Containerized Execution of UDFs: An Experimental Evaluation

Karla Saur
Microsoft
karla.saur@microsoft.com

Tara Mirmira[*]
University of California, San Diego
tmirmira@eng.ucsd.edu

Konstantinos Karanasos[†]
Meta
kkaranasos@fb.com

Jesús Camacho-Rodríguez
Microsoft
jesusca@microsoft.com

ABSTRACT

User-defined functions (UDFs) have long been used as the de facto way to extend the capabilities of data management systems. However, they are restricted to the specificities of each DBMS, and recent demands for advanced analytics have increased the need for complex UDFs that may require execution of arbitrary computation written in any programming language, management of library dependencies, portability across environments and engines, and resource isolation. These requirements go beyond what traditional UDFs were designed for, and have given rise to containerized UDFs that enable encapsulation and portability. However, this approach is nascent and can result in significant performance penalties and usability issues. In this paper, we present the first study that spans all stages of containerized UDFs' life cycle, performance bottlenecks in their execution, and extensibility to support different engines.

Our experiments show that the performance of containerized UDF execution can be greatly affected by system design choices and that there are many trade-offs to consider. For example, regarding the method of communication with the containerized UDF, we show that binary-based implementations minimize overheads and are more than 2.4x faster than widely used text-based ones. Adopting a newer general-purpose communication method such as Arrow Flight can improve performance dramatically, causing a minimal ∼10% slowdown compared to non-containerized UDFs. Additionally, containerized UDF start times vary wildly due to program size and complexity, from .07s to 7s in our experiments. Our insights can help DBMS developers make appropriate choices based on individual use cases when designing their systems.

PVLDB Reference Format:
Karla Saur, Tara Mirmira, Konstantinos Karanasos, and Jesús Camacho-Rodríguez. Containerized Execution of UDFs: An Experimental Evaluation. PVLDB, 15(11): 3158 - 3171, 2022.
doi:10.14778/3551793.3551860

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/cage-paper-artifact/cage-vldb.

[*]Work performed while author was an intern at Microsoft.
[†]Work performed while author was at Microsoft.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551860

1 INTRODUCTION

Users have always sought ways to expand the capabilities of their database. Although the declarative nature of SQL has been the main reason for its success and widespread adoption, there is custom logic that is hard to express in SQL. Therefore, user-defined functions (UDFs) have become the preferred way to encapsulate complex functionality (often implemented in a programming language different from the engine query language) and increase the flexibility of these systems. This need for extensibility is more paramount today than ever before. Data is consolidated in data lakes in order to gain novel insights [28, 58]. There is even an effort to bring data together across industries (see data market platforms [35]). In this setting, modern data engines are asked to do much more than the traditional database [19, 47, 48, 67, 71]: train models, perform inference, drive business intelligence, visualization, etc.

The more broad the reach of data engines, the more users will rely on UDFs to cover an ever-broadening set of use cases. However, traditional UDFs cannot always cover all of these cases. UDFs can fail to provide users with the flexibility and dependency isolation needed for their workloads. Consider the common scenario of a user that wants to use Apache Spark [2] to perform data processing in Spark SQL and score their data using ML models called from UDFs written in Python [47]. Each of these UDFs may require varying dependencies based on the model they are using—the wrong dependency version can lead to runtime errors or incorrect results. However, the user may not be able to alter the engine host environment to install those dependencies, e.g., the query is run in a managed environment, it is not permitted by existing security policies, etc. Even if the user can load the necessary dependencies for their UDFs in the engine environment, several issues still arise: (*i*) UDFs within the same session may use conflicting dependencies, and (*ii*) changing the computation environment requires reinstalling those dependencies.

Over the last decade, **containers**, driven by the popularity of Docker [4], have become the de facto standard for running arbitrary code in an encapsulated environment. Containers are easily portable across computing environments and are lightweight, i.e., they share access to the host operative system, leading to fast startup times and efficient host resources utilization [25, 82]. These reasons make a compelling case for leveraging containers to execute UDFs [22, 78]. In particular, users can create an image packaging their UDF code and dependencies so it can run consistently within a container in different environments. Containerized UDFs can help them overcome the aforementioned challenges, providing support in the following areas:

- **Fine-grained dependency management.** First, containers facilitate that user code can rely on *arbitrary versions* of libraries to compute results. For instance, in the machine learning context, a Python UDF may need to rely on specific versions of packages available in public or private repositories.
- **Programming language flexibility.** Containers allow users to write their function code in any language (e.g., Python, C#, Java, Go). This can be a huge catalyst for productivity, since users can focus on their function logic, relying on the libraries that they consider more convenient to work with, rather than learning a new language or reimplementing custom code each time.
- **Cross-platform portability.** Containers let users save time by writing their UDF code only once and reusing it across engines, as long as the engine can provide a native UDF implementing the communication protocol with the container. Thus, UDFs can be executed easily in engines of completely different characteristics such as SQL Server [10] or Spark.

In a nutshell, combining containers and UDFs can bring multifold advantages for end users, as we further discuss in §2. However, there are many options to consider when implementing containerized execution in existing systems, as we discuss next.

Container image building and distribution. In order to efficiently manage and operate a containerized UDF system for a database engine, containers need to be built and distributed efficiently. We look at aspects of the container life cycle that may apply to containerized UDFs and share design trade-offs and best practices. For example, how can we rapidly build containers when multiple UDFs share similar dependencies? In the case where containers will be co-located with the query execution tasks on the same nodes, what are potential bottlenecks in rapidly distributing the container images to all nodes in the database engine cluster?

Execution. In the case of co-located containers, which system designers would be responsible for setting up, deploying, and operating, what are the start-up costs of different types of UDF containers, and when and how many containers should we start? If, instead, system designers leverage a serverless compute platform, such as Azure Functions [59], to run the containerized UDF, how much performance will we trade for that convenience?

Data exchange. Data needs to be transferred between the compute engine and containerized UDF. One could choose different implementation options for data movement, e.g., sockets, REST, RPC. How does each alternative perform compared to each other, running locally (with containers co-located with engine nodes) vs remotely? In addition, we need to choose the data exchange format. What are the trade-offs of a custom protocol compared to relying on a generic data format such as JSON or Apache Arrow [1]?

Contributions. With the sheer number of choices available, the goal of this paper is to highlight the performance implications of various design alternatives for supporting containerized UDFs in database engines, starting from container creation and all the way through query evaluation. We make the following contributions:

- An analysis and evaluation of containerized UDF execution (§4), given a representative architecture using Spark (§3).
- Performance impact of multiple design choices in a local, self-managed setup (§5), such as data communication implementation or container initialization.
- Trade-offs between local (co-located, self-managed) and remote (external, hosted) container management and deployment implementations, including a thorough performance comparison (§6).
- Containerized UDFs using SQL Server (§7), an engine with different characteristics compared to Spark, hereby showcasing the flexibility of this approach.

2 MOTIVATING EXAMPLE

In this section, we provide a short overview of a common approach to UDF implementation in existing data engines (§2.1), following which, we discuss the limitations of such an approach (§2.2).

2.1 UDF Definition and Execution

Although every data processing system proposes a different flavor of UDFs, there are commonalities in the steps that a user should follow to execute their custom code across them. Apache Spark [2] is an open-source distributed data processing engine. Next, we use Apache Spark to illustrate the aforementioned steps and some of the challenges that can arise in the management of UDFs within Spark and similar systems.

Assume a user wants to perform *batch inference* (also known as *scoring*) on data stored in Spark using a pre-trained scikit-learn [17] model.

Implementation. In this example, the user will start by writing a Python scalar UDF for Spark, which could be done as follows:

```
@udf(returnType=FloatType())
def udf_skl_predict(model_path, data):
    loaded_model = pkl.load(open(model_path, 'rb'))
    return loaded_model.predict(data)
```

The UDF will operate on single input arguments *model_path* and *data*, load the model from the provided path, feed the data into the model, and return a floating number representing the prediction. Similar to other engines, Spark provides auxiliary libraries with pre-defined *annotations* that can be used to decorate the function implementation, e.g., in this particular case, the UDF return type.

Registration. After implementing the UDF, the user needs to expose the new UDF (including its dependencies) to the engine so it can be referenced at query time. In this case, the registration is done through PySpark (native Python interface for Spark) and consists of a single method call providing the user-facing function name and a reference to the UDF.

```
spark.udf.register("PREDICT", udf_skl_predict)
```

Other data that can be provided at registration time varies greatly across engines, e.g., behavior characteristics, security mode, etc. After executing this method, the UDF will be callable within the scope of the current Spark session. In addition to registering the function with the engine, the user also needs to ensure that all Spark nodes have the correct environment to execute the UDF. Concretely, PySpark provides different ways to manage Python dependencies within a Spark session [50], either using Conda, Virtualenv, or PEX.

