White Paper

CoSP, TEMs, ISVs, SIs
Telecommunication

EXFO Accelerates AI Pipelines with Intel® Xeon® Scalable Processors and Optimized Software Framework

Companies work together to fine-tune each stage of a telecom network AI pipeline for maximum cost-efficiency and reduced power consumption. The results include a 57% reduction in total runtime and a 17% reduction in data size.

Authors

Sylvain Nadeau
Director – Strategic Innovation, EXFO

Ludovik Mondou
Applied ML Engineer, EXFO

Michael Burkardsmaier
Solution Architect, Intel

Herve Mer
Segment lead OSS/BSS, Intel

Anas Ahouzi
AI/ML Engineer, Intel

Contributors

Jerome Thiery
Software Specialist, EXFO

Loic Le Gal
Software Specialist, EXFO

Dmitry Chigarev
AI Frameworks Engineer, Intel

Iaroslav Igoshev
AI Frameworks Engineer, Intel

Andreas Huber
AI Frameworks Engineer, Intel

Dmitry Razdoburdin Ph.D
AI Frameworks Engineer, Intel

Executive Summary

In the fast-evolving landscape of telecommunications, automation of networks and services is paramount as telecom operators seek to streamline operations and build agile service platforms. This whitepaper delves into how communications services providers (CoSPs) can use artificial intelligence (AI) to revolutionize the analysis of network performance and health, eliminating static rules and thresholds significantly improving root cause analysis, and paving the way toward autonomous networks.

Testing done for this paper optimized the entire AI pipeline (see Figure 1), from data ingestion and preparation to training and inference. These tests reminded us that the often-overlooked data pre-processing stage consumes considerable compute resources of an overall AI workload.

Through concrete examples and measurements, this paper illuminates how each stage of the pipeline can be fine-tuned to achieve maximum cost-efficiency. With a specific focus on telecom network AI workloads, that are dominated by large amounts of time-series data, the research underscores the critical role of ETL in a machine learning (ML) pipeline and how to leverage Intel® oneAPI Runtime Libraries to achieve nearly linear scaling.

The result of the fine-tuning described is an impressive 57% reduction in total runtime and a 17% reduction in data size. Together these translate into reduced power consumption for an on premises deployment or substantial cost savings, when deployed on cloud infrastructure. Remarkably, these outcomes were achieved with just a few simple code changes, showcasing the immense potential for enhancing both software and hardware aspects of AI pipelines on Intel® architecture servers.

The optimizations described in this paper:

- Investigate hotspots across the entire pipeline, including data ingestion, pre-processing, training and inference.
- Consider data formats and compression schemes for optimal storage size vs. compute time.
- Leverage Intel oneAPI Runtime Libraries for optimal performance.
- For improved total cost of operation (TCO) identify optimal hardware profile or cloud instance types and size for deployment:
  - Memory vs. compute bound, number of cores.
  - Scaling the application with larger datasets.
  - Evaluating the need for accelerators.

Key Features

Telecom Network AI Analytics	Accelerate ETL and Training	Intel® Xeon® Performance Gains
Industry Telecommunications	**Use case** Assurance	**Learn more** EXFO

Figure 1. Overview of how AI process fine tuning was conducted.

EXFO is an Intel® Network Builders ecosystem partner and develops smarter test, monitoring and analytics solutions for the global communications industry. The company is a trusted adviser to fixed and mobile network operators, hyperscalers, and leaders in the manufacturing, development and research sector. EXFO customers count on the company to deliver superior visibility and insights into network performance, service reliability and user experience.

This paper details the company's work with Intel to further enhance EXFO's Adaptive Service Assurance (ASA) platform by optimizing the AI/ML workloads across the data analytics pipeline.

Problem Statement

The advent of 5G is accelerating the replacement of purpose-built network equipment with the virtualization and cloudification of network functions. The value of cloud-native networks lies in decoupling network and service topologies from the underpinning hardware infrastructure. While bringing immense new flexibility to telecom network design, there is a corresponding loss of visibility due to hardware abstraction, making it more difficult to troubleshoot network performance issues. Performance of the cloud is added to network coverage and performance as key determinants of end-user experience.

