DOI: 10.1515/SBEEF-2019-0011

INNOVATIVE DEVOPS FOR ARTIFICIAL INTELLIGENCE

R. CIUCU[1], F.C. ADOCHIEI[1], IOANA-RALUCA ADOCHIEI [2], F. ARGATU[1], G.C. SERIȚAN[1], B.ENACHE[1], S. GRIGORESCU[1], VIOLETA VASILICA ARGATU [1]
[1] University POLITEHNICA of Bucharest, Romania, [2] Technical Military Academy "Ferdinand I" Bucharest Romania
E-mail: felix.adochiei@upb.ro

Abstract. Developing Artificial Intelligence is a labor-intensive task. It implies both storage and computational resources. In this paper, we present a state-of-the-art service-based infrastructure for deploying, managing and serving computational models alongside their respective data-sets and virtual environments. Our architecture uses key-based values to store specific graphs and datasets into memory for fast deployment and model training, furthermore leveraging the need for manual data reduction in the drafting and retraining stages. To develop the platform, we used clustering and orchestration to set up services and containers that allow deployment within seconds. In this article, we cover high-performance computing concepts such as swarming, GPU resource management for model implementation in production environments with emphasis on standardized development to reduce integration tasks and performance optimization.

Keywords: Artificial Intelligence; HPC; Clustering; PAAS; Orchestration; DevOps;

1. INTRODUCTION

Since the emergence of Deep Learning frameworks, teams of researchers have been developing computational models that can synthesize natural language, recognize handwriting and objects and classify specific objects based on learned and identified characteristics [1-2].

Development operations (DevOps) is a new field in software system engineering that tackles with concepts such as "agile infrastructure" and "agile operations", and it deals with all the stages of development, deployment, and production of services and micro services architectures [3].

We decided to build this platform to bring together a team of professionals from different fields and backgrounds to streamline the development, management, and training of computational models.

While neural networks provide potent means to process data, the frameworks themselves are highly dependent on hardware resources such as graphical processing units (GPU), development environments and the core programming language. Out of the many implementations of neural networks, two distinguished players emerged. The first one is Python, which provides digital capabilities (NumPy), ComputerVision (dlib, OpenCV) [5] and data parsing (PyCSV, Pickle). The second one is TensorFlow (TF), which is a relatively new AI framework developed by Google and had widespread success over the past three years [4-6].

While Python and TF provide a flexible approach to developing applications and scripting specific tasks, the combination is highly version dependent, with libraries that are ported over from other languages like C and LUA, many of them requiring to be compiled and built for the machine.

We propose to store specific datasets and parameters in a standardized way, to ease the data flow to the computational models and ensure that each computing session can be addressed individually by the data is processed and the final output.

In the materials and methods section, we describe our approach to distribute the virtual environment and datasets into a containerized format, the way we built our high-availability cluster to support the proposed platform and the hardware components of the system.

The results section deals with the integrated software services and micro services that govern each computing session and orchestrate the swarm of containers that run at any given time within our platform.

We conclude the paper by discussing the performance and the scalability of the proposed architecture, emphasizing the future model implementations and data flow optimizations.

2. MATERIALS AND METHODS

The three main principles of DevOps, as stated by Agile Admin [3], are:

- **Infrastructure Automation** – Create, configure and package each application to be automated
- **Continuous Delivery** – Build test and deploy within an agile environment
- **System Reliability Engineering** – Operation, orchestration, and monitorization of each container

In past years multiple Artificial intelligence frameworks emerged such as TensorFlow (TF), Caffe, Apache SINGA, Microsoft Cognitive Toolkit, and Torch. Devops for such frameworks deals mostly with the environment setup and dataset management.

2.1 TensorFlow VirtualEnvironment

The virtual environment is a bridge between assets and resources such as libraries and the application dependencies. TF has two device layers; one dedicated to the GPU component and the other to the CPU. Therefore, two virtual environments must be developed [7].

The TF kernel encapsulates the computational operations which are called as jobs and tasks by the client application. Thus having multiple jobs and workers a parallelized computing session can be implemented in each model [7].

To each graph within a session, a section of the GPU resources is allocated. To scale the application session arguments need to be exposed as environment variables. Modifying those values affects the performance and the training times of the model. The primary values that the session attributes to the graph are the allocated memory per process and the device that runs each operation [8].