Execution. Finally, users run queries referencing the newly created UDF as they would normally do with any other function.

```
spark.sql("SELECT PREDICT('<file path>', col_x) AS p "
  + "FROM table").show
```

The internal execution mechanism varies greatly from engine to engine. Spark is written in Scala, and thus, only UDFs written in languages supported by the Java Virtual Machine (JVM) can be run natively within the engine. In addition, Spark provides support for UDFs written in Python[1], such as in this example, and the Spark community has recently introduced important improvements to make Python UDF execution more efficient, e.g., vectorized UDFs built on top of Apache Arrow [1], reducing the invocation cost and data serialization overhead between JVM and Python processes.

2.2 Limitations of Existing Approach

As noted in the Introduction, there are several limitations with the aforementioned approach to UDF implementation.

L1: *Fine-grained dependency management.* The dependencies are defined within the scope of a Spark session [33, 65]. Consider a common scenario where the user calls their UDF again within the same query, or in another query within the same session, but this time using a model that was built using a different scikit-learn version, and thus, requiring a different Python environment to avoid an error [18, 20]. The current approach cannot handle dependencies at this fine granularity.

L2: *Programming language flexibility.* The example above considers an engine with built-in support for Python, the programming language used by the UDF. Now assume that the user wants to execute a function written in C#. Without native support for C# in Spark, the user will need to reimplement the function in a supported language.

L3: *Cross-platform portability.* The UDF implementation and handling of dependencies are tightly coupled with the Spark engine. Over time, the user may want to use the same UDF simultaneously in another engine, or the user's organization may decide to migrate to a new engine altogether. Even if the target engine has support for Python, these two scenarios would likely require (*i*) rewriting the UDF, and (*ii*) setting up all the dependencies in the new execution environment correctly, as the current approach is not portable across engines and environments.

3 CONTAINERIZED UDF EXECUTION

In the previous section, we discussed the general process for creating and calling a UDF. In contrast, we introduce containerized UDFs in this section. In particular, we briefly describe containers in §3.1, then present a reference architecture and describe the modified steps to define and execute a containerized UDF in §3.2.

3.1 Background on Containers

Containers are a lightweight way to package code, runtime, and all dependencies into a single runnable unit. Containers are different from VMs in that they virtualize resources at the operating system (OS) level [25, 82].

Containers are stored as *container images*, which consist of a layered file system. The layers in the container image are read-only and represent different components that are added to the container at creation time. The layers are stored in compressed (zipped) format, and each container image also includes a *manifest file* with a list of hashes, one for each layer. Often, a container image starts with a base image in the first layer such as `Ubuntu` or `Alpine` (a common lightweight Linux-based OS), and then the user may include additional layers consisting of packages or software on top of that layer. The layers of the container image are stored in the order of the commands in the *Dockerfile*, a script that describes how to build the image. To launch a container, the image is deployed into a running *container instance*, which will include a thin writable layer to store any state created at runtime.

Container images are usually handled by a *container registry*, a repository for storing and retrieving the different layers of each image. After the images are built, a client can be used to *push* them to the registry. Each layer of an image is compressed and pushed separately, as described in the manifest files. The registry uses the manifest files to track which layers are common among container images. For example, assume a user creates two images with `Alpine` as the base, and then adds different software to each image in subsequent layers. When the user pushes the first image to the registry, all layers will be sent and stored. However, when the user pushes the second image, the registry will identify the `Alpine` layer, and thus, only the subsequent layers will be sent.

To retrieve the container image, the client can *pull* it from the registry layer-by-layer. If any of the image layers are already present on the client's machine, that layer will be skipped. As the layers are pulled, they are also extracted from the zip format so that they can be launched. Fully managed container registry offerings have proliferated over the last few years [5, 9], especially across cloud providers [37, 56, 79].

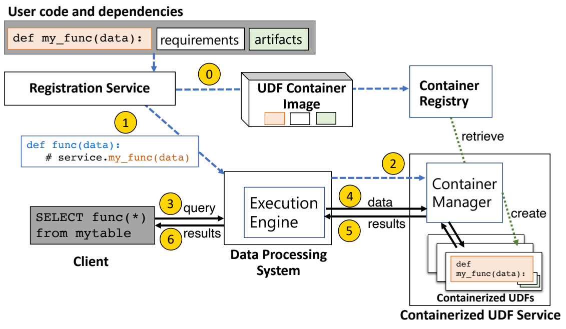

Figure 1: End-to-end workflow. Setup steps shown in blue dashed lines, execution shown in solid black lines.

3.2 Containerized UDF Definition and Execution

Other systems [22, 78] have forayed into the space of containerized UDFs, enabling support for external functions backed by cloud serverless platforms. The design space for utilizing containerized UDFs for generic function execution in data engines covers a wide range of topics. We seek to characterize the design decision landscape and evaluate a variety of possible solutions to aid in decision-making. To this end, we highlight the important components that would need to go into a such a system. In this section, we introduce a reference architecture, based on similar systems, that implements

[1]PySpark is popular among data scientists, who prefer to rely on the rich ecosystem of data science packages available in Python.

containerized UDFs within a data processing system. The architecture and the interaction among its components are presented in Figure 1. Next we describe the steps to define and execute containerized UDFs in contrast with the example introduced in §2.1.

Implementation. Just as in the prior example, the user must bring in their UDF code, such as a Python function for scoring an ML model. Additionally the user must bring in a list of dependencies (such as a Python requirements.txt file [15] and/or a list of Ubuntu packages), and any additional files that the UDF environment might need (such as an ML model). Figure 1 shows these dependencies at the top of the workflow graph in the dark gray box.

Registration. The user submits the UDF code and dependencies to the *Registration Service*. Continuing in Figure 1, the *Registration Service* packages them into a UDF container image ⓪, and pushes it to a container registry. The containerized UDF will include the original UDF with boilerplate code for (*i*) decoding incoming data requests, (*ii*) calling the original UDF logic, and (*iii*) returning the encoded result. The *Registration Service* also generates a user-facing client-side UDF ① (that the client will use to call indirectly the *Containerized UDFs*) and registers it with the *Data Processing System*.

This setup also employs a *Containerized UDF Service*, which can access the container registry and is accessible from the *Data Processing System*. This service, which can be co-located with the engine or running remotely as we will study in §6.2, contains the *Container Manager*, which is responsible for orchestrating container deployment. The *Data Processing System* contacts the *Container Manager* ② which will coordinate distribution of the image to all worker nodes. Depending on its settings, the *Container Manager* may spin up the containerized UDFs (green dotted arrows) at this time to optimize performance, or it may wait until the user issues a query to save resources; we evaluate both options in §5.3.2.

Execution. In the lower-left corner of Figure 1, we show the client submitting their original query ③ referencing the client-side UDF. The query is compiled and submitted to the *Execution Engine*. During execution, the client-side UDF sends the data in batches to a URL provided by the *Container Manager* ④, which acts as a load balancer to distribute these batches across the containerized UDFs. If the containers were not started as part of the Registration phase, they will be started now. The communication between the client-side UDF and the *Containerized UDF Service* is governed by a communication protocol. As we will explore in more detail in §5.1, different data formats and communication implementations can be supported. The *Containerized UDFs* process the data and return the results, which are then sent back to the client-side UDF inside the *Execution Engine* ⑤. Query execution proceeds as usual until it completes and its results are returned back to the client ⑥.

As we can see from this example, containerized UDFs solve the limitations that we described in §2.2. Concretely, UDF dependencies are completely isolated from other UDFs and the *Data Processing System*, and thus, can be handled at a finer granularity (**L1**). In addition, users are not forced to implement their UDF in any specific programming language depending on the *Data Processing System* (**L2**), as the original UDF is not directly called by it. And finally, the *Data Processing System* can be swapped out for any other system as long as it can provide a client-side UDF that can communicate with

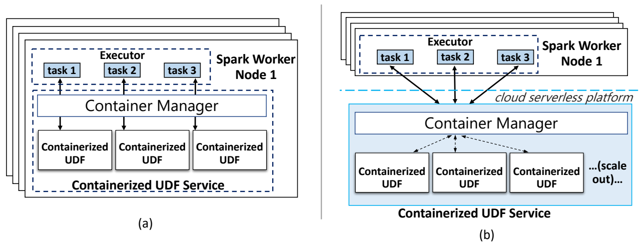

Figure 2: *(a):* Local, self-managed containerized UDF service. *(b):* Remote, managed containerized UDF service.

the *Containerized UDF Service*, which prevents the user from being locked to using their UDF with only a specific data engine (**L3**).

In the experiments presented in §4-§6, we evaluate this architecture using Spark as the *Data Processing System*. However, as we mentioned previously, our study aims to provide results and insights that are applicable across engines that can rely on containerized UDFs. In Figure 1, the user-facing client-UDF generated by the *Registration Service* and the code to register it with the *Data Processing System* are specific to Spark. However, the rest of the implementation would still be similar for other engines supporting UDFs. We discuss and validate the generality of the reference architecture in §7, where we replaced Spark with SQL Server, and we only needed to change the user-facing client-UDF, and the code to execute the registration with the engine; the rest of the implementation remained the same.

