult than functionality. Whereas functionality requirements are rather binary in nature, architecture goals require more qualitative assessment. Furthermore, a bad architecture can decrease the value of a software artifact far more than a simple missing functionality, which can always be added. The most remarkable architectural decisions were:

1. How to orchestrate the interoperation of dependent algorithms?
2. How to implement the interoperation of homogeneous components without performance penalties?
3. How to allow integration into products?

These are discussed based on the requirements listed in section 2.4.2.

Integration went well. The context recognition analyzer was integrated by experienced software engineers in less than one man week and the supplied sample algorithm server was sufficient. Object recognition required only surface-level changes to be plugged into the platform. The runtime registration of algorithms forming a processing pipeline dynamically was based on tested architecture patterns and the experiences with this solution were positive. The current version which supports two dependency modes and may cause multiple passes of the video is not optimal; it does not support the 16-frame "windows" used by context recognition. To achieve the best results, new dependency modes such as windows need to be added. This may require somewhat more complicated implementation, but the architecture already takes into account the possibility of more modes.

Data flow in the platform was implemented such that analyzers receive a single copy of the data in the memory, so the possibility of duplication in step 3 of the process in Section 3.2 isn't realized. However, while FFmpeg proved to be surprisingly well-integrable for an application best known as targeting end users, it was still a separate CLI application. Thus, there is one extra copy in the main memory, as the platform logic does not use the same memory area as the decoder. While that does not slow the system down in practice due to the DDR bandwidth, each analyzer utilizing the GPU makes its own copy of the images in VRAM, meaning also multiple copies of the data over the PCI-E bus, which may in the future form a bottleneck to the system. On the other hand, no persistent storage is used for raw image data, as it was noted to be obviously too slow.

The choice not to utilize GPU resources proved to be sensible based on the results currently known. Because the known analyzers are GPU-bound, it makes sense to introduce as little extra GPU load as possible, although it should be noted that a considerable of the decoding work on the GPU is performed by an ASIC not used for other purposes. Even if a platform-owned "data manager" GPU process is introduced, or if the whole analysis system architecture is developed into a tightly-integrated single-process one, decoding on a powerful CPU will likely make sense in order to utilize all available computing resources.

The implementation of analyzer communication using a shared `tmpfs` directory was a rather "custom" practical solution. Although earlier literature suggests more rigid inter-process communication methodology with abstractions such as message passing, these were deemed to be too demanding to implement, especially considering they would have required extensive modifications to the analyzers rather than a simple wrapping layer. No particular problems were encountered, but the solution can be characterized as "ad hoc" which means it may seem unfamiliar for new developers coming into touch with the project. The system memory side was found to be performant enough, but the performance benefits that could be gained from multiple analyzers using the same VRAM need to be studied, as this side of the current implementation is far further removed from optimal. Implementation of either a purpose-built decoder binary or especially a VRAM memory manager would require more development work than done in this project. If resources for a larger-scale development project become available, a redesign of the involved analyzers to operate as components of a single software artifact could be useful for realizing highest possible performance. The filter pipeline of FFmpeg could be useful, either in practice or as inspiration.

Usage of Docker to achieve isolated execution environments on a single physical computer is an established practice in the industry. The 360VI platform can rather easily be deployed by launching a few Docker containers from pre-built images. However, even the management of systems of multiple containers is typically complex enough to be automated in any nontrivial systems, or at least supported with scripts. The analysis service developed needs the network settings and `tmpfs` volume mappings for intra-platform communication to be manually set with Docker command-line parameters, which is obviously inconvenient, but still a great improvement over building the system from scratch. The automation of managing containers is more of an unimplemented feature than a deficiency in the design. Once Docker container networks get complicated, with duplication and persistence requirements, usage of orchestration tools such as Kontena[1] becomes an interesting option. This question is in no way unique to this product, and should be interesting to anyone employing microservices.

The evaluation of the architecture was rather informal. Technically speaking, the architecture does mostly meet the goals set to it. In order to arrive at a better assessment of the system, a more rigorous review should be conducted, ideally involving stakeholders. One

[1]https://kontena.io/

methodology option might be Decision-Centric Architecture Reviews by van Heesch *et al.*, in which the main architecture decisions are explicated and evaluated [20]. Even without a proper review, it can be noted that the usage of shared directories and Docker containers to implement the process seemed somewhat unfamiliar to many developers involved, requiring a considerable amount of explanation to provide even an overview of how integration occurs. Another overall observation which can be made is that Conway's law [10] applies: "organizations which design systems …are constrained to produce designs which are copies of the communication structures of these organizations." In this case, analyzers were developed at different organizations, and the platform in yet another, so the end result was loosely coupled.