2.2 TensorFlow Models as a Service

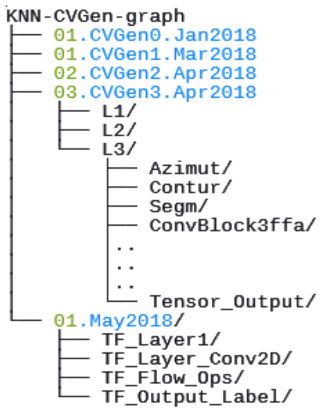
For passing the input data from python to TF, the objects need to be serialized and transformed into vector values. If the given model has a known data structure, the local Pickle and SVM files can be leveraged for a distributed resource approach.

A memory resident database (MRDB) is a type of database that stores keys and values in-memory for fast access. It allows for a flexible data structure design, class-object relations, and asynchronous transactions. Rediscan be used as a task execution queue and fast data serving [9-10].

In Figure 1 is illustrated the MRDB model structure; each model is categorized by their specific type, while the stored weights can be generational and are serialized.

Tf graphs depend on the session in which are called, the operations executed within the blocks of the graph and the resource allocation that is specific to the session.

Serializing such a request can be done using a JSon (JavaScript Object Notation) object then passed as an execution command to the data intake node.



```
KNN-CVGen-graph
├── 01.CVGen0.Jan2018
├── 01.CVGen1.Mar2018
├── 02.CVGen2.Apr2018
├── 03.CVGen3.Apr2018
│   ├── L1/
│   ├── L2/
│   └── L3/
│       ├── Azimut/
│       ├── Contur/
│       ├── Segm/
│       ├── ConvBlock3ffa/
│       │..
│       │..
│       │..
│       └── Tensor_Output/
└── 01.May2018/
    ├── TF_Layer1/
    ├── TF_Layer_Conv2D/
    ├── TF_Flow_Ops/
    └── TF_Output_Label/
```

Figure 1. Redis Tree View of Generational Models

2.3 Persistent Memory Operations

Persistent memory (PMEM) programming is a relative new field in computer engineering. Historically RAM mapping and persistent memory have been used in order to study and analyses software behavior during malfunctions by analyzing what was stored or what was the state of the software at the moment of a crash or a fatal error. Persistent memory allows for instant delivery of information from certain data structures by using a MRDB specialized for storing and altering data directly into the memory [11].

2.4 Container pipeline

The container development workflow implies a set of conventions such as environment variables, packaged dependencies, and local storage data tables.

The containers can depend on resources that are shared and called locally within the compose file thus deploying a bundle of containers that are interlinked.

Containers can be built automatically by the orchestration engine on specific triggers which alter environment changes [8].

The pipeline consists of developing and adapting each model to a preferred state, packaging the dependencies using package managers such as NPM, PIP and building local caches of libraries and assets.

The Container is built as an image which contains a Linux kernel, a set of instructions for the container engine (compose.yaml) and the application itself.

Each image is stored within a registry or a local volume from which the orchestration software pulls, deploys and monitors the container.

The container lifetime depends on the initialization parameters and the exit code. Each container should be built with a Command Line Interface Logging component to monitor activities such as API-Calls, debugging information and process states.

2.5 Container Orchestration

Orchestration implies the organizing and planning, containers runtime.

The container engine provides a set of tools which are iterated upon for load balancing and runtime optimization. A lot of empiric data based on logs can be useful for manual balancing, but for long-term training sessions with a model that span multiple layers and data processes, automated load balancing is employed.

A typical orchestration architecture employs two types of nodes [12]:

- The worker nodes primarily execute the machine learning tasks with load balancing

- Manager nodes do the routing tasks and request handling, based on the runtime conditions

The proposed orchestration is illustrated in Figure 2. The architecture was implemented on three swarm nodes, with each node having a swarm manager container. The central manager node acts as a bridge between the swarm of containers and the PMEM database. By managing requests with an execution queue, the serving process can be insured [13].

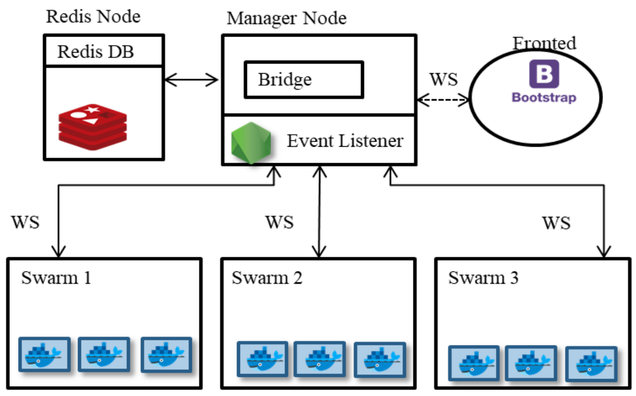


Figure 2. Proposed Swarm Architecture

Another aspect of the orchestration service is that each new container that is deployed is allocated to a swarm that follows a set of rules from on a container distribution table. The set of rules can also be implemented as a task scheduler or event scheduler based on API Calls or data flow inputs. As mentioned above, based on the container exit code, runtime parameters can be set manually by processing the log file and querying each request made to the database node.