4 END-TO-END EVALUATION

In this section, we look at the end-to-end performance of our reference architecture. We focus on some high-level takeaways before digging into sub-component performance and providing more details in subsequent sections. To understand how the characteristics of the UDF affect the implementation choices, we used random data on UDFs with different characteristics:

- **Data-intensive UDF.** It takes a pre-trained scikit-learn [73] random forest model with 28 inputs, scores and returns the data.
- **Compute-intensive UDF.** It takes an integer as input and returns the largest prime factor of that integer.

Experimental setup. Our experiments were performed on a Spark 3.0 cluster with 2 head and 4 worker nodes (provisioned by HDInsight [8]), each with Intel®Xeon®Platinum 8171M CPU @ 2.60GHz (8 Cores), 64GB RAM, and Python 3.7. Each experiment was run 7 times, dropping the low/high values and then averaging. We assume that the containers have already been built and deployed.

End-to-end results. Figure 3 shows end-to-end runtime for both UDFs using three different implementations across three different data set sizes (100K, 1M, and 10M rows), which are scaled-up replications to show how the performance scales across the range of sizes. For our baseline implementation (shown as solid lines), we executed both UDFs as regular Python scalar UDFs and we did not use containers, as described in Section 2.1.

Using the reference architecture presented in §3.2, we used two representative implementations for locally and remotely managed containerized UDF services. For the first implementation (*Local*), the containerized UDF service was co-located with each Spark executor

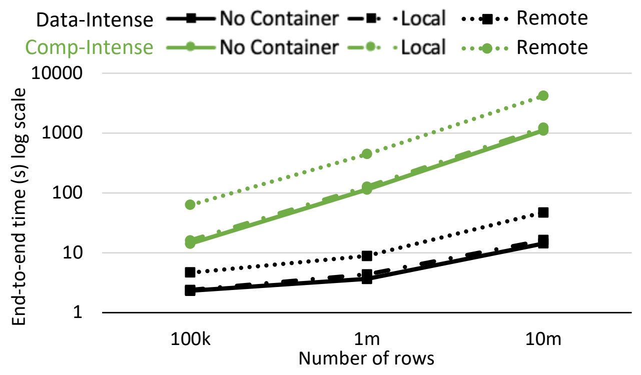

Figure 3: End-to-end times for data and compute intensive UDFs for no container, local (co-located self-managed containers with Arrow Flight), and remote (JSON-based communication to a remote externally-managed endpoint).

on the same VM as shown in Figure 2a. For each Spark task, we deployed a container locally on the same VM. For this experiment we used Arrow Flight [54], a mostly zero-copy gRPC-based protocol, to send the data and responses over `localhost` between Spark and the containers. The performance (shown in dashed lines) is very close to the no-container case, ~10% slower across both UDFs and all data sizes. The overhead is minimal because (*i*) the network latency is low over localhost, and (*ii*) Arrow Flight enables efficient data transfer through zero-copy [26].

For the second implementation (*Remote*), we used Azure Functions (AZF) [59] to act as the remote externally-managed containerized UDF service, shown in Figure 2b. The data is sent from Spark to AZF as numpy [12] serialized using pickle [14] embedded in a REST/JSON protocol. (Arrow Flight was not available on any of the remote services we considered.) The performance (shown in dotted lines in Figure 3) is significantly slower than the no-container version: a ~2.0-3.2x slowdown for the data-intensive UDF and a ~3.8-4.4x slowdown for the compute-intensive UDF due to (*i*) the overhead of serialization/deserialization to/from JSON, (*ii*) the lack of state between subsequent calls that would allow model caching in scoring for the data-intensive UDF, (*iii*) network latency due to the remote endpoint, and (*iv*) limitations [60] we encountered in scaling up the remote service that made it difficult to match Spark.

Although the performance of the remote managed setup is the poorest, there are several advantages to it. In particular, the co-located setup requires operating and maintaining infrastructure. Additionally, it needs sufficient local compute and memory to host the containers in the executor nodes. On the other hand, a managed service may be able to quickly scale the number of containers up and down more easily, and may provide additional isolation guarantees.

§4 Key Takeaways:

- With local (co-located) self-managed containers, we saw only ~10% overhead with our Arrow Flight-based implementation.
- With remote (external) managed containers, we saw a significant slowdown (~2-4.4x) compared to the no-container version.

Experimental Roadmap We next break down the process and dive into performance factors related to local self-managed containerized UDFs in §5. We then look at trade-offs when using a remote managed system for containerized UDFs in §6. Finally, we discuss the extensibility of this design in §7.

Table 1: Containerized UDFs communication alternatives.

Method	Selected Implementation	Overhead	Generality
File	***Parquet*** to shared disk	Medium	High ✓
Binary	***Sockets***: serialized numpy over TCP	Low ✓	Low ✗
Binary	***Arrow Flight*** over gRPC	Low ✓	High ✓
Text	***Web Server*** Endpoint	High ✗	High ✓
Text	***MLflow*** over HTTP	High ✗	High ✓

5 SELF-MANAGED CONTAINERIZED UDFS

In this section, we dive deeper on running containerized UDFs when they are self-managed, meaning when system designers implement containers management themselves as opposed to relying on an externally managed service to host the containerized UDFs. First, we describe different communication alternatives between the engine and the containers in §5.1. The choice of data communication implementation has an impact on performance. In §5.2, we discuss this impact in the context of the end-to-end performance of containerized UDFs, while in §5.3, we explore the impact on initialization of containerized UDFs as well as how this initialization can be affected by UDF requirements and system architecture. Finally in §5.4, we examine additional factors impacting performance throughout the containerized UDF life cycle.

5.1 Data Communication

As indicated in Figure 1, the data engine needs to send data to the containerized UDF service endpoint and the endpoint needs to execute the UDF over this data and send the result back. It is important that the method of transport adds as little overhead as possible while remaining as language and platform agnostic as possible. In this section we experiment with *how* the data is communicated between the database engine and the container, and the impact of different communication implementations on performance.

In §3.2, we explained that we need a client-side UDF to communicate with the containers. Since we are using Spark, we use a Pandas UDF [13]. This client-side Pandas UDF handles the communication of data from the database engine to the container and back. The client-side UDF is registered in the Spark environment and, from the user's perspective, gets called as if it were the user's UDF. Unless otherwise specified, our experiments call the client-side UDF in batches of size 10K (the default batch size for a Pandas UDF); one batch corresponds to one call to the container.

Throughout the experiments in this section, we use the same two example UDFs described in §4. To communicate data between the database engine and the container, there are several options which we summarize in Table 1 and describe next:

File-based: Passing the data by reading and writing from shared files. This requires setting up a file share (which avoids network communications if the database engine and container are on the same machine) or a network file share between the database engine

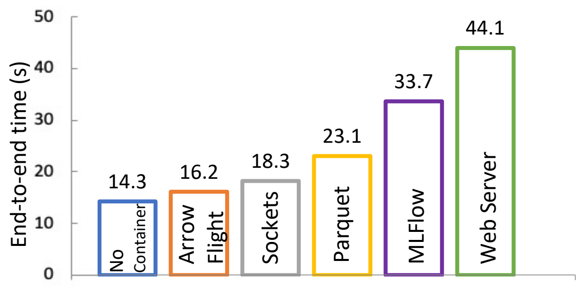

Figure 4: End-to-end times for scoring 10M rows with different transport layers with local containers.

and the container. It is also relatively simple to implement and troubleshoot. The performance of this method can suffer depending on how the data is stored and how fast the storage can be accessed (disk speed or network speed using a remote disk).

Selected implementation(s). We mounted a volume into the container on the same VM as the database engine to use as the shared reading and writing location and used Apache *Parquet* as the serialization format.

Binary-based: Network-based communication with a customized serial/binary format. This method minimizes the amount of data that is sent and and the cost of serialization is relatively low depending on format (pickle [14], Apache Thrift [3], etc).

Selected implementation(s). For this method, we experimented with two different implementations. The first is *Sockets*, which is serialized numpy [12] over TCP. The sockets approach may limit the language and library versions in which the user can write a UDF (depending on serialization implementation). For example, the usage of serialized numpy for data communication necessitates that the numpy version in the database engine and in the container match. However, we still chose to experiment with this option to assess the overhead introduced by other implementations. In particular, as serialization is amongst the fastest way to transport data (as opposed to verbose JSON), it represents a practical lower bound for data transportation. The limitations of the sockets approach can be overcome with the second implementation we evaluated, which is Apache *Arrow Flight* [54] over gRPC.

Text-based: Network-based communication in a standard verbose format such as JSON. The advantage of using a common format such as JSON-based methods is that it can be read in any platform that can parse JSON, and many REST endpoints support or expect this format, making it an extremely flexible solution. A disadvantage is that it can negatively impact performance due to the overhead of JSON parsing and the larger quantity of data in general that must be sent over the network.

Selected implementation(s). For this method, we experimented with two different implementations. We used a REST/JSON-based *MLflow* endpoint [11], and a REST/JSON-based *Web Server* implementation provided by Azure Functions (AZF) [59].

