Heterogeneous Hardware Support for Apiary

by

Nathan Weckwerth

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2021

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

© Massachusetts Institute of Technology 2022. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 6, 2022

Certified by ....
Michael Stonebraker
Adjunct Professor
Thesis Supervisor

Accepted by ....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Heterogeneous Hardware Support for Apiary

by

Nathan Weckwerth

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Function-as-a-Service (FaaS) platforms are an appealing option for developers because they save time and money by eliminating the work spent managing application servers. Apiary is a novel FaaS platform which performs extremely well on data-centric tasks by tightly integrating computation and storage layers, eliminating the time spent transferring data between the two. Moreover, Apiary's robust provenance system and straightforward programming model provide compelling reasons for developers to use it for both data-centric and compute-intensive tasks. In this paper, we detail a general architecture for using Apiary's asynchronous programming model to implement compute-intensive tasks as external services. These external services are free to make usage of specialized hardware such as GPUs, which provide extremely good performance for many typical compute-intensive tasks such as machine learning inference.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor

Acknowledgments

I would like to thank both Michael Stonebraker and Çağatay Demiralp for all of their advice and direction for the project. I would also like to thank Qian Li and Peter Kraft for their continuous guidance and support.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction 13

1.1 Apiary . . . 14

1.1.1 Programming Model . . . 14

1.1.2 Exactly-Once Semantics . . . 14

1.1.3 Provenance Capture . . . 15

1.2 Heterogeneous Hardware . . . 15

1.2.1 Data Centers and Computation . . . 15

1.2.2 Machine Learning . . . 16

1.3 Contributions . . . 16

2 Related Work 19

2.1 FaaS Platforms . . . 19

2.1.1 Benefits . . . 19

2.1.2 Data-Centric FaaS Tasks . . . 20

2.2 Heterogeneous Hardware . . . 20

2.2.1 Integration with DBMS . . . 20

2.2.2 Task Scheduling . . . 22

2.2.3 Machine Learning Deployment Systems . . . 23

3 System Architecture 25

3.1 Overview . . . 25

3.2 Primary Stored Procedure . . . 26

3.3 External Process . . . 27

4 Computational Test Selection 29

4.1 Image Classification 29

4.2 Datasets 29

4.2.1 MNIST 30

4.2.2 ImageNet 30

4.2.3 COCO 32

5 Results 35

5.1 Features 35

5.2 Inference Performance 36

5.2.1 MNIST 36

5.2.2 ImageNet 37

5.2.3 COCO 38

6 Closing Remarks 41

6.1 Discussion 41

6.2 Future Work 42

6.3 Conclusion 42

List of Figures

3-1 System architecture. Upward lines represent flow of data originating from the underlying database, downward lines represent flow of information computed by external process. 26

3-2 Structure of a message sent to the external process. Each message includes an identifier for which task it is for and, if necessary, an identifier for which piece of the puzzle it is when pooling data from multiple stateless functions. 28

4-1 A handwritten 1 from the MNIST dataset. 30

4-2 Two examples of hierarchy chains in WordNet, with example images from ImageNet representing each synset. Figure taken from [37]. . . . 31

4-3 An example image from the COCO dataset, with bounding shapes around various objects shown. Figure taken from [42]. 32

5-1 Comparing the external process on a CPU and GPU against TensorFlow-Serving and the baseline time spent executing the model for MNIST testing. 37

5-2 Comparing the external process on a CPU and GPU against TensorFlow-Serving and the baseline time spent executing the model for ImageNet testing. 38

5-3 Comparing the external process on a GPU against TensorFlow-Serving and the baseline time spent executing the model for COCO testing. . 40

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

5.1 Overhead incurred for Apiary with external process running on a GPU against TensorFlow-Serving and raw model execution time for MNIST example. 36
5.2 Overhead incurred for Apiary with external process running on a GPU against TensorFlow-Serving and raw model execution time for ImageNet example. 38
5.3 Overhead incurred for Apiary with external process running on a GPU against TensorFlow-Serving and raw model execution time for COCO example. 39

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

In recent years, FaaS platforms have found support in both industry [1, 2, 3] and research [4, 5, 6] applications. The principal attraction of FaaS platforms is that they save developer time and money by eliminating the work spent managing application servers and providing a straightforward pricing model based on auto-scaling, since the FaaS platform provider can manage the application servers and charge according to usage.

Broadly, we can divide FaaS applications into two types according to how they spend the bulk of their resources. They may be either compute-intensive, such as image or video processing, or data-centric, such as microservices and web-serving applications. Existing FaaS platforms perform poorly on the latter type of applications because they typically disaggregate computation and storage layers. Several of the most popular industry FaaS platforms, such as AWS Lambda, Azure Functions, and Google Cloud Functions, were created by cloud vendors who offer cloud storage platforms. For these vendors, disaggregating computation and storage makes sense as it allows the FaaS platform to directly integrate with their cloud storage platform. Unfortunately, this approach limits performance on data-centric applications because it incurs the cost of a round trip between the computation and storage layers for each operation.

1.1 Apiary

Apiary [7] is a novel FaaS platform which tightly integrates computation and storage layers. This allows Apiary to perform extremely well on data-centric tasks by eliminating the time spent transferring data between the two layers.

Apiary is the result of a joint effort by MIT, Stanford, CMU, Google, and VMware to use a distributed database management system (DBMS) as the foundation of a novel FaaS platform [8]. By tightly integrating computation and data layers, Apiary leverages its underlying highly-performant DBMS to provide impressive performance on data-centric tasks. Several data-centric workloads have been experimentally shown to exhibit higher performance running on Apiary than on popular FaaS platforms such as Openwhisk [9] and Boki [5]. Apiary guarantees that all functions run as ACID transactions, provides exactly-once semantics, and offers automatic data provenance.

1.1.1 Programming Model

Apiary provides a straightforward programming model to developers. Functions are written in a high-level language (Java), embedding SQL queries to a relational database (VoltDB [10]). These functions are compiled to stored procedures, which run natively as database transactions.