# 6. CONCLUSION

The objective was to build a product for the 360 Video Intelligence ecosystem, offering video analyses as a service over the web and integrating various analyzers into a single system. The service platform now exists, and implementation work of applications building on it can take place.

The design with the platform handling decoding and using `tmpfs` for inter-process communication does not slow down analyses, as the platform performs its task of providing image data faster than the tested analyzer can process it. The operations running on the platform, while not trivial, do not greatly impact the analyzer since the former runs exclusively on CPU and the latter mostly on GPU. However, these conclusions comes from testing with a single analyzer in isolation, and the results could be different with multiple analyzers. One of the motivations for this platform was to allow interoperation of algorithms, and while this was implemented, practical performance could not be tested as no algorithm using the results of another has yet been integrated to the platform. It is not entirely certain whether the platform would become a bottleneck in a more complex pipeline due to the redundant work of each GPU-utilizing analyzer copying image data from system memory over the PCI-E bus into its own VRAM address space. On the other hand, no implementation where data sharing is handled optimally previously existed, so the new design does not introduce a bottleneck. Since the performance of the platform is good to the extent that could be determined as of now and analyses are provided as fast as the analyzer runs, the performance of the system meets the goals that were assigned to it.

A sound architectural base for the analysis service was formed. The design of Dockerized analyzers dynamically registering themselves with the platform, which provides them with the image data and exposes an analysis interface over the internet, answers the needs that existed between analyzers and the API. The dynamic registration, inter-analyzer result passing and automatic dependency handling are novel approaches of the platform. The `tmpfs` data sharing mechanism chosen due to being very easy to implement, while rather ad-hoc compared to more sophisticated structures described in literature, lead to no problems. The integration architecture as a whole is robust, supporting dependency modes well, and makes integration of analyzers easy. The description of the design includes some relatively simple features that could be added with some development effort to improve the service, such as result querying, support for DASH streams and a more complete SaaS implementation with authentication requirements.

The most critical need for further work is moving forward with producing applications utilizing the service. While application developers from involved companies have tested the API, the needs arising from real products should show the way for further development,

as the role of the platform is to provide a service for client applications. Techniques that could be explored for increased performance would be to also share GPU memory, define dependency windows to reduce needs for analysis passes, and to find efficient ways to meet the image scaling needs of all involved algorithms. If performance at scale is desired, a more tightly coupled software architecture with various analyzers and the decoder running in the same process would be necessary to utilize all available computing resources to the fullest extent is possibly needed. In practice, this would likely involve lower-level programming with manual memory management, as well as using CUDA computing even for the platform in order to ensure that no redundant operations are performed. One similar system to review is the FFmpeg video filtering pipeline. Any effort to reach production scale should test performance when handling a large number of requests. This testing was consciously left out, but stress tests could reveal e.g. the Node.js garbage collection as opposed to manual memory management growing to be an issue when the system has to handle large numbers of videos at once.

Looking at the project from a wider perspective, it is clear that the project organization had an effect on the premises. A single large organization building the same products in-house could have resulted in an assumption of a single analyzer product, with all algorithms developed conforming to same technical guidelines. This would likely have affected performance-related design and lessened the need for dynamic registration. The sensibility of much work is subject to advances in computing hardware, and if one day mobile devices are powerful enough to run complex video analyses on the fly, many current assumptions become invalid.

The current platform is a useful intermediate step towards end-to-end 360 video analysis products. It provides a previously missing well-defined, extensible architecture for interfacing between analyzers with a novel approach to analyzer collaboration and dependency handling. The platform can act as a useful tool for algorithm developers figuring out the major questions regarding interoperation, before they move onto software architecture and performance optimization. It can also serve as a demo backend for client applications while they are being developed. The system achieves good enough performance and integration of analyzers is easy.

# REFERENCES


[1] I.F. Alexander, A taxonomy of stakeholders: Human roles in system development, *International Journal of Technology and Human Interaction*, Vol. 1, Iss. 1, 2005, pp. 23–59.

[2] A.T. Bahill, F.F. Dean, Discovering system requirements – Chapter 4, in: Sage, A.P., Rouse, W.B. (eds.), *Handbook of Systems Engineering and Management*, John Wiley & Sons, New York, 1999, pp. 175–220.

[3] J. Bigun, *Vision with Direction: A Systematic Introduction to Image Processing and Computer Vision*, Springer-Verlag, Berlin, 2006, 396 p.

[4] D. Buntinas, G. Mercier, W. Gropp, Data transfers between processes in an SMP system: Performance study and application to MPI, in: *Proceedings of the 35th IACC International Conference on Parallel Processing (ICPP), Columbus, August 14–18, 2006*, IEEE Computer Society, Los Alamitos, pp. 487–496.

[5] S.D. Burd, *Systems Architecture*, 6th ed., Cengage Learning, Boston, 2010, 631 p.