5.2 End-to-end Evaluation Time

In this experiment, we used the same Spark cluster and setup introduced in §4 to measure end-to-end performance across the data communication options described in Table 1. All experiments sent the data in 10K batches (which is both the default size and what we confirmed to be optimal for Spark-side performance) except for *Parquet*. With *Parquet*, we found 100K batches minimized the overhead of creating and opening files. For the experiment, the containerized UDF service is co-located with each Spark executor on the same VM. Note that although containers using text-based communication such as *MLflow* and *Web Server* are designed to be used in a remote managed environment, we also chose to run them in this local setting for completeness. This way we can isolate the parsing and protocol-related overhead in a local setting (where network latency was not a major factor). It also allowed us to control the exact number of deployed containers and keep the underlying hardware consistent throughout the experiments.

Figure 4 shows the end-to-end time for executing the data-intensive scoring UDF introduced in §4 on 10M rows. In the column labelled *No Container*, we show the UDF written as a standard Pandas UDF, and we use this as the baseline. The next two fastest performers are the binary-based methods *Arrow Flight* and *Sockets*, with a ~1.1 and 1.3x slowdown respectively. Following these is the file-based method *Parquet* with a ~1.6x slowdown. The slowest methods were the text-based methods *MLflow* and the *Web Server*, which had a ~2.4x and ~3x slowdown respectively from no container. The *Web Server* container performs worse than *MLflow* because MLflow has built-in optimizations such as model caching [69].

We repeated the experiment for 100K, 1M, and 100M rows, and the six options always finished in the same order with the scalability remaining roughly constant as Figure 4, e.g., for *MLflow*, 33.7s for 10M rows and 338s for 100M rows; for *Parquet*, 23.1s for 10M rows and 214s for 100M rows. In general, we found that Arrow Flight is the clear winner in terms of both flexibility (with support for 11 languages and counting) and performance. To experiment with larger data sizes, we executed a query that performed scoring using a scikit-learn model over 725.5GB of the Criteo click log dataset [31]. The execution for *No Container* took 2903s, while the *Arrow Flight* variant took 3028s. Thus, the overhead was slightly less than the experiment shown in Figure 4: ~4.3% vs ~10%. This difference is due to the smaller size of the Arrow Flight payload in the Criteo example, i.e., the selected features contain many repeated values, which in turn lead to better compression. The Arrow Flight setup could potentially be improved further by mapping the data directly from the database engine into the container [83], but this setup may not work easily with all languages, setups, or database engines.

Impact of UDF profile. In §4, we introduced both the data-intensive scoring UDF and the compute-intensive primes UDF. To understand how the characteristics of the UDF affect the overhead ratio, we will use those UDFs in addition to a *compute/data-intensive* UDF, which is a modified version of the compute-intensive UDF where we send 28 inputs, like the data-intensive UDF, and return the largest prime factor of one of the inputs.

We compared those UDFs using two different implementations: (*i*) without using a container (baseline), and (*ii*) using a container based on the AZF *Web Server* image. The results of this experiment are shown in Figure 5. To normalize across the runtimes of the different UDFs, the comparison between the baseline and *Web Server* is represented as a ratio of runtimes. The relative overhead of encoding and transferring data to the container and receiving

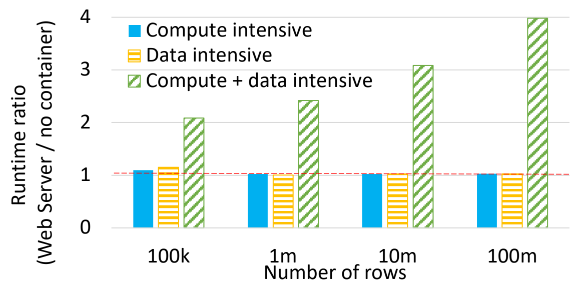

Figure 5: Comparison of local REST (self-managed) with no container for different types of UDFs.

and decoding data from the container is less for compute-intensive UDFs as compared to non-compute-intensive UDFs. For compute-intensive UDFs like the primes UDF, the majority of the runtime goes in performing the computations of the UDF itself. For computationally lightweight UDFs, such as the data-intensive scoring UDF, the runtime of the original UDF is so fast that any additional time needed for parsing and network latency significantly adds to the relative runtime. These results suggest that the required compute and data profile of the UDF will impact the selection of the data communication method. For a single containerized UDF, the challenge of optimizing a computationally lightweight function that is also data-intensive, is that the cost of sending data to and from the container must be paid without outweighing the cost of performing the function itself. As a result, optimizing the performance of lightweight UDFs requires using a data transport technique that has low absolute cost, such as Arrow Flight. More compute-intensive UDFs have more flexibility, as any data transport method might be acceptable due to small relative overhead. Although a binary communication method may offer the absolute optimal solution in terms of performance, there are benefits, discussed in §5.1, of other data communication methods. *Thus, the profile of the UDF and consequently, the relative overhead of data transport to and from the container, play a role in choosing a data communication method.*

Impact of query profile. Previously we discussed experiments involving `SELECT-FROM` queries. To gain insight into the effect the query profile has on the runtime and overhead of containerized UDFs, we experimented with (1) a `GROUP BY` query with an aggregate UDF and (2) a `JOIN` query with scalar UDFs on its inputs.

For (1), we experimented with an aggregate query using the NYC Taxi dataset [66]. The aggregate groups by start longitude (10K-90K rows/group), leading to a shuffle read of 65.6GB, and returns a serialized scikit-learn model trained on that group. Computation for the `GROUP BY` in conjunction with the compute-intensive model training UDF results in an overall query workload that is more compute-intensive than any of our other experiments and causes CPU bottlenecks. Our baseline (no container) setup had an end-to-end runtime of 910s (CPU utilization was 80-95%) and local Arrow Flight took 1829s (CPU utilization over 100%). Compared to Figure 3, the overhead for Arrow Flight increased from ~1.1x to ~2x due to the CPU bottleneck. We ran the same experiment with the local *Web Server*, and were also CPU-bound, with overhead of ~8x. To obtain more compute, we reran *Web Server* remotely on AZF and

Table 2: Memory footprints and initialization times.

	Size on Disk	RSS (KB)	docker run (s)	UDF Init (s)
Parquet	490MB	67,248	0.58	0.49
Sockets	352MB	47,489	0.59	0.07
Arrow Flight	516MB	70,832	0.61	0.10
Web Server	1.58GB	121,831	0.61	3.07
MLflow	1.60GB	201,994	0.62	7.02

saw only ~5x slowdown, which was still worse than the ~2-4.4x slowdown shown for remote in Figure 3.

For (2), we used a self-`JOIN` query that executes the data-intensive UDF on each of its inputs. We locally pre-spun up twice the number of containers than we used in the previous examples, repeated the data-intensive scoring experiment with Arrow Flight, and observed that the overhead was ~10%, as we expected. Note however that in a resource-constrained cluster, we might not be able to spin up containers for all UDFs ahead of time (warm start, further discussed in §5.3.2), leading to additional overhead. If this were an issue, one could consider leveraging a remote containerized UDF service. The optimization of queries with multiple containerized UDFs is discussed in §8 and §9 but left to future work.

From these results, we can observe how more complex queries and operators may lead to increased utilization of cluster resources, and thus, fewer resources available for running the containerized UDFs. For instance, in (1), the resource competition between the shuffle read from the aggregate and the compute-intensive UDF resulted in CPU bottlenecking that created additional overhead. On the other hand, in (2), both the query and the UDF are less compute-intensive, hence we could run twice the number of containers and see the same performance overhead as with a single containerized UDF, despite having multiple containerized UDFs in the query.

5.3 Initialization and Memory Footprint

In addition to any resources needed to facilitate database engine to container communication, UDFs can have arbitrarily complex requirements such as mounting a file share or launching a complex program. For that reason, it is critical to measure the impact of these requirements on the time taken by the container to be ready to serve requests from the engine. We refer to this time as the initialization time. Additionally, the questions of when and how many containers to initialize give rise to performance trade-offs that should be considered, which we study here.

5.3.1 Program resources and complexity. The five images introduced in §5.1 (described in Table 1) have different requirements and implementation complexity. For each of those images, Table 2 reports the image size on disk and the resident set size (RSS) as reported in the `memory.stat` file. (RSS is the memory footprint consisting of stacks, heaps, and memory maps.) This information can be used as a very rough measure of program complexity.

There are two things that must happen before requests can be served by the UDF program. First, the container must be in *Running* state, initiated by calling `docker run`, which means that resources are allocated for the container (including port forwarding/volume

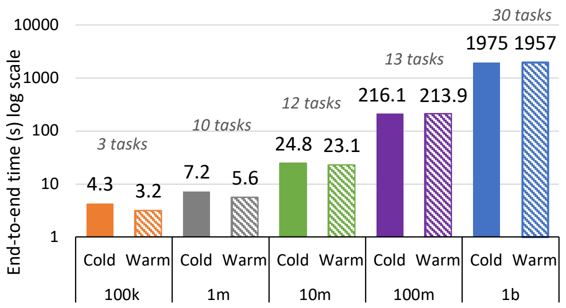

Figure 6: Log-scale end-to-end times for the *Parquet* file share UDF container with both cold and warm starts.

mounting inside the container). For each of the 5 images, we measured the time to return from `docker run` and transition to *Running* state on our Spark cluster setup. All 5 images took 0.6s ±0.02*s*.