Cloud infrastructure abstraction

Cloud infrastructure abstraction decouples the capability of associating elements of the network and service topologies to specific hardware. While cloud-native principles simplify deployment, provisioning and maintenance, the resulting visibility constraints make it difficult to correlate customer quality-of-experience (QoE) issues with cloud infrastructure issues.

Operations silos

Network and service operations teams manage quality of service (QoS) and QoE issues in the virtualized cloud-native network. IT and cloud operations teams manage the cloud infrastructure. They traditionally use different tools and measure different metrics, making it challenging to address cross-domain issues.

Data tsunami

5G cloud-native networks generate a lot of performance data—more than 40 petabytes per hour. Combine this with infrastructure observability data and the result is a data tsunami. Moving, storing, processing, and extracting actionable insight from this data will be cost prohibitive without a significant rethink of assurance in cloud networks.

Cross-domain analysis

In collaboration with Intel, EXFO has developed a full-stack service assurance solution that combines infrastructure observability with telecom-specific key performance indicators for AI-enabled analytics. Thanks to this collaboration, EXFO's ASA platform is now closing the visibility gap in cloud-native networks, delivering true bare-metal-to-customer-experience visibility (see Figure 2).

EXFO leverages Intel® Platform Telemetry Insights to achieve service-to-infrastructure cross-domain analysis, giving operators an integrated view of services and underlying infrastructure performance. Using correlated visibility across network, service and cloud infrastructure layers, combined with automated diagnostic tests, we were able to pinpoint the origin of degradations.