[6] M.C. Calatrava Moreno, T. Auzinger, General-purpose graphics processing units in service-oriented architectures, in: *Proceedings of the 6th IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Kauai, December 16–18, 2013*, IEEE Computer Society, Los Alamitos, pp. 260–267.

[7] N. Chapman, J. Chapman, *Digital Multimedia*, 3rd ed., John Wiley & Sons, Chichester, 2009, 724 p.

[8] T.P. Chen, H. Haussecker, A. Bovyrin, R. Belenov, K. Rodyushkin, A. Kuranoc, V. Eruhimov, Computer vision workload analysis: Case study of video surveillance systems, *Intel Technology Journal*, Vol. 9, Iss. 2, 2005, pp. 109–118.

[9] *Coding of audio-visual objects – Part 14: MP4 file format*, International Organization for Standardization, ISO/IEC 14496-14, Geneve, 2003, 10 p. + app. 1 p.

[10] M.E. Conway, How do committees invent?, *Datamation*, Vol. 14, Iss. 4, 1968, pp. 28–31.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 2nd ed., MIT Press, Cambridge, 2001, 1180 p.

[12] A. DuVander, JSON's eight year convergence with XML, 2013. Available (accessed on 8.1.2018): https://www.programmableweb.com/news/jsons-eight-year-convergence-xml/2013/12/26

[13] *Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats*, International Organization for Standardization, ISO/IEC 23009-1, Geneve, 2012, 107 p. + app. 37 p.

[14] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, dissertation, University of California, Irvine, 2000, 162 p. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[15] W. Fischer, *Digital Video and Audio Broadcasting Technology: A Practical Engineering Guide*, 3rd ed., Springer, Heidelberg, 2009, 811 p.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, 1994, 395 p.

[17] C. Gregg, K. Hazelwood, Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, in: *Proceedings of the 12th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, April 10–12, 2011*, IEEE, Piscataway, pp. 134–144.

[18] T. Gruber, Ontology, in: Liu, L., Özsu, M.T. (eds.), *Encyclopedia of Database Systems*, Springer Science+Business Media, New York, 2009, pp. 1963–1965.

[19] A. Hammar, *Analysis and Design of High Performance Inter-core Process Communication for Linux*, master's thesis, Uppsala University, UPTEC IT 14 020, 2014, 43 p. + app. 4 p. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-236686

[20] U. van Heesch, V.P. Eloranta, P. Avgeriou, K. Koskimies, N. Harrison, Decision-centric architecture reviews, *IEEE Software*, Vol. 31, Iss. 1, 2014, pp. 69–76.

[21] J. Hestness, S.W. Keckler, D.A. Wood, GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors, in: *Proceedings of the 11th IEEE International Symposium on Workload Characterization (IISWC), Atlanta, October 4–6, 2015*, IEEE Computer Society, Los Alamitos, pp. 87–97.

[22] C.H. Hong, I. Spence, D.S. Nikolopoulos, GPU virtualization and scheduling methods: A comprehensive survey, *ACM Computing Surveys*, Vol. 50, Iss. 3, 2017, pp. 1–37.

[23] *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, RFC Editor, RFC 7231, June 2014, 101 p.

[24] JSON API Specification v1.0, API specification, 2015. Available (accessed on 27.10.2017): http://jsonapi.org/format/1.0/

[25] Y.K. Kim, C.S. Jeong, Large scale image processing in real-time environments with Kafka, in: Wyld, D.C., Nagamalai, D. (eds.), *Proceedings of the 6th AIRCC International Conference on Parallel, Distributed Computing Technologies and Applications (PDCTA), Zürich, January 2–3, 2017*, AIRCC Publishing Corporation, Computer Science & Information Technology 63, Chennai, pp. 207–215.

[26] P.B. Kruchten, The 4+1 view model of architecture, *IEEE Software*, Vol. 12, Iss. 6, Nov. 1995, pp. 42–50.

[27] C. Lee, W.W. Ro, J.L. Gaudiot, Boosting CUDA applications with CPU–GPU hybrid computing, *International Journal of Parallel Programming*, Vol. 42, Iss. 2, 2014, pp. 384–404.

[28] Z. Lei, H. Lin, H. Du, W. Shen, H. Zhang, Y. Lei, A Spark-based study on the massive video-receive IO issues, in: *Proceedings of the 2nd S&E International Conference on Computer Science and Applications (CSA), Wuhan, November 20–22, 2015*, IEEE Computer Society, Los Alamitos, pp. 30–36.

[29] L. Li, W. Chou, W. Zhou, M. Luo, Design patterns and extensibility of REST API for networking applications, *IEEE Transactions on Network and Service Management*, Vol. 13, Iss. 1, 2016, pp. 154–167.