1.1.2 Exactly-Once Semantics

Apiary offers exactly-once semantics for general graphs of functions. Existing FaaS platforms either require all functions to be idempotent or use external transaction managers which incur a high performance cost. Apiary solves this problem by logging executions from stored procedures and using a novel dataflow analysis algorithm to keep the overhead at less than 5% [7].

1.1.3 Provenance Capture

Apiary provides automatic provenance capture by recording all data which is read or written by each database operation. Provenance information is grouped by functions, providing application-specific context. This makes it easier for Apiary to handle common queries in large-scale FaaS deployments, such as:

- Which operation most recently updated this record?
- Which records were produced by this data pipeline?
- Does this operation have the right to access this data?

Apiary records both the full history of function execution and the exact information read from and written to the database, feeding this information to an analytical database (Vertica [11]). This analytical database can then be queried with standard SQL queries to answer provenance questions which might be difficult to answer in other systems that need to query multiple sources to gather this information.

1.2 Heterogeneous Hardware

In the last few decades, the computation landscape has drastically changed. In particular, the scale has changed significantly; as a concrete example, the MIT Supercloud has approximately 10,000 cores and a hundred terabytes of main memory [12]. With this increase in scale comes a fundamental shift in how users run compute tasks. Fifty years ago, a programmer might run a computation locally on a single, trusted machine. The compute tasks of today may be much larger, running on hundreds or even thousands of machines. Moreover, modern clusters often contain multiple different types of hardware specifically optimized for purposes such as simulation, data analysis, and machine learning.

1.2.1 Data Centers and Computation

Computing hardware choices now include options such as graphics processing units (GPUs) and tensor processing units (TPUs), as well as field programmable gate arrays

(FPGAs) and multiple integrated cores (MICs) and many more [13, 14]. Certain tasks can be performed more efficiently on some of these specialized hardware options; this efficiency translates into lower cost and/or better performance for the end user. Modern data centers make full use of this wide array of hardware, making it more important than ever to embrace the power of heterogeneous hardware.

1.2.2 Machine Learning

Machine learning tasks are ideal candidates for utilizing specialized hardware such as GPUs. The dramatic increase in usage of machine learning models aligns with the similar increase in usage of specialized hardware. The massive, rigid computations which occur in a typical machine learning model can greatly benefit from the architecture of the GPU, often by orders of magnitude over running on a CPU. Popular machine learning frameworks such as TensorFlow [15] and PyTorch [16] provide native support for running models on GPUs, making it easy for an end user to take advantage of the computational power of the GPU.

1.3 Contributions

As described earlier, existing FaaS platforms perform poorly on data-centric tasks because their computation and storage layers are disaggregated. Apiary is a novel FaaS system that performs well on these tasks by tightly integrating computation and storage, and its robust provenance system and straightforward programming model provide compelling reasons for developers to use Apiary to deploy both data-centric and compute-intensive tasks. However, there are several reasons to implement compute-intensive tasks differently from data-centric tasks. Since Apiary tightly integrates computation and storage, it must scale according to the data storage requirements, which may not directly align with the computation requirements. Moreover, compute-intensive tasks may take drastically more time than data-centric tasks, and it is impractical to lock a partition of the underlying database for a significant amount of time.

Motivated by these reasons, we detail a general architecture for using Apiary's asynchronous programming model to implement compute-intensive tasks as external services. Doing so avoids locking a partition of the underlying database for a significant amount of time, only doing so when directly retrieving or inserting data. These external services are free to make use of specialized hardware such as GPUs or TPUs, which provide extremely good performance for many typical compute-intensive tasks such as machine learning models. We conduct performance tests on several typical machine learning inference tasks and show that the overhead incurred by Apiary is negligible compared to the time spent on computation for these tasks.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Related Work

2.1 FaaS Platforms

In recent years, FaaS platforms have found much support in both industry [1, 2, 3] and research [4, 5, 6] applications. For example, the most widespread option, AWS Lambda, has 3.5× as many function invocations as two years ago, and other popular cloud storage providers such as Azure and Google Cloud have an increasing proportion of their users also making use of their FaaS platforms (Azure Functions and Google Cloud Functions, respectively) [17]. This significant growth can be attributed to the strong appeal of FaaS platforms for saving user time and money.

2.1.1 Benefits

A traditional web serving application architecture consists of three distinct tiers:

- A frontend where the user interacts with the application
- A stateless tier of web servers which respond to user requests and implement application logic
- A stateful backend for application data storage

Each of these tiers presents another layer of infrastructure to manage, and the time and money required for upkeep of this infrastructure can quickly rise to unreasonable levels. FaaS platforms provide an appealing alternative to this architecture

by reducing the scale of the infrastructure an engineer must manage. In particular, a FaaS platform replaces the middle tier of web servers with functions implementing application logic and the stateful backend with a cloud storage system. This eliminates the need for engineers to manage any failures occurring in the middle tier as well as the need to auto-scale this tier according to the load, minimizing both time and money spent.

2.1.2 Data-Centric FaaS Tasks

Increasingly, FaaS platforms are used for data-centric applications such as microservices and web-serving applications [17, 6]. Some examples include:

- A social network where users create posts and interact with others' posts [18]
- A store where users view items, add them to their cart, and check out [19]
- A hotel reservation service where users find nearby hotels and reserve rooms in them [20]

We have already discussed how existing FaaS platforms incur a significant performance penalty because they disaggregate computation and storage layers. For example, a simple function performing a point database update in popular FaaS platform Openwhisk takes over a millisecond to run to completion, with only 23 µs of this time spent actually performing query execution; over 98% of the time is spent connecting to and communicating with the remote database [7].

2.2 Heterogeneous Hardware

2.2.1 Integration with DBMS

With the ever-increasing prevalence of specialized hardware, much work has been done on integrating standard DBMSs with specialized hardware in recent years [13, 14, 24, 25]. The ability to harness both a powerful DBMS and modern specialized hardware promises to yield impressive results.