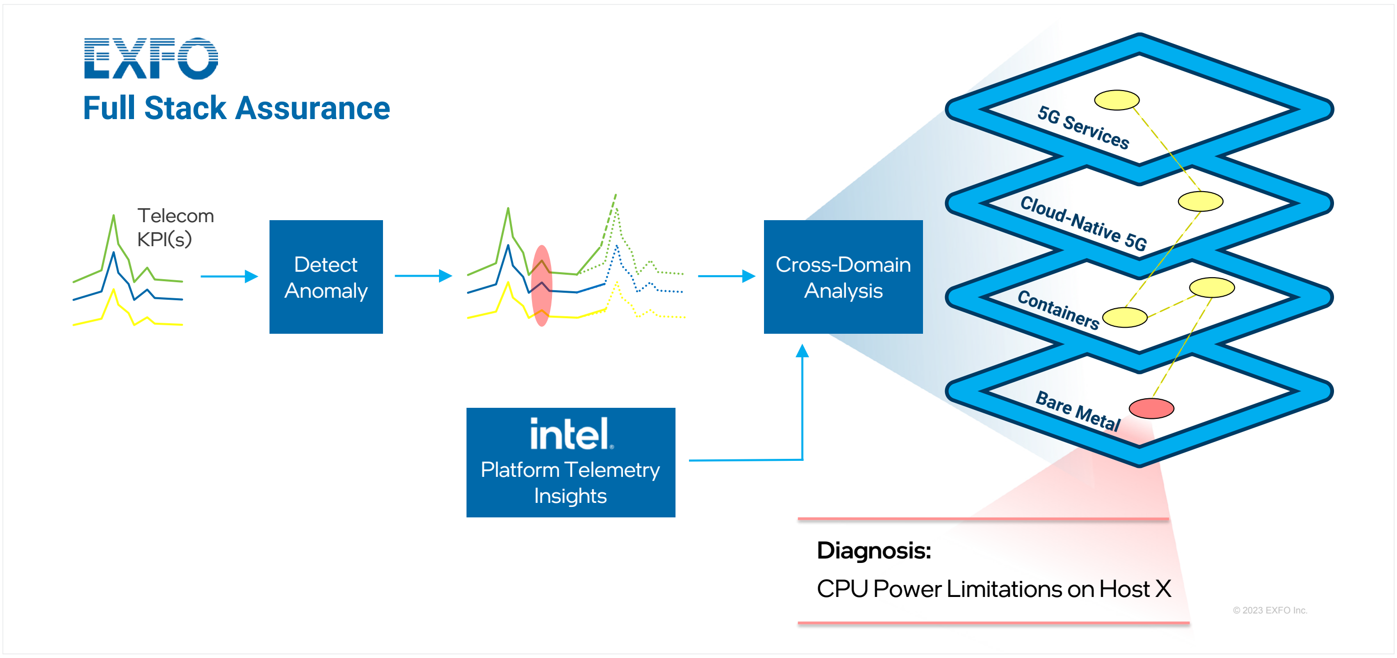


Figure 2. EXFO and Intel have developed Full Stack Assurance that provides visibility into cloud-native networks resulting in the ability to see network issues from the bare metal server all the way to the customer's services.

Using the capabilities of EXFO's ASA platform, it is possible to locate performance issues across domain layers and accelerate troubleshooting by focusing on customer-impacting network degradations.

The cross-domain analysis module can detect typical degradation use cases related to:

- **Performance monitoring:** resource utilization and exhaustion
- **Fault monitoring:** networking congestion, unstable software components
- **Power monitoring:** impact of power management

Additionally, EXFO and Intel jointly collaborated to optimize the entire AI/ML pipeline of the ASA platform cross-domain analysis module.

Pipeline Testing Methodology

Every stage of the pipeline was thoroughly examined (see Figure 3), starting with data ingestion and conversion (from JSON to Parquet), followed by the discovery of features and dimensions, and ultimately encompassing training and model interpretation. During this analysis, it became obvious there was potential for improvements in all three pipeline stages.

It's crucial to emphasize that all the data utilized in this analysis originates from real, live data sources and is not synthetic data.