2.6 Hardware and Software Resources

The bare-metal hardware consists of two 8 core CPUs with 16 threads and 68 GB of DDR4 Ram Memory for each of the three machines. Two of the servers used two Nvidia TitanV GPU cards while the production machine used 2 Nvidia Quadro integrated graphics cards.

To orchestrate our architecture, we used the open-source virtualization and containerization engine Docker CE. The services implemented to handle each request and event within our stack are based on node.js with WebSocket pipelines and docker API integration.

For the first layer of clustering, virtualization and persistent memory partitioning was used VMWare ESXi and each node ran on Ubuntu CE server 18.04.

2.7 Machine Learning Pipeline

A production pipeline is often used to train and deploy computational models. The pipeline is centered on the main container which is connected to the microservices that govern and feed serialized datasets. The main player is the fast access to data through a persistent memory database and a set of parameters that are constantly checked by using an event listener type microservice. The final results and constant statistics are sent to the frontend for analysis.

3. RESULTS AND DISCUSSIONS

Into our container stack, we integrated software services that handle each requested object and managed each computing session. The software services ensure the data flow between the in-memory data structures components and each container. Container orchestration is done within a dedicated swarm manager per cluster; in total we have three swarms. The first two swarms are used for training and managing datasets while the third is used for production operations. Our training clusters have a direct GPU and RAM allocation, addressed by the event handler service (Figure 3).

3.1 Event Handler

Within each container, several environment variables are exposed. Those are specific values and parameters such as which port is mapped for the PMEM database connector, Boolean indicators that tell the handler if the session uses either GPU or CPU training or what dataset the client container has requested.

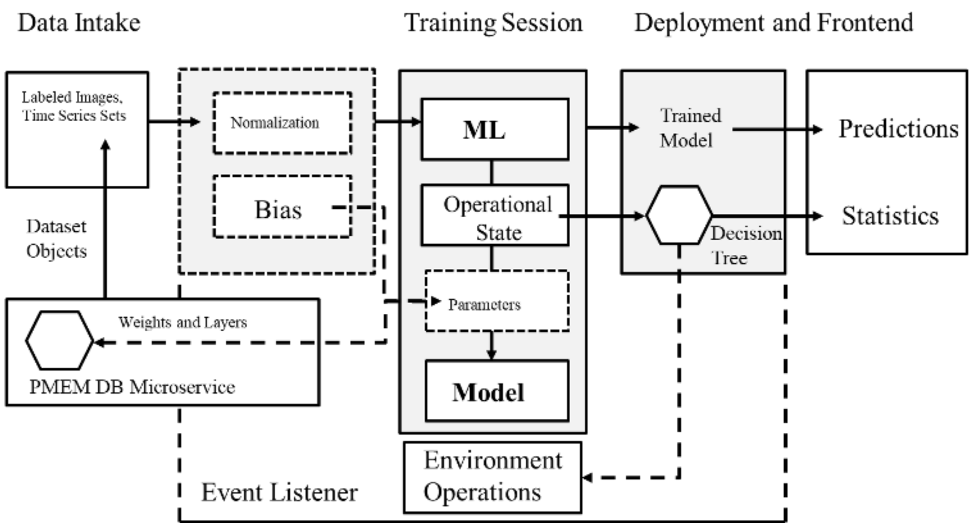


Figure 3. Pipeline Overview

After all the values are parsed, a new session is recorded in PMEM, and the declared dataset is served into the requesting container. By using serialized objects into our database, each dataset can be transferred through JSON arrays into the container and interpreted into the specific computational structure. Objects such as pictures are stored as byte-arrays with the adjacent fields storing dimensional attributes and the corresponding unique keys. Serializing SVM files is done by exposing the label tensors as entity names and the following fields parsed as numeric vector values between the database and the container. Building the service as a request handler ensures that the pipeline between each resource can be indirect queried and the response time can be measured to ensure the lifetime of the container. Direct links can cause surges of requests, without any load-balancing measures one session can take more resources than needed and thus harming the performance of each node.