Second, after a container is running, the UDF program itself needs to be ready. To measure this, in the *Parquet* case, which uses file share, the program needs to launch so that it can read and write data from the mounted volume. Thus, we modified the container code to write a health check file to the shared volume when the program is ready, and we timed when the file was created. In the network-based cases (all others), the program needs to launch and open a server socket before it is ready. Thus, we took an approximate measure using `curl` on the socket over `localhost` and waited for it to send a `reset` message. (If the program inside the container was not running, we received a connection refused message.) Table 2 shows the results of this in the *UDF Init* column.

The time for the program to be ready in the network-based containers varies greatly across our examples, with the simple TCP *Sockets* taking only 0.07s while the *MLflow* endpoint (which launches multiple threads) took more than 7s, 100x slower. *Program complexity plays a large role in start times; this significant variation should be taken into account for the design of containerized UDFs.*

5.3.2 Prewarmed containers. It is well documented (ex: [53]) that container cold start (i.e., container is not spun up until it is needed) has a significant penalty over warm starts (i.e., container is spun up preemptively). The trade-off that occurs here is that cold start minimizes the amount of time that a container is running, which saves resources, but by not starting the container soon enough, it incurs a performance penalty that is especially noticeable in short running queries. Hence, it is important to quantify the impact that prewarming a container has on the containerized UDF scenario.

For this experiment, we run the ML model scoring UDF introduced in §4 inside the *Parquet* image which uses file share. We chose the *Parquet* image because it had median initialization time from our Table 2 experimental results. The cluster, described in §4, has 4 nodes and a maximum of 3 concurrent tasks per node. For the warm start case, we pre-spin 12 containers (one per task) before running our query. For the cold start case, each Spark task launches its own container before processing its input data.

Figure 6 shows the time to run the *Parquet* containerized UDF within a query using different input sizes: 100K–1B rows of data. The total number of tasks that Spark executes for a given query is computed based on the size of the input data to be processed. For our data sizes, Spark uses from 3 to 30 tasks, shown above the bars. Thus, for the warm start, there are idle containers for the smaller data sizes, while for 100M, one task cannot be executed until one of the first 12 scheduled tasks has finished. For the cold start, the number of containers that are spun up is equal to the number of tasks run by Spark. Our results show that the difference between warm and cold start for each of the data sizes is 1.1, 1.6, 1.7, 2.2, and 18s, respectively. This roughly fits the expected slowdown for the cold start based on the results of our previous experiment shown in Table 2. For instance, the 100K row example requires 3 Spark tasks, and for cold start this means there will be ~1 container per node (because the 3 tasks that each spin up a container are distributed across the 4 node cluster for a total of 12). Adding the time to run the container (.58s) and the time for the program to be ready (.49s) amounts to ~1.1s, the approximate slowdown we obtained. For 10M rows, there is likely some contention since each of the 4 nodes spins up 3 containers at once. As Spark does not return a result until all tasks are finished, the performance is bound by the slowest container's start time on the high end of the variance bounds. For 100M rows, the startup cost almost doubles since it cannot spin up the 13th container until the fastest of the 12 tasks has finished. Finally, for the 1B container with 30 tasks, 2.5 waves of tasks result in additional slowdown, but the ratio of warm:cold slowdown remains on par (99%) with 100M.

For the *MLflow* container, the amount of overhead (~7s) would make smaller queries with a cold start prohibitively expensive. For larger runs (for example, in Figure 3 we see that *MLflow* took 33.7s for 10M rows), adding a ~7s+.6s startup penalty starts to become amortized, but would still be non-trivial, adding an additional ~1.2x slowdown in the 10M row case in addition to the slowdown that comes from the communication overhead. The faster container start times for *Parquet* means that cold start adds only a ~7% overhead for 10M rows. *Deciding on whether to pre-spin containers therefore depends on use case, data size, and resource utilization requirements.*

5.3.3 Number of Containers. In the previous experiment, we chose to launch one container per Spark task for the warm start scenario. As we can see in Table 2, the amount of memory (shown as RSS size) needed for running each container is reasonably small. However, we wanted to evaluate the performance impact of reducing the number of containers for use in memory-constrained environments. For this experiment, we spun up only one container per node, i.e., the container could have been shared by at most 3 tasks. In general, we found a prohibitively large performance degradation when Spark tasks were sharing the container. The main reason is that unless the container code is adapted to handle multiple requests concurrently, e.g., using multiple threads, the degree of parallelism is effectively reduced from the number of tasks to the number of nodes.

5.4 Containerized UDF Life Cycle Performance Considerations

In the previous section, we introduced a variety of container images. It is considered a best-practice to keep images as small as possible [34], but as we saw in Table 2 and as described in other studies [88], commonly used images can vary widely in size and complexity. We performed studies to understand the factors impacting performance when building and distributing containerized

```
 1 FROM python:3.8-slim-buster as build
 2 WORKDIR /install/

 3 COPY /local/reqs.txt reqs.txt  # Install the Python packages
 4 RUN pip install --prefix=/install -r reqs.txt

 5 FROM python:3.8--slim-buster # Now the next phase of the build
 6 COPY --from=build /install /usr/local  # Copy the Python packages from step 4

 7 WORKDIR /workdir/

 8 COPY main.py /workdir

 9 COPY $MODELNAME /workdir/model   # Do this as late as possible

10 CMD [ "python3", "-u", "/workdir/main.py" ] # execute the command
```

Figure 7: Dockerfile example for scoring.

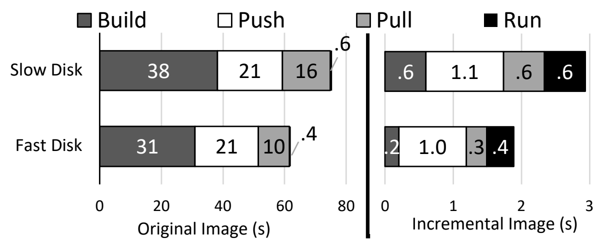

Figure 8: Container creation and distribution with slow and fast disks. Left graph: original image. Right graph: incremental image. ***(Notice x-axis scale.)***

UDFs. In this section, we study the steps leading up to UDF execution related to the container building and distribution. We first look at building additional images when there are only minor changes to the UDF, allowing the user to rapidly generate additional containerized UDFs if their dependencies do not change significantly. We then look at the impact of disk speed on the container life cycle.

Experimental setup. All experiments in this section were performed on a VM with 4 vcpus, 16 GB RAM (Azure Standard D4s_v3 SKU) using Docker v20.10.7 with Ubuntu 18.04. The VM had a 1 Gbps network connection. Each experiment was run 5 times, dropping the low/high values then averaging. We used the premium offering of Azure Container Registry (ACR) [57], which has a minimum of 100 Mbps download and 50 Mbps upload speeds as of this writing, located in the same cloud region as our other resources.

5.4.1 Container Image Build. As we described in §3.1, the *Dockerfile* contains a list of commands specifying how to build the container image. This file also determines the resulting image size and structure, and thus, it can affect the performance of container life cycle operations. *Dockerfiles* can be divided into multiple stages that are built sequentially. Each stage starts with a `FROM` statement referencing a base image, followed by statements that add layers to that image. Figure 7 shows a *Dockerfile* creating an image that scores the ML model (`$MODELNAME`), which we used with minor modifications for our *Parquet*, *Sockets*, and *Arrow Flight* containers. We start with a minimal 114MB Python base image [6] on line 1. Lines 2-4 download the necessary python packages. Line 5 starts a new stage with a fresh base image, and line 6 copies the packages into that image. The "build" container from line 1 is discarded, and so are the unnecessary build artifacts that were part of the `pip install` process, i.e., `wheels`, `setuptools`, `pycache`. The resulting image is 463MB on disk. (If we did not discard the build artifacts by using the multi-stage build, this image was 565MB on disk, meaning the image is 102MB smaller due to leftovers from installation.)

Container clients such as Docker have built-in support for caching image layers so that they do not have to be rebuilt, referred to here as "incremental" images. In containerized model scoring, it is likely that many of the dependencies (and thus, layers of the container) remain consistent across UDFs. On line 8 we copy the main file into the container, which contains the UDF code and the communication code (such as Arrow Flight code), and on line 9, we copy the model as the last step before the execution command. Doing these steps last allows the user to change their UDF or their model without having to rebuild the prior layers of the container, as the first layers, (e.g., Python packages), will be cached.

For the next experiment, we wanted to evaluate the impact of layer caching in builds. For that purpose, we built incremental images with the same *Dockerfiles*, substituting only the model referenced by the `$MODELNAME` variable on line 9. The new model was trained with the same version of scikit-learn and had the same dependencies, as would be the case in a common model selection workload, so all the layers except the last one can be reused from the original image. In Figure 8, the right graph shows the build times for the incremental image. We see that the incremental image takes only .6s (slow disk) and .2s (fast disk), only 1.5% and .01% of the time spent building the original image. This could allow users to quickly iterate on their UDF and accompanying artifacts.