One such application of this integration is using the combination of a standard DBMS and specialized hardware as the foundation of a new operating system. This is a direction explored by TabulaROSA [13], which uses a DBMS to manage operating system state. Broadly, operating systems such as Linux are quite old and, by today's standards, fairly slow, which motivates using a high-performance DBMS instead. Moreover, Linux is particularly bad at managing heterogeneous hardware, as this was simply not relevant when it was created. TabulaROSA defines key operating functions in terms of high-level mathematical semantics which can be directly translated into DBMS operations and can be formally verified to be correct. This shows that an operating system using a DBMS is possible on many different types of hardware. Initial performance tests of forking, a fundamental operating system function, showed promising results [13].

Another application is using specialized hardware to build a more performant query processing engine for a DBMS. Traditional relational DBMSs are generally "pull-based" in the sense that the data needed for each computational thread is pulled from internal storage. In contrast, a "push-based" DBMS passively relies on a streaming mechanism which feeds it data. Recently certain "push-based" DBMSs have become popular, such as Aurora [21], Qpipe [22], and DataPath [23]. These DBMSs tend to become limited by computation when the data arrival rate is quite high. This fact motivates taking advantage of specialized hardware to build a more performant query processing engine [24]. Initial testing on a proof of concept yielded promising results, as a query processing engine running on a GPU can outperform a comparable system on a CPU by an order of magnitude or more on certain operations [24].

There has also been work to develop a DBMS with built in support for acceleration via specialized hardware. One such project is CoGaDB [14], a main-memory DBMS which supports GPU acceleration and is optimized for online analytical processing (OLAP) workloads. CoGaDB implements efficient algorithms for common database operations for both CPUs and GPUs and uses a self-tuning optimizer framework [25] to build an optimizer which is hardware-oblivious and efficiently schedules tasks according to processor availability. CoGaDB can quickly adapt to take full advantage

of the power of the underlying hardware and shows great promise for a DBMS with built in support for specialized hardware.

2.2.2 Task Scheduling

As the hardware landscape grows more diverse, data centers are increasingly faced with jobs which have complex dependencies and heterogeneous resource demands. Significant work has been done to investigate how to handle these jobs efficiently while taking full advantage of the available hardware.

Much research on this problem considers the simpler case where each task can only be executed on a specific type of hardware. Even in this simpler case, finding an optimal schedule is intractable [26]; however, an optimal schedule can be approximated. Graphene [27] is a cluster scheduler aimed at jobs with strict heterogeneous resource demands which utilizes an offline and an online component to schedule tasks efficiently. The offline component identifies "troublesome" tasks, i.e. those which are long running or with complex resource demands, and schedules those first; the online component uses heuristics to tightly pack tasks and to trade-off between fairness and speed of completion. With this approach, Graphene finds a solution within 1.75$\times$ the cost of an optimal solution in the worst case, outperforming more naive schedulers such as the Apache Tez [28] scheduler [27].

Generalizing the problem to account for the fact that many tasks may be performed on multiple types of hardware makes the scheduling problem at hand much more complicated. Some work in this area focuses on using an initial test suite to gather various information such as latency, transfer bandwidth, and computational power about the available hardware, and combining this information with an online learning approach utilizing placement heuristics to schedule tasks efficiently [29]. This approach is hardware-aware in the sense that the information gathered is manually curated with knowledge of the underlying hardware and this information is crucial to the resulting schedule. Other approaches are hardware-agnostic, relying on online machine learning based models to adapt to the underlying hardware [25, 30].

2.2.3 Machine Learning Deployment Systems

With popular machine learning frameworks such as TensorFlow and PyTorch providing native support for using specialized hardware such as GPUs to train and utilize models, several systems have been developed to make it easier to deploy these models on specialized hardware. Systems such as TensorFlow-Serving [31], Clipper [32], and Amazon Sagemaker [33] make it easy for end users to deploy their models as microservices to a GPU for inference. Deployed models may be interacted with using REST or RPC APIs. Users may utilize a pipeline to retrain and redeploy models in the background, ensuring the deployed model stays current.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

System Architecture

Although the high-level idea of calling an external process is straightforward, there are several performance concerns and infrastructure requirements that inform these architectural decisions. In this chapter, we detail the general architecture for using Apiary's asynchronous programming model to implement compute-intensive tasks as external services.

3.1 Overview

In Apiary, users write code in the form of stored procedures written in Java which use embedded SQL queries to communicate with the underlying VoltDB instance. These stored procedures are run natively as ACID transactions in the database. For compute-intensive tasks, the computation time may be quite long relative to data-centric tasks, meaning that it may be unreasonable to lock the underlying database partition for the duration of the task. To circumvent this problem, users can make use of Apiary's asynchronous programming model, which allows them to asynchronously invoke stateless functions which are not tied to the database. These stateless functions connect via TCP to an external process which manages computation. Figure 3-1 shows a high-level overview of this architecture.

The entry point for an end user to invoke a compute-centric task is a single primary stored procedure. This stored procedure retrieves data from the database