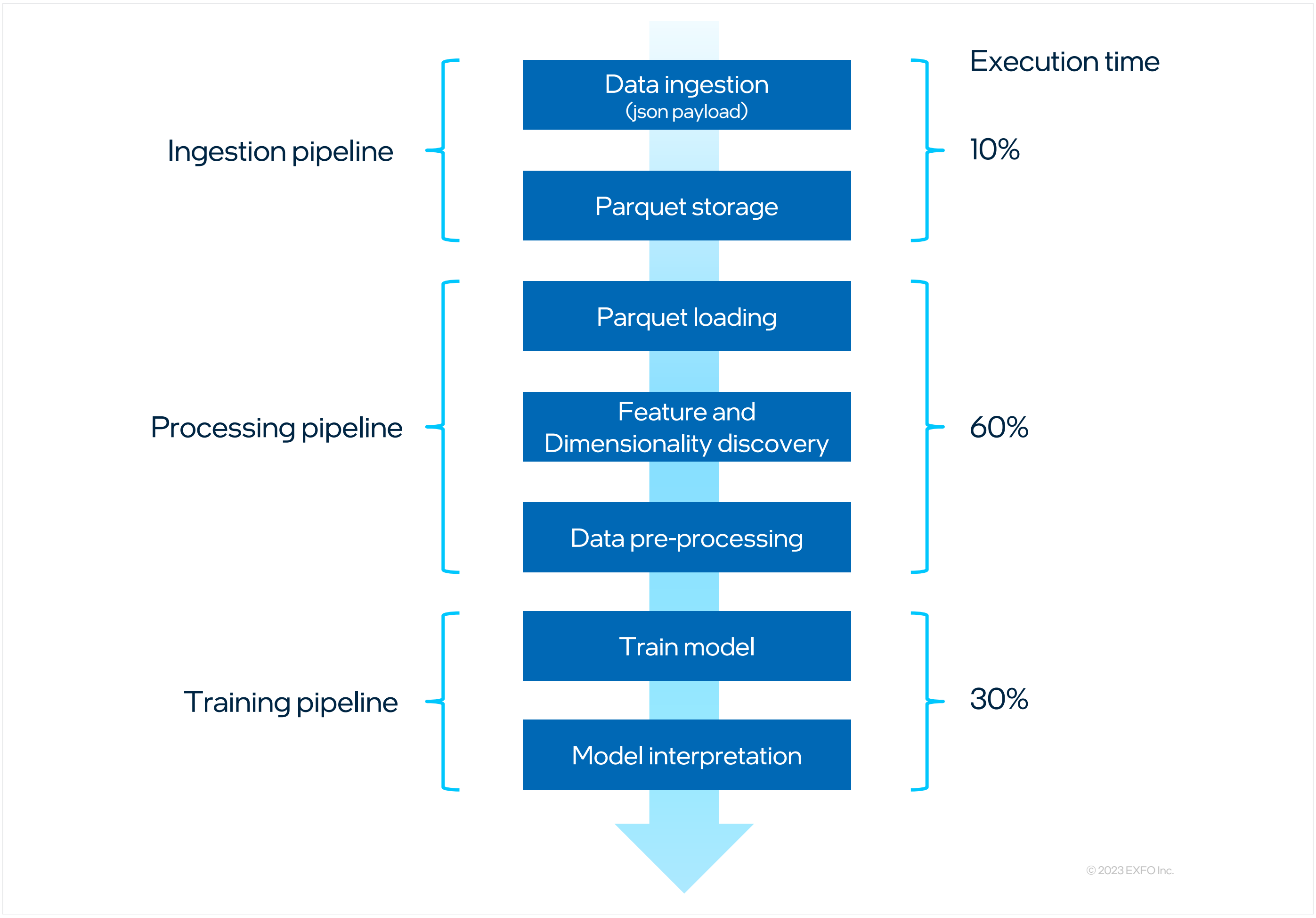


Figure 3. Diagram of AI pipeline stages and their relative time to execute.

Figure 3 shows the steps invoked in the pipeline and their associated percentage of execution time.

Even though this analysis shows that the highest potential performance gains reside in the pre-processing pipeline, the goal was to accelerate the entire AI pipeline. Note that the ingestion and processing pipeline is constantly injected with new data while the training is only triggered a dozen times per day.

Pipeline Optimizations and Benchmarks

Data conversion and compression

In historical performance management contexts, SNMP, XML, and CSV data have been collected at time intervals from appliances and/or EMS systems. The collected data was then typically analyzed through fixed statistical functions. This process aimed to yield key performance indicators (KPIs), typically necessitating a batch processing architecture to collect and process data.

While these traditional data sources continue to be valuable, the integration of various other data sources from decomposed network functions on highly virtualized infrastructure enhances its utility providing new problem diagnosis possibilities for telecom networks. The multiplication of data sources converging in near-real time prompts a shift in architectural requirements toward streaming data processing.

The convergence of multiple data sources not only facilitates more robust analysis but also opens avenues for employing advanced techniques such as machine learning. While data can be continuously ingested and processed in order to continuously train machine learning models, this makes implementation more difficult, costly and harder to scale. Thus, adopting a periodic (or on-demand) training approach makes the implementation more effective.

In the AI pipeline described in this paper, the data is ingested from Kafka as a stream of JSON payloads and then converted and stored periodically as files to disk. Apache Parquet was selected as the storage format due to its ability to load individual columns and its advantages in terms of size, speed, and schema enforcement, making it better suited for high-performance live feed applications.

For the data conversion and storage, the impact of different compression algorithm standards was considered in terms of compression ratio and the time to compress and write back to disk.

The `pyarrow.parquet function pq.write_table()` allows the selection of various compression algorithms, i.e. none, Snappy, GZIP, Brotli, LZ4 and ZSTD:

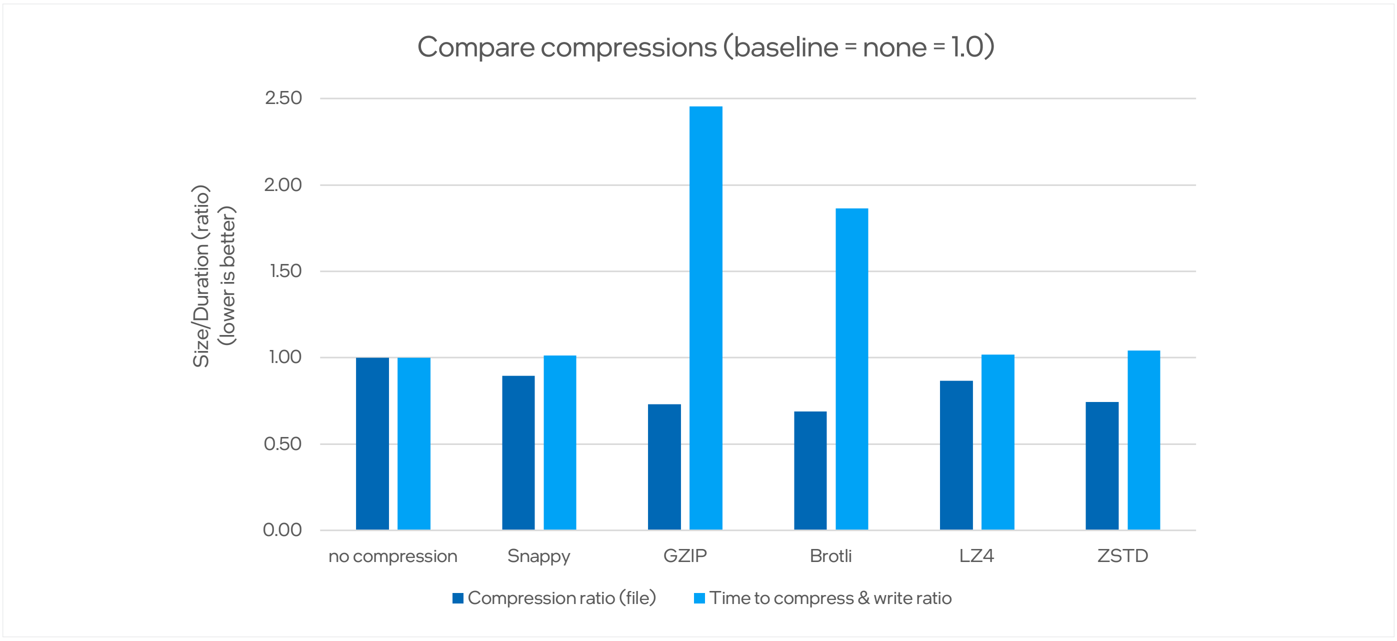


Figure 4. Tests of various compression standards comparing compression ratio and time to compress and write ratio (lower is better).

Compression standard results

For our dataset (mostly double float format), ZSTD is the best fit of compression ratio vs. time to compress and write (see Figure 4). With that, the Parquet file was reduced in size by 17% (vs. Snappy default) with only minimal increase in time to compress (+ 3%).

Overall, this conversion reduces the ingested JSON payload size to Parquet file size from 950 MB to 30 MB (using ZSTD).

Since the time to write varies significantly, another goal of the testing was to understand the impact of a particular compression for loading the compressed file again at the subsequent pre-processing stage. For that the source data files were read and written back using different codecs. After this, the time to load the files was measured.

As opposed to writing, the impact for file loading is negligible. Snappy and ZSTD formatted files took minimal more time as without any compression. GZIP (which was the hardest for writing) only took approximately 10% more time (None 19.6s, Snappy 19.9s, ZSTD 20s, GZIP 22s).

The test conclusion is that the compression codec used primarily impacts the conversion stage but does not significantly affect subsequent data frame loading during pre-processing.

Finally, the workload was run on three AWS EC2 memory optimized instances based on three generations of Intel Xeon Scalable processors. These tests measured the runtime performance against these generations:

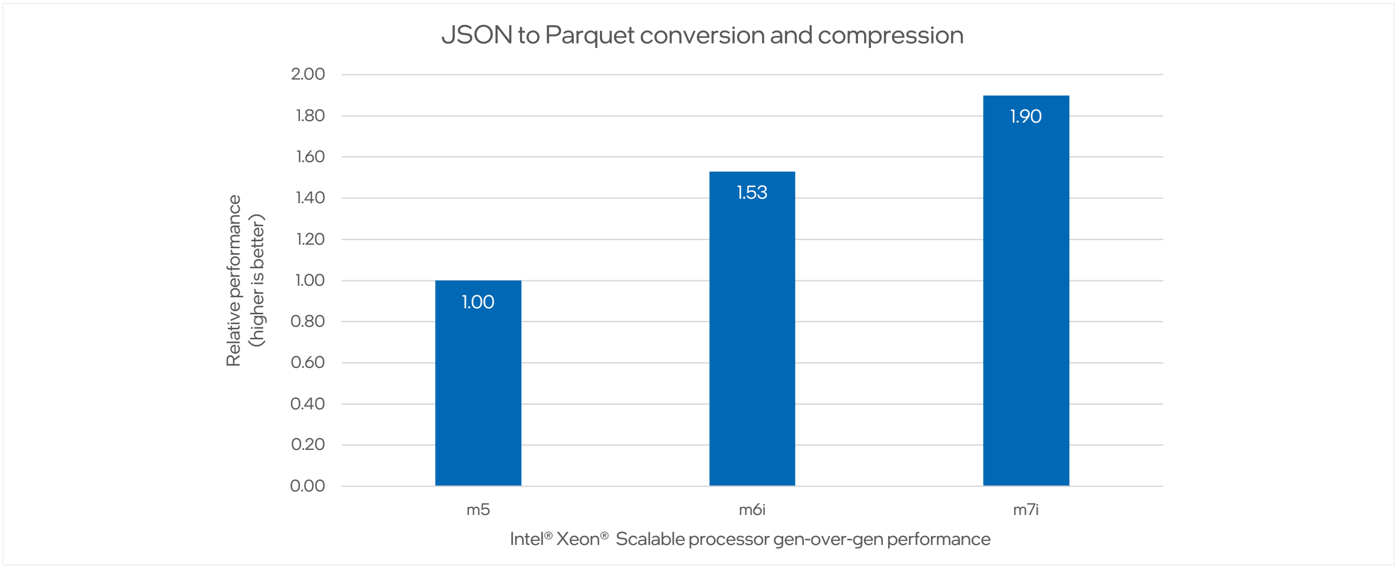


Figure 5. Gen-on-gen comparison: parsing + compressing (ZSTD) and writing on the 2nd Gen Intel® Xeon® Scalable processor, the 3rd Gen Intel® Xeon® Scalable processor and the 4th Gen Intel® Xeon® Scalable processor using ZSTD compression.

The use of the m6i.16xlarge instance[2] is 53% faster than the m5.16xlarge instance[1] to complete JSON/Parquet data conversion and ZSTD compression.

The m7i.16xlarge instance[3] is 90% faster than m5.16xlarge in the time it takes to complete the JSON/Parquet data conversion and ZSTD compression.