Note that it is possible to combine multiple UDFs in a single container provided that no UDFs have conflicting dependencies. For example, if a user created two UDFs with different models (a linear regression and a decision tree) with the same version of scikit-learn, the user could access each UDF from the same container using two different URL paths (ex: `myURL/linear` and `myURL/decisiontree`). This means there is *no* overhead for additional UDFs as the same container could be reused. However, they may chose to keep only a single UDF per container to simplify their workflow.

5.4.2 Impact of disk speed. When we created our test Spark cluster, local disk speed was not something that we considered, as our Spark data was all stored in a network-based location. However, during our exploration of VM configurations, we observed that the build and pull performance was heavily affected by disk access speed. Recall from §3.1 that the build phase involves compressing the image layers and writing them to disk, and the pull phase extracts them. We wanted to compare the impact of different disk alternatives on build performance. For that purpose, we used two different disks: (*i*) the default SSD for the VMs used in our Spark cluster experiments (120 Max IOPS and max throughput of 25 MBps), which we refer to as "slow" disk, and (*ii*) an ephemeral disk [64] (8000 max burst IOPS and max burst throughput of 200 MBps), which we refer to as "fast" disk. We monitored the disk usage using `iostat`, observing that the slow disk was saturated during the build step (the I/O utilization of the slow disk was higher than 200% at its peak), while the fast disk did not ever reach 100% utilization.

Figure 8 shows the results for the slow and fast disk variants. Observe that disk speed has noticeable impact over the build phase: 38s vs 31s for the original image, a ~1.2x speedup. This is bounded by the fixed cost of downloading the Python dependencies. The incremental image has a bigger speedup (3x) because the build step for the incremental image only copies the ML model into the last layer and compresses it. Pushing the image is network-dependent and therefore does not show much change, but the pull operation again shows a significant speedup due to the decompression phase.

In general, we see that disk speed plays a surprising role on multiple stages of the container's life cycle, and should be taken into consideration in local hosting.

§5 Key Takeaways:

- We found binary-based Arrow Flight to be a minimum of ~2.4x faster than the most efficient text-based alternative.
- Container start times vary wildly due to program size and complexity, from .07-7s in our experiments.
- Once a UDF is containerized, subsequent minor changes to the UDF can be done in .2-.6s in our experiments.
- Disk speed plays a large role in building and pulling container images due to compression, up to 3x in our experiments.

6 MANAGED CONTAINERIZED UDFS

In the previous section, we studied the performance of UDFs with an implementation using self-managed containerized UDF services and containers *local* to the cluster, i.e., co-located with the Spark executors. Another possible implementation would rely on a *remote* managed cloud service, e.g., Azure Machine Learning (AML) [61] or Azure Functions (AZF) [59], to act as the containerized UDF service and manage the containers. (Figure 2 depicted this difference.) Next we describe the trade-offs, across different dimensions, involved in choosing between local and remote solutions:

- **Security.** Locally hosted containers do not provide isolation and require self-management of security, whereas the use of a remote managed service could support multitenancy with isolation guarantees (if the service offers it). However, in contrast to the remote alternative, the local setup ensures that data does not leave the engine environment when the UDF is invoked.
- **Simplicity.** Managed solutions avoid the need for any infrastructure management and are often easier to setup. However, they can still require tuning confusing settings, and make troubleshooting performance issues and debugging more challenging. Locally hosted containers require self-management of infrastructure, but can provide more access to monitor and tune the system.
- **Overhead.** The local container solution introduces some overhead due to the cost of communicating between the container and data engine. These overheads are amplified with remote services due to network latency and potential service limits [60, 81]. However, the relative overhead of the remote versus local solution can depend on the UDF profile, discussed in this section.
- **Resource Flexibility.** Local containers are limited to the compute resources available on the database engine. With remote solutions, however, resources can be configured on a per UDF basis. For example, compute-intensive UDFs could be run on containers with specialized hardware support such as GPUs.

First, in §6.1, for the containerized UDFs designed to be run with remote services, we look at the impact of running them in a remote setting as opposed to the local setting. Then, in §6.2 we look at a self-hosted local container registry vs a remote managed one, and see how this impacts container distribution time.

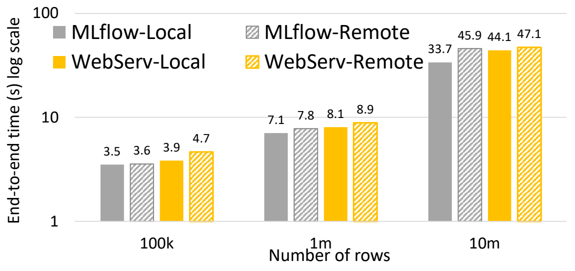

Figure 9: Comparing local and remote performance.

6.1 AML and Azure Functions

The goal of our next experiment was to understand the performance impact of using remote hosted services to manage the containerized UDFs. In Table 2, we introduced both the *Web Server* container, which was a container built from an AZF base image, and an *MLfLow* container built to be compatible with AML [61]. Both of these contain the same scoring UDF. We first ran these containers locally on our same Spark cluster. Then, we set up the managed services to run them in Azure in the same region that was hosting the Spark cluster. As our local container experimental Spark setup involved a pre-spun container-per-task, we requested that the remote service pre-spin the same number (12) of containers prior to running our experiments to match the local setup as closely as possible.

In Figure 9, we show the time to perform the scoring experiment with the AML *MLflow* containers using their online endpoint (in gray) and AZF (in yellow). Recall from §5.2 that the *Web Server* performs worse than *MLflow* because MLflow has built-in optimizations such as model caching. The same optimizations could be implemented in our *Web Server*. Also, *Web Server* is generic to all UDFs, which makes it a much broader solution than *MLflow*.

In general, one of the biggest downsides of using the remote managed services was difficulty in controlling the amount of compute to match with the output of the Spark cluster. Across the experiments, the remote setup adds roughly a 1.02-1.36x slowdown in the AML *MLflow* case and a 1.07-1.21x slowdown in the AZF *Web Server* case when compared to the same experiments locally. (These slowdowns are in addition to the overhead of the JSON/REST based protocols vs no container.) This is due to a combination network latency, throttling, and other overhead incurred in the managed services that we do not have insight into. The simplicity of the remote managed service is offset by the complications of performance optimizations.

Impact of UDF profile for a remote managed service. In §5.2, we discussed the impact of UDF profile on selection of a data communication method. The results from Figure 5 suggest that the UDF profile can also impact the choice between local or remote managed containers. For lightweight UDFs, a containerized solution with lower absolute overhead might be required indicating local containers. For compute-intensive UDFs, a remote service might have acceptable overhead as long as sufficient compute can be obtained. Although local containers may offer the optimal solution in terms of pure performance, there are other considerations for which a user might want to use a remote managed service as discussed earlier in this section. In such situations, *the profile of the user's UDF and thus the relative overhead of data transport to and from the container play a role in choosing the type of container service.*

6.2 Container registry location

It is common to rely on a fully managed container registry service to handle image distribution. However, administrators can also manage their own registry, e.g., the head node of a database cluster could host the registry used by the worker nodes. While the former alternative may provide built-in reliability, scalability, and security, the latter provides more flexibility and can potentially reduce latency depending on the network speed and node location.

We also measured the performance of the remote managed container registry compared to a registry set up locally. For that purpose, we started a new VM to act as the local registry, connecting it to the same vnet [62] as the original VM and enabled accelerated networking [55] between them. We then compared the end-to-end times to push/pull images using each registry implementation variant. First, when pushing/pulling the entire original images, we found that the local registry consistently provided a ~1.1-1.2x speedup over the remote one. For example, in the prior experiment, the push operation for the original image took 21 seconds, while it took 18 seconds with the local registry. This speedup would vary based on network speeds and remote registry configuration.

In a nutshell, network latency played a moderate role in the distribution of images in our setup. However, our results suggest that there is a trade-off between ease of use (including aspects such as reliability and security) and performance when choosing between a remote managed container registry and a local registry. An additional option not studied here is using a peer-to-peer registry setup between data engine nodes for even faster distribution [7, 46, 86].

§6 Key Takeaways:

- The overhead of using a remote managed service as opposed to a local self-managed one was ~1.02-1.36x in our experiments.
- Using a local container registry provided a speedup of at least 10% over the remote registry in our experiments.

7 EXTENSIBILITY

Effortless portability across platforms (different database engines) and the ability to run arbitrary code (for any UDF purpose and not locked to a language) opens a new set of use cases for UDFs. Suppose the data engine only supports UDFs written in language X and the user has a UDF written in language Y. Communication between the data engine and containerized UDF remains possible as long as languages X and Y have bindings in some standardized protocol. Standardized protocols such as REST/JSON or Arrow Flight have many language bindings, enabling greater flexibility in the language choices for UDFs for a particular data engine (§2.2 **L2**) and the reuse of the same UDF container across different data engines (§2.2 **L3**). The ability of the containerized approach to support UDFs in many languages enables users to rely on any library, and to transfer functions across data engines easily. For example, any data engine could call routines from the specially optimized library LAPACK [27] through its C/C++ APIs without modifying the database environment. This section comments on extending containerized UDFs to different platforms and different languages.