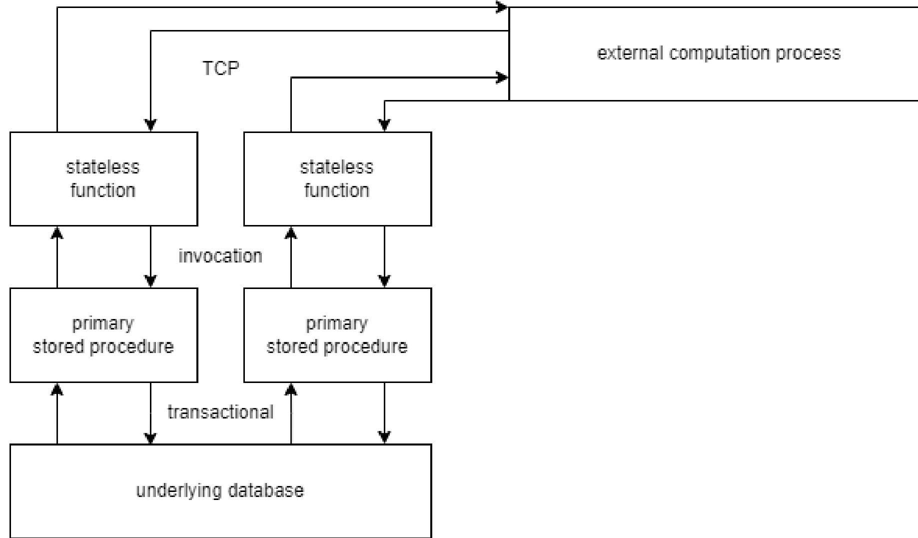

Figure 3-1: System architecture. Upward lines represent flow of data originating from the underlying database, downward lines represent flow of information computed by external process.

and asynchronously invokes a stateless function. This stateless function does not have access to the database and thus does not need to lock the underlying partition. It connects over TCP to a long-running external process managing computation. After computation occurs, the computed information is transferred back through the chain and eventually reaches the original stored procedure, at which point it may be inserted into the database. Multiple stored procedures may be invoked, leading to multiple stateless functions making TCP connections with the external process.

3.2 Primary Stored Procedure

The primary stored procedure performs the following steps:

1. Invokes another stored procedure to fetch the desired data from the database.

2. Asynchronously invokes a stateless function and gives it the retrieved data. When the stateless function receives a response from the external process managing computation, it returns the computed information.

3. Invokes another stored procedure to insert the computed information into the

database.

Note that steps (1) and (3), the points which make direct contact with the database, run as ACID transactions.

Even for a single computational task, data may need to be transferred using multiple invocations of the primary stored procedure. This is due to some fundamental size limitations of VoltDB arising from the fact that VoltDB is a main-memory database, namely:

- The maximum size of any column is 1MB;
- The maximum size of any row is 2MB;
- The maximum size of any stored procedure return value is 50MB.

Moreover, the current Apiary implementation restricts stored procedures to at most 1MB due to the way it serializes tasks. In any case, whether the limit is 1MB or 50MB one can easily envision computational tasks which require significantly more data than this to be transferred, which is not an issue since the external process will accept as many connections as needed.

3.3 External Process

The external process is a long-running process which accepts TCP connections from potentially many stateless functions. This has a number of crucial advantages over doing computation directly in stateless functions:

- The external process can initialize computation structures, e.g. load saved machine learning models, before receiving any connections, eliminating a potentially massive delay in computation.
- The limitation on the amount of information returned by a stored procedure means that data from multiple stateless functions may need to be pooled to run a single computation.
- Transferring data over TCP means the external process may be run in any high level language, not just Java, and may potentially be run on any machine.

Figure 3-2: Structure of a message sent to the external process. Each message includes an identifier for which task it is for and, if necessary, an identifier for which piece of the puzzle it is when pooling data from multiple stateless functions.

Figure 3-2 shows the structure of a message sent to the external process. The external process determines which task the data is for and how to assemble multiple pieces of data based on the identifiers at the start of the message.

Chapter 4

Computational Test Selection

The architecture we have described thus far is quite general, and one can envision using this architecture to run many types of compute-centric tasks. However, to investigate the performance of the architecture, we focus on machine learning inference for image classification. In this chapter, we discuss the datasets and the models we have selected for performance testing.

4.1 Image Classification

Image classification is a standard problem handled by many machine learning models. The typical approach to this task is to treat it as a supervised learning problem, i.e. using labeled examples to train a model to classify arbitrary images. This approach naturally requires a great deal of labeled data, so well-constructed datasets are valuable assets.

4.2 Datasets

We have selected three popular image classification datasets with images at different scales. Here we will provide a discussion of the datasets and the models we use for classifying images from each of them. We believe that our models represent realistic use cases for a user deploying an image classification model.

4.2.1 MNIST

The MNIST dataset [34] is perhaps the most iconic image classification dataset. It was created by the National Institute of Standards and Technology (NIST) and consists of 70,000 images of handwritten digits generated by hundreds of writers including both Census Bureau employees and high school students. Each image is black and white and is fairly small at 28x28 pixels.

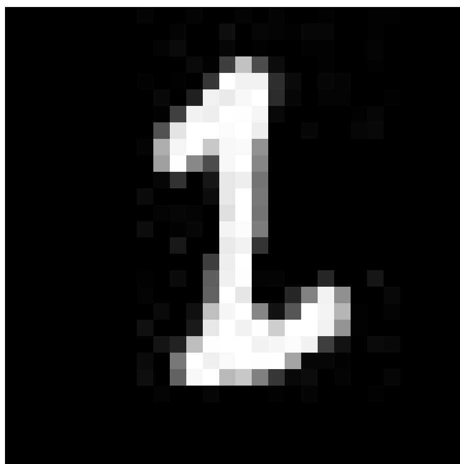

Figure 4-1: A handwritten 1 from the MNIST dataset.

There has been much work in developing models to recognize the different digits. Recent models [35] have attained an accuracy as high as 99.8%, which is quite impressive given that the images in the dataset are so small. Some of the images in the testing dataset are nearly unreadable [36] and may prevent even the most advanced models from reaching perfect accuracy.

4.2.2 ImageNet

The ImageNet dataset [37] is an image dataset organized by the WordNet hierarchy [38]. WordNet is a large lexical database of English words and phrases, with synonyms grouped together into *synsets*. These synsets may have various relationships with other synsets, the most common being the super-subordinate relationship. For example, the “chair” synset has a super-subordinate relationship with the “armchair” synset. These relationships naturally structure synsets into hierarchy trees.

There are around 80,000 synsets in WordNet which correspond to nouns. The ImageNet dataset aims to provide around 1,000 quality-controlled and human-annotated images for each of these synsets, and currently provides well over 20,000 different image categories. These images are taken from various sources on the web and vary significantly in size, with an average resolution of around 470×390; most models resize the images to a standard 256×256 or 224×224 before using them.