The event handler is implemented as a separate container, introduced within the stack as a manager node. The WebSocket server is built with Node.js leveraging the need to integrate API-Calls to the orchestration engine manually. Mediating the requests is the PMEM component which creates a cache list of each active session, along with each parameter. We developed and archiving component which stores every event passed to the database along with the execution and exit code of each container. Such logs can be used to optimize performance and monitor the behavior of each implemented model (Table 1).

Table 1. Platform Metrics

Deployment Metrics	Container Metrics	Swarm Metrics
Boot time	~5 s	120 s
AVG Memory Allocation	2.5 GB RAM /Container	64GB /Node
CPU Allocation	2CPU / Container Min Or *Requested*	4 /Manager Node
GPU Allocation	*Model Dependant*	Per Container
Size	780 MB / Image	128 GB Total/ Node

3.2 Event Listener

The event listener service has two purposes, one is to pass the requests to the event handler through a WebSocket connection, and the other is to bridge the software resources to the event handler. This service directly implemented within each production unit. Specific structure tables and object deserialization procedures are directly implemented to ensure that the data passed from PMEM is formatted for TF. SVM injection within specific layers of the model is done if the data reduction stages have already been performed and the dataset has already been processed. The event listener also parses the requests between the shell execution and the frontend (Figure 4).

Our event listener node uses a simple decision tree based on the success rate and weights comparison. The microservices can interrupt the session and flag each specific container if and exit code is given during training times. The event listener takes the following parameters into consideration: the state of the operational environment (training, deep learning or production), how fast the state or stage changed and what is the order of operations before this state. Since weights are generated on each dataset and specific layers of parameters can be instated we considered creating manually intervals in which we accept a specific model or we end the session.

3.3 Data Intake Node and Production pipeline

The production pipeline depends heavily on how the persistent memory is used in order to facilitate the dataset loading and model management.

We integrated a database connector that has direct access to the persistent memory of the cluster, up to this point no further studies on persistent memory use were published so we entered an uncharted territory. On the other hand RAM memory has its advantages while it can cause memory leaks from the container system so we leveraged our key-based system in order to maximize the data intake of this component.

This microservice bundles serialized objects coming from direct queries and fast access to storage on-the-go for session related data such as: order of operations, epoch information and container hardware usage. Our personal contribution to this microservice is the combination of neural networks and persistent memory data structures. In future works we will try to implement a short-term and long-term memory system for training neural networks

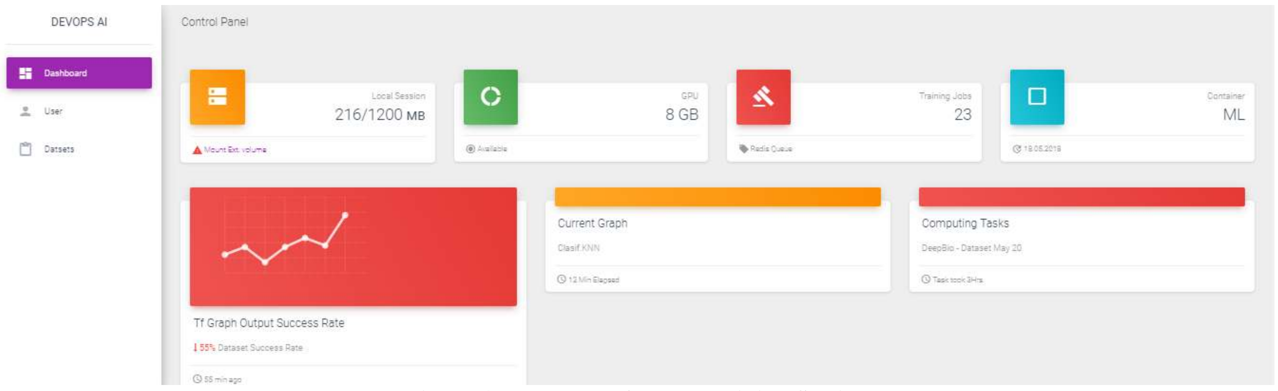


Figure 4. Dashboard for the Training Session

on specialized subjects and also benchmark our pipeline using this approach.

The data intake microservice feeds all the dataset from memory to multiple containers that house the computational model. Using a persistent memory system we can also store layers of hyper-parametrization and weights without altering the production model and allowing the reuse of a container collection without building it every time we need to be retrain and deploy a model for a study or application.

3.4 Discussions

We developed this platform to ensure continuous development of computer vision algorithms. We tested some public models, from the implementations stages into our data structure. For each model a section of GPU resources was manually allocated if the model was compatible with a GPU implementation. Tighter load balancing operations can be made by allocating CPU resources directly to the container. Training operations that exceed more than 5 hours are automatically passed to a low priority node. Shortest GPU benchmark took 4.5 seconds on the Inception Resnet V1 model with a dataset of 250.00 serialized and labeled objects.