[30] F.L. Lin, B.F. Smith, Y. Wang, Pacing of multiple producers when information is required in natural order, Pat. US 6,055,558 A, appl. US 08/653,908, May 28 1996 (Apr 25 2000), 2000, 16 p.

[31] K. Liu, B. Liu, E. Blasch, D. Shen, Z. Wang, H. Ling, G. Chen, A cloud infrastructure for target detection and tracking using audio and video fusion, in: *Proceedings of the 28th IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Boston, June 7–12, 2015*, IEEE, Piscataway, pp. 74–81.

[32] K.Y. Liu, T. Zhang, L. Wang, A new parallel video understanding and retrieval system, in: *Proceedings of the 11th IEEE International Conference on Multimedia and Expo (ICME), Singapore, July 19–23, 2010*, IEEE, Piscataway, pp. 679–684.

[33] T.G. Mattson, B.A. Sanders, B.L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley Professional, Boston, 2004, 384 p.

[34] S. Mittal, J.S. Vetter, A survey of CPU-GPU heterogeneous computing techniques, *ACM Computing Surveys*, Vol. 47, Iss. 4, 2015, pp. 1–35.

[35] R. Olson, J. Calmels, F. Abecassis, P. Rogers, NVIDIA docker: GPU server application deployment made easy, 2016. Avail-

able (accessed on 21.2.2018): https://devblogs.nvidia.com/nvidia-docker-gpu-server-application-deployment-made-easy/

[36] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, GPU computing, *Proceedings of the IEEE*, Vol. 96, Iss. 5, 2008, pp. 879–899.

[37] C. Pahl, Containerization and the PaaS cloud, *IEEE Cloud Computing*, Vol. 2, Iss. 3, 2015, pp. 24–31.

[38] H. Pan, L.H. Ngoh, A.A. Lazar, A buffer-inventory-based dynamic scheduling algorithm for multimedia-on-demand servers, *Multimedia Systems*, Vol. 6, Iss. 2, 1998, pp. 125–136.

[39] S.J. Park, J.A. Ross, D.R. Shires, D.A. Richie, B.J. Henz, L.H. Nguyen, Hybrid core acceleration of UWB SIRE radar signal processing, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, Iss. 1, 2011, pp. 46–57.

[40] C. Pautasso, O. Zimmermann, F. Leymann, RESTful web services vs. "big" web services: Making the right architectural decision, in: *Proceedings of the 17th IW3C2 International World Wide Web Conference (WWW), Beijing, April 21–25, 2008*, pp. 805–814.

[41] S. Potluri, H. Wang, D. Bureddy, A.K. Singh, C. Rosales, D.K. Panda, Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication, in: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Shanghai, May 21–25, 2012*, IEEE Computer Society, Los Alamitos, pp. 1848–1857.

[42] C. Rohland, H. Dickins, M. Kosaki, Tmpfs, in: *Linux kernel documentation*, 2010, p. filesystems/tmpfs.txt. Available (accessed on 15.11.2017): https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt

[43] G. Salvaneschi, P. Eugster, M. Mezini, Programming with implicit flows, *IEEE Software*, Vol. 31, Iss. 5, 2014, pp. 52–59.

[44] G. Teodoro, T.M. Kurc, T. Pan, L.A.D. Cooper, J. Kong, P. Widener, J.H. Saltz, Accelerating large scale image analyses on parallel, CPU-GPU equipped systems, in: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, May 21–25, 2012*, IEEE Computer Society, Los Alamitos, pp. 1093–1104.

[45] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil, Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, in: *Proceedings of the 10th SCO2 Computing Colombian Conference (CCC), Bogotá, September 21–25, 2015*, IEEE, Piscataway, pp. 583–590.

[46] R. Wu, Y. Chen, E. Blasch, B. Liu, G. Chen, D. Shen, A container-based elastic cloud architecture for real-time full-motion video (FMV) target tracking, in: *Proceedings of the 43rd IEEE Applied Imagery Pattern Recognition Workshop (AIPR), Washington, D.C., October 14–16, 2014*, IEEE, Piscataway, pp. 6–13.

[47] M.T. Yang, R. Kasturi, A. Sivasubramaniam, A pipeline-based approach for scheduling video processing algorithms on NOW, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, Iss. 2, 2003, pp. 119–130.

[48] Youtube Data API v3: Videos, Google Inc., API documentation, 2017. Available (accessed on 26.10.2017): https://developers.google.com/youtube/v3/docs/videos

[49] H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang, Y. Gao, Container based video surveillance cloud service with fine-grained resource provisioning, in: *Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD), San Francisco, June 27 – July 2, 2016*, IEEE Computer Society, Los Alamitos, pp. 758–765.