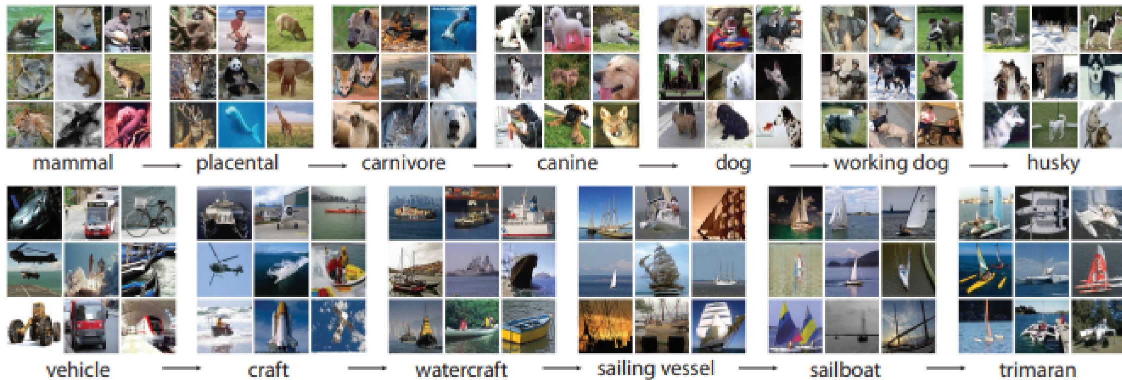

Figure 4-2: Two examples of hierarchy chains in WordNet, with example images from ImageNet representing each synset. Figure taken from [37].

Training a model to perform with high accuracy on the ImageNet dataset can be quite costly for a few reasons:

- Many of the synsets are quite similar to each other and will be difficult to distinguish even for a human; for example, "lapdog" and "toy dog".
- The images, even scaled down to 256×256 or 224×224, are significantly larger than the MNIST images, and as such the relevant computation for training is significantly more expensive.

In addition, it is not the goal of this paper to develop novel or incredibly accurate models. Thus, we have taken several steps to reduce the training expense while maintaining a reasonably accurate model. Instead of using all 20,000+ image categories, we use only the standard 1,000 categories that are used by the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), an annual image classification competition run by the creators of ImageNet. More importantly, we perform transfer learning using a pre-trained headless model [39]; that is, we start with a pre-trained model

which extracts a small, information-dense number of features from the image, and we train a layer to map these features to our categories. The pre-trained model uses the MobileNetV2 [40] architecture and is provided by Google on TensorFlow Hub. This model was originally trained on the ImageNet dataset. It takes a 224×224 image as input and outputs a feature vector with 1,280 features; on top of this, we add a layer acting as the classification head, mapping these features to the 1,000 categories from ILSVRC. This approach saves a great deal of time on training and allows the model to perform quite well.

4.2.3 COCO

The COCO (Common Objects in Context) dataset [41] is an image dataset prepared by Microsoft. It consists of 328,000 images of 91 distinct and easily recognizable object types such as "clock", "horse", and "traffic light". Not only is each image labeled with each object type found in it, but each individual object is marked with a bounding shape. The images have an average resolution of around 640×480, and models typically resize them to standard sizes such as 512×512 before using them.

Figure 4-3: An example image from the COCO dataset, with bounding shapes around various objects shown. Figure taken from [42].

Similarly to the ImageNet dataset, there are high costs in training a model to perform with high accuracy. In fact, the images in the COCO dataset are slightly

bigger than the images in the ImageNet dataset, and identifying bounding shapes for objects in an image is significantly more complicated than just detecting their presence. Thus for simplicity we directly use a pre-trained model provided by TensorFlow on TensorFlow Hub. The model [43] uses the CenterNet [44] architecture with the Hourglass [45] backbone and was originally trained on the COCO dataset. It takes a 512×512 image as input and outputs data identifying the objects in the images and bounding boxes for each object.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Results

5.1 Features

A user deploying machine learning models using Apiary can take advantage of several features:

- Data sent to the external process can easily be used for either inference or training the model itself, by simply changing the task ID referenced in Figure 3-2 to tell the external process which purpose to use it for.

- Inference and training can be performed immediately by the external process, or wait until a certain batch size appears. This allows users to balance work done by the external process between different tasks or prioritize a specific task over others.

- The models themselves can be stored in Apiary and loaded into the external process by simply storing and sending 1MB pieces of the serialized model. Combined with Apiary's robust provenance system, this allows users to easily track not only which data was used for training and inference, but also the history of the model itself as it is trained on new data or experiences architectural changes from the end user.

5.2 Inference Performance

For performing inference, each invocation of the primary stored procedure fetches as many images as can comfortably fit in 1MB. In all three datasets, the sizes we have chosen to scale the images to are less than 1MB, so there is no need to break an image into multiple pieces in the database.

The external process runs on a virtual machine equipped with a GPU configured using Google Cloud Platform [46] and uses the GPU to perform inference. To measure inference performance, we provide comparisons against running on a CPU and against the time spent directly running the model, as well as a comparison against TensorFlow-Serving [31], a popular machine learning deployment system mentioned earlier. Of course we anticipate some overhead on directly running the model, but even against TensorFlow-Serving there will be some overhead incurred primarily by invoking stored procedures. For each comparison we record performance numbers for 1, 10, 100, and 1,000 invocations of the primary stored procedure. The performance numbers provided are the median of five runs.

5.2.1 MNIST

The MNIST model we use has a single hidden layer with 512 neurons. This is in line with the scale of other MNIST models [34] and while simple, it is already enough to perform with 96% accuracy or more on the testing set.

Since the MNIST images are only 28×28 and they are black and white, each only takes up 784 bytes, meaning 1,200 images can comfortably fit into a single MB and thus we fetch this many with each invocation of the primary stored procedure. The external process simply returns the classification of each image.