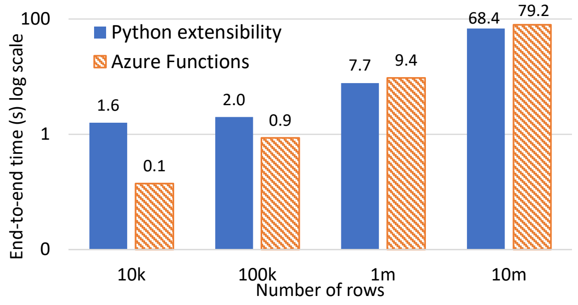

Figure 10: Scoring a model in SQL Server with Python extensibility and with Azure Functions.

Cross-platform portability. As we mentioned previously, the architecture introduced in §3.2 is not specific to Spark. To demonstrate the generality of this approach as well as the extensibility of running containerized UDFs across data engines, we set up a similar experiment on SQL Server. Using a single Azure VM (16 vcores, 32GB RAM) for SQL Server with DOP 16, we first scored the model from §4 without containers inside SQL using the *Python extensibility* feature which allows external scripts such as Python to be executed. We then reused the *Web Server* UDF container for model scoring with SQL Server using AZF as the remote endpoint.

Figure 10 shows that the Python extensibility has a fixed cost in this setup, but eventually at 10M rows the remote Azure Functions (AZF) experiment becomes slightly slower than the local setup mainly due to computational limitations of the specific AZF plan. In general, this experiment demonstrates the portability of the AZF *Web Server* from Spark to SQL Server due to the JSON language bindings that support many languages, including C++ (for SQL Server) and Python (for the containerized scoring UDF). Note that although the UDF model container is the same for both Spark and SQL Server, the underlying hardware, number and size of VMs, and settings in the AZF configurations across the experiments do not allow for a head-to-head comparison of the database engines.

Programming language flexibility. As mentioned before, the containerized approach to execution of UDFs also enables users to write UDFs in any programming language, even those not natively supported. For example, Spark does not support C# UDFs, but with the containerized UDF approach, we were able to implement a C# string manipulation UDF. Once again, this flexibility was enabled by JSON language binding support for both C# and Python.

§7 Key Takeaways:

- Reusing a containerized UDF between database engines is possible if both engines can implement the same protocol.
- Containers open up endless UDF extensibility, e.g., flexible programming language and library dependencies.

8 RELATED WORK

Containerized UDFs. Popular engines such as Snowflake and Redshift have recently added support for external functions [22, 78] backed by cloud serverless platforms that let users bring their own container images, e.g., AWS Lambda [80], Azure Functions [59], or Google Cloud Functions [36]. Communication between the engines and these platforms is done in JSON over HTTPS. These external functions allow users to write code in languages that native UDFs do not support or rely on libraries that native UDFs cannot access [21, 77]. Another system is InfluxDB Kapacitor, a real time stream processing engine that supports containerized UDF execution using protocol buffers for communication over UNIX sockets [43]. Additionally, Clipper [30] is a prediction serving system that uses containers for each model to achieve process isolation, and CEPLESS [52] uses a containerized user-defined operator interface to enable complex event processing. To the best of our knowledge, our work is the first to highlight the performance implications of various design choices in using UDFs implemented with containers, including relying on serverless compute platforms.

Containerized databases. Containerized UDFs could be seen as a step towards a broader containerized architecture for the whole engine. However, to provide granularized dependency isolation for example, the system components must be modularized. Currently most systems containerize the DB engine as a whole [63, 68].

Container operations. Various works have analyzed the performance of container life cycle operations. One early study [40] focused on container startup based on a representative Docker benchmark, showing that 76% of the time was spent on pulling the container image. Additional works [41, 88] examine container image characteristics that can impact push/pull times. In contrast to those works, we examine container performance within the context of database systems and in the form of containerized UDFs.

Further, the increasing popularity of containers over the last few years has led to significant improvements in container startup time and optimizations in container runtimes in general [24, 41, 44, 45, 51, 53, 87]. In addition, other recent works have focused on reducing container data access overhead. For example, Shimmy [23] and Faasm [83] rely on shared memory, while Cloudburst [85] uses a low-latency key-value store and caching. We are confident that efforts to improve container efficiency and performance will continue, broadening the range of use cases for containerized UDFs.

UDF optimizations and extensions. UDF usage in data processing systems is ubiquitous, and therefore, efficient execution of UDFs has received much attention over the years [29, 32, 38, 39, 42, 49, 74–76, 84]. For instance, multiple works have focused on transforming UDFs into SQL [39] or relational algebraic expressions [74] that can be more easily optimized and executed by a DBMS, while other approaches compile queries and UDFs written in different implementation languages into a unifying IR [32, 38] to enable optimizations across language boundaries. These works often focus on specific implementation languages for the UDFs, in contrast to containerized UDFs which provide more flexibility by supporting any languages and libraries in a sandbox environment. Further, some works have studied techniques to improve the integration between the data processing system and the UDF runtime, e.g., [76] uses row batches and buffer reutilization to accelerate the execution of Java UDFs in a C++ engine, while [49] focuses on compiling Python UDFs into optimized native code that can be efficiently executed by an engine written in C. Many of those techniques are applicable to UDF implementations based on containers. A possible direction for future work would be the optimization of queries with multiple containerized UDFs, for which some of this prior work on UDFs could apply. For instance, one could imagine cases where the execution performance could be improved by relying on, e.g., data reuse across inputs to the UDFs, caching of intermediate results in the containerized UDF, or allowing the containerized UDF service to execute multiple UDFs before returning control to the data engine.

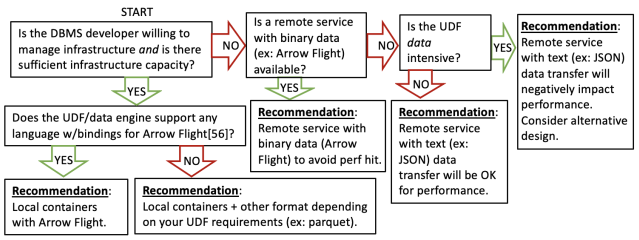
START
Is the DBMS developer willing to manage infrastructure *and* is there sufficient infrastructure capacity?
NO
YES
Is a remote service with binary data (ex: Arrow Flight) available?
NO
YES
Is the UDF *data* intensive?
YES
NO
Recommendation: Remote service with text (ex: JSON) data transfer will negatively impact performance. Consider alternative design.
Does the UDF/data engine support any language w/bindings for Arrow Flight[56]?
YES
NO
Recommendation: Remote service with binary data (Arrow Flight) to avoid perf hit.
Recommendation: Remote service with text (ex: JSON) data transfer will be OK for performance.
Recommendation: Local containers with Arrow Flight.
Recommendation: Local containers + other format depending on your UDF requirements (ex: parquet).

Figure 11: Flow chart of high level recommendations.

Finally, frameworks like Transport [70] and Portable [72] let the user implement a UDF only once using an engine-agnostic interface, and invoke it from multiple engines, e.g., Spark, Presto [16], etc. Currently those frameworks do not use containers, however they could be extended to use containerized UDFs and thus provide easier dependency management and portability across environments.

9 CONCLUSION

This paper presents a detailed study of containerized UDFs. Throughout the paper, we evaluate alternative implementations using self-managed vs hosted solutions, and discuss trade-offs across dimensions such as performance, simplicity, or flexibility. We also demonstrate the flexibility of containerized UDFs by executing them across platforms and languages. A simplified summary of implementation recommendations based on our findings is depicted in Figure 11. Although this diagram overlooks some of the complexities of containerized systems, it aims to provide some initial concrete recommendations based on our results.

Our results suggest important and promising directions for future work. For instance, wide adoption of more optimal (yet general-purpose) communication frameworks, such as Arrow Flight, could lead to significant performance improvements, especially for remote container services. In addition, native support for containerized UDFs in data engines could open up new possibilities for query optimization, such as UDF/operator fusion to reduce data access overhead as well as optimizing data communication for queries with multiple UDFs. Another interesting direction is integrating containers within frameworks aimed at generating cross-platform UDFs from engine-agnostic definitions, such as Transport UDFs. We expect the insights from our work will be useful to researchers and practitioners that are trying to make sense of the large number of alternatives to consider while designing this type of system.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their detailed and constructive comments that helped us improve the paper. We thank the MDCS (Microsoft Development Center Serbia) team, especially Andrea Batina and Vukashin Stefanovic, for the SQL Server experiment and related code. We also thank the following people that contributed to this work through their insightful feedback and collaboration: Jason Wilder, Carlo Curino, and Markus Weimer.