Container distribution was realized by bundling the development nodes with the TitanV servers, and the production of containers with the Quadro server. Models are passed into production if specific internal requirements are met, from scalability to dataset optimization. Using a bigger GPU memory pool (TitanV) ensures that more containers can be tested on the same datasets and concurrently.

Other models that are friendly to SVM injection used pattern recognition for cellular classification, as "DeepBio", pattern recognition for time series such as "Energy Price Forecast", and physiological signs correlations with convolutional nests [14].

4. CONCLUSIONS

This paper presented innovative software architecture for managing virtual environments, computational model resources and data structures for artificial intelligence applications. Furthermore, we described production cases, leveraging local computing sessions for a platform as a service-centric approach, to bring together multiple computational models with applications in computer vision, pattern recognition.

By ensuring a standardized workflow, the development environment is controlled and stable. We provided a scalable solution for event management that is automated and can process many requests. Our contribution to the container platform by serving the datasets and graphs proved to be successful, because of our data structure approach where each object can be called directly from memory. Every node and graph can be executed and evaluated based on the requests, delivery and execution times.

4.1 Difficulties

While the graph and dataset serving component that implemented in PMEM is fast, it inhabits almost all the available memory as seen in the metrics table. We can only store serialized objects that have already been processed by data reduction procedures. Furthermore, the Docker engine is prone to memory leaks within large numbers.

4.2 Future Perspectives

In future papers, we will address our internal computer vision models, and the input data flows of real-time video capture devices. On the HPC side, we are looking into migrating the services from Docker containers to Kubernetes pods. We are confident that our approach proves that small teams of researchers can develop, operate and maintain PAAS stacks.

5. ACKKNOWLEDGMENTS

This paper was developed within the program National Project: GNaC 2018 ARUT no. 19/06.02.2018, Contract nr. ET01-19-03, Act. 4000.228 from Politehnica University of Bucharest.

6. REFERENCES

[1] Mo, Y. J., Kim, J., Kim, J.-K., Mohaisen, A. and Lee, W., Performance of deep learning computation with TensorFlow software library in GPU-capable multi-core computing platforms, In: 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN). doi: 10.1109/icufn.2017.7993784.

[2] Agile Admin "What is DevOps: https://theagileadmin.com/what-is-devops/ As of 19 May 2018.

[3] Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, Computer Vision and Pattern Recognition Going Deeper with Convolutions, In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[4] K. Wongsuphasawat et al., Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow, IEEE Transactions on Visualization and Computer Graphics, vol. 24, no. 1, pp. 1-12, Jan. 2018.

[5] Rutkowski, Leszek, Image classification with recurrent attention models, In: Artificial Intelligence and Soft Computing, 11th International Conference, ICAISC 2012, Zakopane, Poland, April 29 - May 3, 2012: Proceedings. Springer, 2012.

[6] Ian Miell, Aidan Hobson Sayers, Docker in Practice, 1st Manning Publications Co. 2016.

[7] Nishant Shukla, Machine Learning with TensorFlow Manning Publications Co. 2018.

[8] Srdjan Grubor, A practical guide to rapidly and efficiently mastering Docker containers, along with tips and tricks learned in the field Packt Publishing Ltd, Nov 22, 2017.
[9] Jeremy Nelson, Mastering Redis, Packt Publishing Ltd, May 31, 2016.
[10] Maxwell Dayvson Da Silva, Hugo Lopes Tavares, Redis Essentials, Packt Publishing Ltd, Sep 8, 2015.
[11] P. Mehra and S. Fineberg, Fast and flexible persistence: the magic potion for fault-tolerance, scalability and performance in online data stores, In: 18th International Parallel and Distributed Processing Symposium, 2004.
[12] Fabrizio Soppelsa, Chanwit Kaewkasi Native Docker Clustering with Swarm, Packt Publishing Ltd, Dec 20, 2016.
[13] Clouds Andrew J. Younge, Kevin Pedretti, Ryan E. Grant, Ron Brightwell, A Tale of Two Systems: Using Containers to Deploy, In: HPC Applications on Supercomputers and 2017 IEEE 9th International Conference on Cloud Computing Technology and Science.
[14] Adochiei Felix Constantin, “Contributions to Biological Signal Processing using Embedded Systems”, PhD Thesis, POSDRU CUANTUMDOC - RESEARCH GRANT.