	1 Invocation		1,000 Invocations	
	TF-Serving	Model Execution	TF-Serving	Model Execution
Apiary	28.8%	72.7%	10.6%	23.8%
TF-Serving	N/A	34.1%	N/A	11.9%

Table 5.1: Overhead incurred for Apiary with external process running on a GPU against TensorFlow-Serving and raw model execution time for MNIST example.

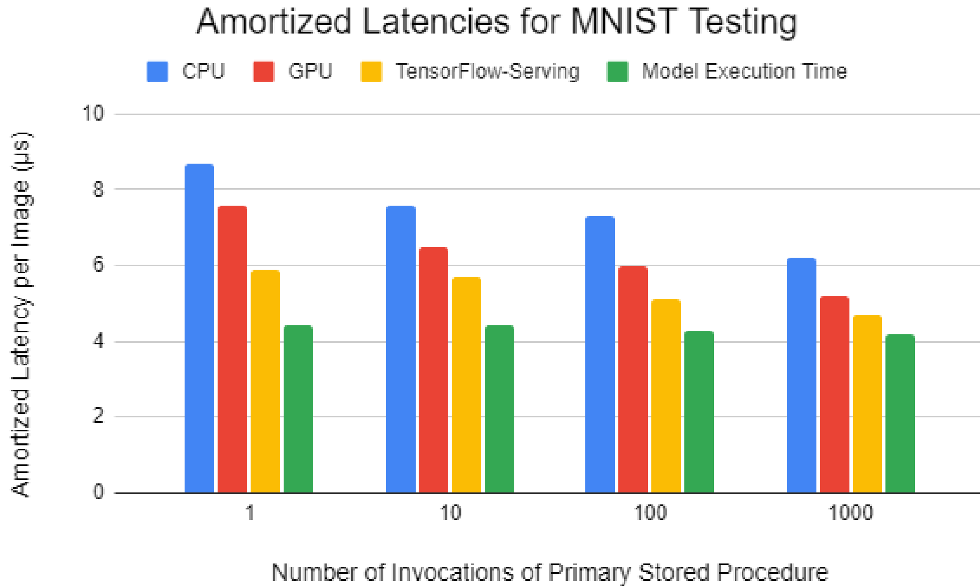

Figure 5-1: Comparing the external process on a CPU and GPU against TensorFlow-Serving and the baseline time spent executing the model for MNIST testing.

Table 5.1 shows the results of the comparison. We can see that we pay a relatively high overhead compared to both model execution time and TensorFlow-Serving, especially when only invoking a single stored procedure. We can also see in Figure 5-1 that the GPU is not dramatically better than the CPU for this example. Both of these facts are understandable as the computation here is relatively simple compared to the other inference tasks, with inference taking only a few microseconds for each image. Even so, the overhead is not too large; with 1,000 invocations of the primary stored procedure the overhead against TensorFlow-Serving falls to just over 10% and the overhead against the base model execution time is 23.8%.

5.2.2 ImageNet

The ImageNet images have been scaled to 224×224 and they are color images, meaning 6 images can comfortably fit into a MB. Thus we fetch 6 images with each invocation of the primary stored procedure. The external process simply returns the classification of each image.

Table 5.2 shows the results of the comparison. Here the overhead against model

	1 Invocation		1,000 Invocations	
	TF-Serving	Model Execution	TF-Serving	Model Execution
Apiary	9.1%	26.3%	5.0%	16.7%
TF-Serving	N/A	15.8%	N/A	11.1%

Table 5.2: Overhead incurred for Apiary with external process running on a GPU against TensorFlow-Serving and raw model execution time for ImageNet example.

execution time and TensorFlow-Serving is, unsurprisingly, drastically better than for the MNIST example; Figure 5-2 shows the performance numbers in more detail. Note that for this example the GPU outperforms the CPU by a factor of 3 due to the computation being more complex. For our system using Apiary on a GPU, even when invoking a single stored procedure the overhead against TensorFlow-Serving is less than 10%, dropping to just 5% at 1,000 invocations.

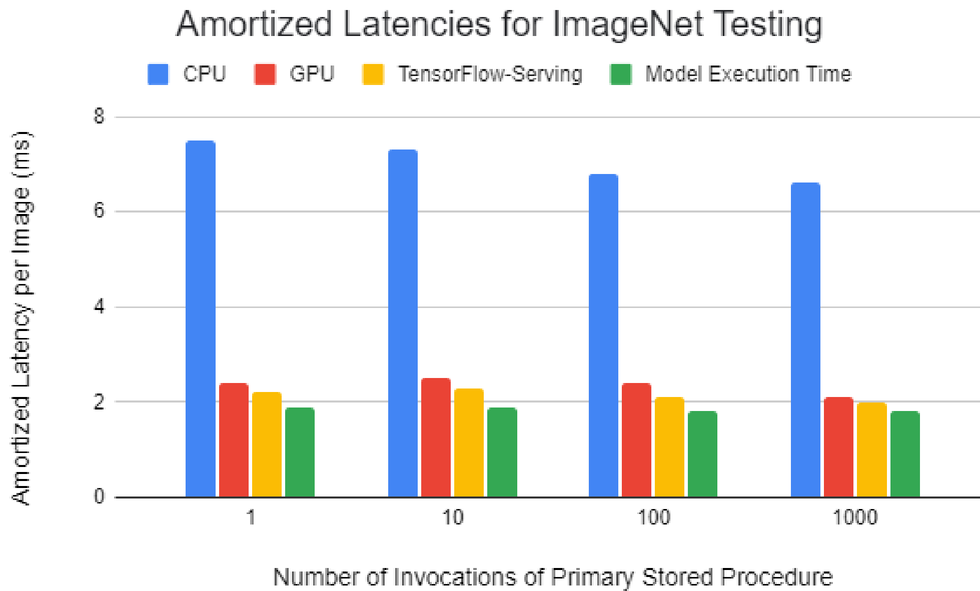

Figure 5-2: Comparing the external process on a CPU and GPU against TensorFlow-Serving and the baseline time spent executing the model for ImageNet testing.