REFERENCES

[1] 2022. *Apache Arrow*. https://arrow.apache.org/

[2] 2022. *Apache Spark*. https://spark.apache.org/

[3] 2022. *Apache Thrift*. https://thrift.apache.org/

[4] 2022. *Docker*. https://www.docker.com/

[5] 2022. *Docker Hub*. https://hub.docker.com/

[6] 2022. *Docker Hub Python Images*. https://hub.docker.com/_/python/

[7] 2022. *Dragonfly*. https://d7y.io/en-us/

[8] 2022. *HDInsight*. https://azure.microsoft.com/free/hdinsight

[9] 2022. *JFrog Artifactory*. https://jfrog.com/artifactory/

[10] 2022. *Microsoft SQL Server*. https://www.microsoft.com/sql-server/

[11] 2022. *MLFlow Rest API*. https://www.mlflow.org/docs/latest/rest-api.html

[12] 2022. *numpy*. https://numpy.org/

[13] 2022. *Pandas UDF API - PySpark SQL Module*. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.functions.pandas_udf.html

[14] 2022. *pickle*. https://docs.python.org/3/library/pickle.html

[15] 2022. *pip - Requirements File Format*. https://pip.pypa.io/en/stable/reference/requirements-file-format/

[16] 2022. *PrestoDB*. https://prestodb.io/

[17] 2022. *scikit-learn*. https://scikit-learn.org/

[18] 2022. *SGDRegressor documentation refers to non-existent loss metric*. https://github.com/scikit-learn/scikit-learn/issues/23375

[19] 2022. *Sigma Computing*. https://www.sigmacomputing.com/

[20] 2022. *sklearn.utils.fixes.MaskedArray removed in 0.23.0*. https://github.com/scikit-learn/scikit-learn/issues/17198

[21] 2022. *Snowflake: Advantages of External Functions*. https://docs.snowflake.com/en/sql-reference/external-functions-introduction.html#advantages-of-external-functions

[22] 2022. *Snowflake: Introduction to External Functions*. https://docs.snowflake.com/en/sql-reference/external-functions-introduction.html

[23] Marcelo Abranches, Sepideh Goodarzy, Maziyar Nazari, Shivakant Mishra, and Eric Keller. 2019. Shimmy: Shared Memory Channels for High Performance {Inter-Container} Communication. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*.

[24] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*. 419–434.

[25] Kavita Agarwal, Bhushan Jain, and Donald E. Porter. 2015. Containing the Hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*. 1–9.

[26] Tanveer Ahmad, Zaid Al Ars, and H. Peter Hofstee. 2022. Benchmarking Apache Arrow Flight – A wire-speed protocol for data transfer, querying and microservices. https://doi.org/10.48550/ARXIV.2204.03032

[27] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[28] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[29] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 3–14.

[30] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.

[31] Criteo. 2022. *Criteo 1TB Click Logs dataset*. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[32] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.* 8, 12 (2015), 1466–1477.

[33] Databricks. 2022. *Databricks Notebook-scoped Python libraries*. https://docs.databricks.com/libraries/notebooks-python-libraries.html

[34] Docker. 2022. *Best practices for writing Dockerfiles*. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#use-multi-stage-builds

[35] Raul Castro Fernandez, Pranav Subramaniam, and Michael J. Franklin. 2020. Data Market Platforms: Trading Data Assets to Solve Data Problems. *Proc. VLDB Endow.* 13, 11 (2020), 1933–1947.

[36] Google. 2022. *Cloud Functions*. https://cloud.google.com/functions

[37] Google. 2022. *Google Cloud Container Registry*. https://cloud.google.com/container-registry

[38] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient Execution of Polyglot Queries. *Proc. VLDB Endow.* 15, 2 (2021), 196–210.

[39] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *11th Conference on Innovative Data Systems Research, CIDR*.

[40] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast distribution with lazy docker containers. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 181–195.

[41] Zhuo Huang, Song Wu, Song Jiang, and Hai Jin. 2019. Fastbuild: Accelerating docker image building for efficient development and deployment of container. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 28–37.

[42] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. *Proc. VLDB Endow.* 5, 11 (2012), 1256–1267.

[43] InfluxDB. 2022. *Write socket-based user-defined functions (UDFs)*. https://docs.influxdata.com/kapacitor/v1.6/guides/socket_udf/

[44] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 257–262.

[45] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[46] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. 2017. Fid: A faster image distribution system for docker platform. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 191–198.

[47] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020*.

[48] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies. *ACM Trans. Database Syst.* 45, 2 (2020), 7:1–7:66.

[49] Steffen Kläbe, Robert DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Accelerating Python UDFs in Vectorized Query Execution. In *12th Conference on Innovative Data Systems Research, CIDR*.

[50] Hyukjin Kwon. 2020. *How to Manage Python Dependencies in PySpark*. https://databricks.com/blog/2020/12/22/how-to-manage-python-dependencies-in-pyspark.html

[51] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 159–169.

[52] Manisha Luthra, Sebastian Hennig, Kamran Razavi, Lin Wang, and Boris Koldehofe. 2020. Operator as a service: Stateful serverless complex event processing. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 1964–1973.

[53] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. 2018. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 181–188.

[54] Wes McKinney. 2019. *Apache Arrow Flight*. https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/

[55] Microsoft. 2022. *Accelerated Networking*. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli

[56] Microsoft. 2022. *Azure Container Registry*. https://azure.microsoft.com/services/container-registry/

[57] Microsoft. 2022. *Azure Container Registry Service Tiers*. https://docs.microsoft.com/en-us/azure/container-registry/container-registry-skus

[58] Microsoft. 2022. *Azure Data Lake Storage*. https://azure.microsoft.com/en-us/services/storage/data-lake-storage/

[59] Microsoft. 2022. *Azure Functions*. https://azure.microsoft.com/services/functions/

[60] Microsoft. 2022. *Azure Functions hosting options: service limits*. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#service-limits

[61] Microsoft. 2022. *Azure Machine Learning*. https://docs.microsoft.com/azure/machine-learning/

[62] Microsoft. 2022. *Azure Virtual Network*. https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview

[63] Microsoft. 2022. *Create Azure Arc-enabled SQL Managed Instance using Kubernetes tools*. https://docs.microsoft.com/en-us/azure/azure-arc/data/create-sql-managed-instance-using-kubernetes-native-tools

[64] Microsoft. 2022. *Dv3 and Dsv3-series*. https://docs.microsoft.com/en-us/azure/virtual-machines/dv3-dsv3-series

[65] Microsoft. 2022. *Manage libraries for Apache Spark in Azure Synapse Analytics.* https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-azure-portal-add-libraries

[66] Microsoft. 2022. *NYC Taxi & Limousine Commission - yellow taxi trip records.* https://docs.microsoft.com/en-us/azure/open-datasets/dataset-taxi-yellow?tabs=pyspark

[67] Microsoft. 2022. *Power BI.* https://powerbi.microsoft.com/

[68] Microsoft. 2022. *Running Apache Spark jobs on AKS.* https://docs.microsoft.com/en-us/azure/aks/spark-job

[69] MLflow. 2022. *MLFlow Models / Model API.* https://www.mlflow.org/docs/latest/models.html#model-api

[70] Walaa Eldin Moustafa. 2018. *Transport UDFs.* https://engineering.linkedin.com/blog/2018/11/using-translatable-portable-UDFs

[71] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

[72] Tejas Patil. 2021. *Portable UDFs: Write Once, Run Anywhere.* https://databricks.com/session_na21/portable-udfs-write-once-run-anywhere

[73] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[74] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.

[75] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. 2017. Optimization of Complex Dataflows with User-Defined Functions. *ACM Comput. Surv.* 50, 3 (2017), 38:1–38:39. https://doi.org/10.1145/3078752

[76] Viktor Rosenfeld, Rene Mueller, Pinar Tözün, and Fatma Özcan. 2017. Processing Java UDFs in a C++ environment. In *Proceedings of the 2017 Symposium on Cloud Computing.* 419–431.

[77] Brandon Schur. 2022. *Data Tokenization with Amazon Redshift and Protegrity.* https://aws.amazon.com/blogs/apn/data-tokenization-with-amazon-redshift-and-protegrity/

[78] Amazon Web Services. 2022. *Amazon Redshift: Creating a scalar Lambda UDF.* https://docs.aws.amazon.com/redshift/latest/dg/udf-creating-a-lambda-sql-udf.html

[79] Amazon Web Services. 2022. *Elastic Container Registry.* https://aws.amazon.com/ecr/

[80] Amazon Web Services. 2022. *Lambda - Serverless Computing.* https://aws.amazon.com/lambda/

[81] Amazon Web Services. 2022. *Lambda quotas.* https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[82] Prateek Sharma, Lucas Chaufournier, Prashant J. Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference.* 1–13.

[83] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20).* 419–433.

[84] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data.* 1718–1731.

[85] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452.

[86] Uber. 2022. *Kraken.* https://github.com/uber/kraken

[87] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18).* 133–146.

[88] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K Paul, Keren Chen, and Ali R Butt. 2020. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems* 32, 4 (2020), 918–930.