5.2.3 COCO

The COCO images have been scaled to 512×512 and they are color images, meaning only a single image fits within a MB. Thus each invocation of the primary stored

procedure fetches a single image. For this example the external process returns more than just a simple classification for each image; instead it returns data corresponding to multiple objects identified in the image and estimated bounding boxes for each. Functionally this does not change the implementation much as the information is still just sent back to the stateless function.

	1 Invocation		1,000 Invocations	
	TF-Serving	Model Execution	TF-Serving	Model Execution
Apiary	5.1%	6.7%	3.3%	9.8%
TF-Serving	N/A	1.5%	N/A	6.3%

Table 5.3: Overhead incurred for Apiary with external process running on a GPU against TensorFlow-Serving and raw model execution time for COCO example.

Table 5.3 shows the results of the comparison. Here the overhead against model execution time and TensorFlow-Serving is better than both of the previous examples. Figure 5-3 shows the performance numbers in more detail, although the CPU numbers are omitted as for this test the CPU is orders of magnitude slower due to the complexity of the computation. In all cases for this model the overhead incurred is less than 10%, even when comparing directly against the model execution time, and the overhead against TensorFlow-Serving is 5% or less.

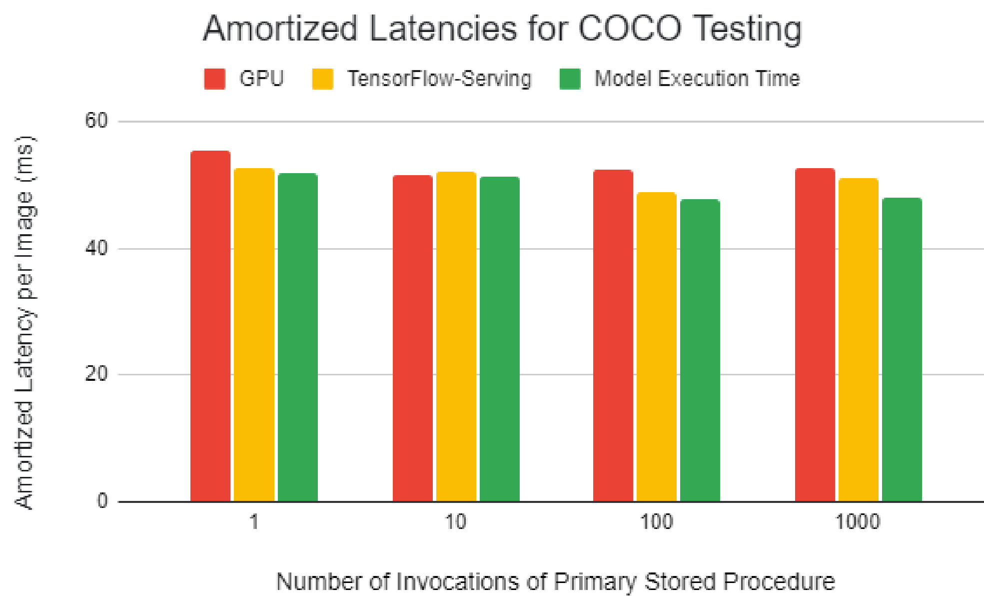

Figure 5-3: Comparing the external process on a GPU against TensorFlow-Serving and the baseline time spent executing the model for COCO testing.

Chapter 6

Closing Remarks

6.1 Discussion

The initial testing shows very promising results, with overheads of less than 10% when compared with TensorFlow-Serving for both the ImageNet and COCO datasets, which contain significantly larger images than the MNIST dataset. Even for the MNIST dataset, with enough invocations of the primary stored procedure the overhead drops to just over 10%. These numbers make sense as the expected overhead from retrieving data from the database and inserting computed information into it should be quite small, and as the complexity of the computation grows this overhead should look smaller and smaller by comparison. The absolute worst case in our performance testing occurs when performing a single invocation of the primary stored procedure for the MNIST dataset, where the performance is just under 30% worse than that of TensorFlow-Serving. Even so, in this case over half the time is spent running the model itself. In general, we conclude that the performance hit incurred by deploying machine learning models in Apiary is negligible compared to the cost of computation. This, combined with Apiary's robust provenance system and straightforward programming model, makes Apiary a good choice for users who want to deploy compute-centric tasks.

6.2 Future Work

One obvious goal for future work is to build more infrastructure taking advantage of Apiary's robust provenance tracking. For example, one can imagine a simple frontend showing the history of a model saved into the database, or possibly displaying which data in the database has been used for training or inference. This could provide a more positive user experience.

Another avenue for future work is to explore computationally-intensive tasks other than machine learning. While we have focused on this topic, there are a wealth of other computationally-intensive tasks that could be explored, and the architecture we have described could surely be modified to accommodate these tasks.

Finally, another interesting question to explore is how the architecture could be improved if the size restrictions presented by VoltDB were relaxed. While these restrictions are understandable, they do affect the architecture significantly as one cannot simply grab all the relevant data using a single invocation of the primary. This could come in the form of changes to VoltDB itself allowing larger outputs from stored procedures.

6.3 Conclusion

Apiary is a novel FaaS platform which performs extremely well on data-centric tasks by tightly integrating computation and storage layers, eliminating the time spent transferring data between the two. Apiary also offers a robust provenance system and straightforward programming model, making it an appealing option for developers. We detail a general architecture for using Apiary's asynchronous programming model to implement compute-intensive tasks as external services. These external services are free to make usage of specialized hardware such as GPUs, which provide extremely good performance for many compute-intensive tasks such as machine learning models. Initial performance testing on machine learning inference tasks shows promising results, giving hope that Apiary will prove a good fit for all types of

computationally-intensive tasks.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] AWS. AWS Lambda Customer Case Studies, 2021. https://aws.amazon.com/lambda/resources/customer-case-studies/.

[2] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 205–218.* USENIX Association, July 2020.

[3] Google Cloud. 2022. https://cloud.google.com/serverless/whitepaper/.

[4] OpenWhisk. Apache OpenWhisk, 2022. https://openwhisk.apache.org/.

[5] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Symposium on Operating Systems Principles (SOSP 21).* USENIX Association, November 2021.

[6] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pages 152–166, 2021.*

[7] P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, D. Kumar, D. Cho, J. Li, R. Redmond, N. Weckwerth, B. Xia, M. Cafarella, G. Graefe, J. Kepner, C. Kozyraki, M. Stonebraker, L. Suresh, X. Yu, M. Zaharia, "Apiary: A DBMS-backed transactional function-as-a-service framework," under submission.

[8] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. Cafarella, V. Gadepally, G. Graefe, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, L. Suresh, and M. Zaharia, "DBOS: A DBMS-oriented operating system," in *Proceedings of the VLDB Endowment 15, pages 21–30, 2021.*

[9] K. Djemame, M. Parker, and D. Datsev, "Open-source serverless architectures: an evaluation of apache openwhisk," in *13th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2020, Leicester, United Kingdom, pages 329–335.* IEEE, 2020.

[10] VoltDB. 2022. https://www.voltdb.com/.

[11] Vertica. 2022. https://www.vertica.com/.

[12] MIT Supercloud. 2021. https://supercloud.mit.edu/.

[13] J. Kepner, R. Brightwell, A. Edelman, V. Gadepally, H. Jananthan, M. Jones, S. Madden, P. Michaleas, H. Okhravi, K. Pedretti, A. Reuther, T. Sterling, and M. Stonebraker, "TabulaROSA: Tabular operating system architecture for massively parallel heterogeneous compute engines."

[14] S. Breß, "The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS," in *Datenbank Spektrum 14, pages 199–209, 2014.*

[15] Tensorflow. 2022. https://www.tensorflow.org/.

[16] PyTorch. 2022. https://pytorch.org/.

[17] DataDog. DataDog's The State of Serverless, 2021. https://www.datadoghq.com/state-of-serverless/.

[18] Salvatore Sanfilippo. Retwis, 2021. https://github.com/antirez/retwis/.

[19] GCP. Google Cloud Microservices Demo (Online Boutique), 2022. https://github.com/GoogleCloudPlatform/microservices-demo/.

[20] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 3–18, New York, NY, USA.* ACM, 2019.

[21] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, "Aurora: a new model and architecture for data stream management," in *VLDB Journal 12, pages 120–139, 2003.*

[22] S. Harizopoulos, V. Shkapenyuk, A. Ailamaki, "QPipe: A simultaneously pipelined relational query engine," in *SIGMOD, pages 383–394, 2005.*

[23] S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, L. Perez, "The DataPath system: A data-centric analytic processing engine for large data warehouses," in *SIGMOD, pages 519–530, 2010.*

[24] Y. Tu, A. Kumar, D. Yu, R. Rui, and R. Wheeler, "Data management systems on GPUs: Promises and challenges," in *SSDBM '13, July 29 - 31 2013, Baltimore, MD, USA.* ACM, 2013.

[25] S. Breß, B. Köcher, M. Heimel, V. Markl, M. Saecker, and G. Saake, "Ocelot/hype: Optimized data processing on heterogeneous hardware," in *Proceedings of the VLDB Endowment 7, pages 1609-1612, 2014.*

[26] Y. Azar, I. Cohen, A. Fiat, and A. Roytman, "Packing small vectors," in *SODA (2016).*

[27] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and dependency-aware scheduling for data-parallel clusters," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), Savannah, GA, USA.*

[28] B. Sahah, H. Shahh, S. Sethh, G. Vijayaraghavanh, A. Murthyh, C. Curino, "Apache Tez: A unifying framework for modeling and building data processing applications," in *SIGMOD, pages 1357-1369, 2015.*

[29] T. Karnagel, D. Habich, B. Schlegel, W. Lehner, "Heterogeneity-aware operator placement in column-store DBMS," in *Datenbank Spektrum 14, pages 211–221, 2014.*

[30] A. Abdennebi, A. Elakaş, F. Taşyaran, E. Öztürk, K. Kaya, S. Yıldırım, "Machine learning-based load distribution and balancing in heterogeneous database management systems," in *Concurrency and Computation: Practice and Experience 34, 2022.*

[31] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, J. Soyke, "TensorFlow-Serving: Flexible, high-performance ML serving," in *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.*

[32] D. Crankshaw, X. Wang, G. Zhou, M. Franklin, J. Gonzalez and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17), Boston, MA, USA.*

[33] Sagemaker. 2022. https://aws.amazon.com/sagemaker/.

[34] Y. LeCun, C. Burges, "The MNIST database of handwritten digits." 2022.

[35] K. Kowsari, M. Heidarysafa, D. Brown, K. Meimandi, L. Barnes, "RMDL: Random multimodel deep learning for classification," in *Proceedings of the 2018 International Conference on Information System and Data Mining.*

[36] "Classify MNIST digits using convolutional neural networks." GitHub, 2022. https://github.com/j05t/mnist.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *IEEE Computer Vision and Pattern Recognition (CVPR), 2009.*

[38] "About WordNet." WordNet. Princeton University. 2010.

[39] TensorFlow Hub. 2022. https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4/.

[40] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 4510-4520, 2018.*

[41] T. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, P. Dollár, "Microsoft COCO: Common objects in context." 2014.

[42] Microsoft COCO. 2022. https://cocodataset.org/

[43] TensorFlow Hub. 2022. https://tfhub.dev/tensorflow/centernet/hourglass_512x512/1.

[44] X. Zhou, D. Wang, P. Krähenbühl, "Objects as points." 2019.

[45] A. Newell, K. Yang, J. Deng, "Stacked hourglass networks for human pose estimation." 2016.

[46] Google Cloud Platform. 2022. https://cloud.google.com/.