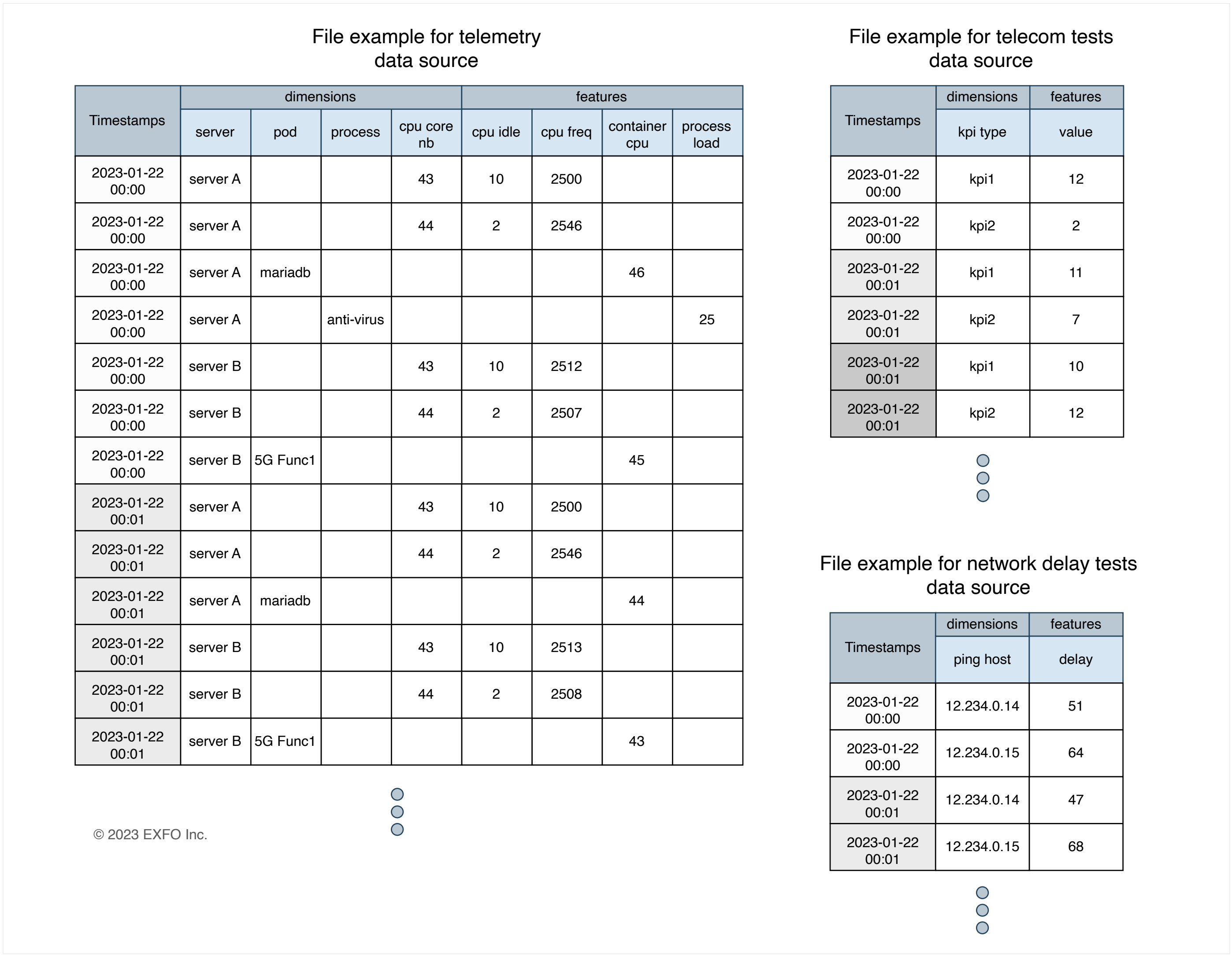
Note that in this example, datasets arrive periodically from subsystems and are being converted on-the-fly as they arrive. Hence, there is no need for massive parallel processing services that are supported by these instance types. However, this can certainly be a requirement in other scenarios, where you may want to batch process a large set of historical data. In this case, you'll need to consider multiprocessing options such as `pool.map()` and also provision for high IOPS disks to make optimal use of the compute resources available.

Important considerations:

- Data source formats vs. ideal file/storage formats (here JSON -> Parquet).
- Best fit compression algorithm (depends on your datatype and formats).
- Minimum / optimal compute resources (# of cores, memory, disk IOPS) depending on amount of data.
  - Note that high IOPS disks have a considerable charge on cloud, e.g. AWS EC2-other.

Pre-processing stage

Now that the ingested raw data has been successfully converted, the Parquet-formatted data files are ready for pre-processing. For this test scenario, real world datasets were used consisting of 7, 14, 28, and 56 days of data. For the largest dataset, this accounts for 24,000 Parquet files with a total file size of 3.78 GB.



File example for telemetry data source

Timestamps	dimensions				features			
	server	pod	process	cpu core nb	cpu idle	cpu freq	container cpu	process load
2023-01-22 00:00	server A			43	10	2500		
2023-01-22 00:00	server A			44	2	2546		
2023-01-22 00:00	server A	mariadb					46	
2023-01-22 00:00	server A		anti-virus					25
2023-01-22 00:00	server B			43	10	2512		
2023-01-22 00:00	server B			44	2	2507		
2023-01-22 00:00	server B	5G Func1					45	
2023-01-22 00:01	server A			43	10	2500		
2023-01-22 00:01	server A			44	2	2546		
2023-01-22 00:01	server A	mariadb					44	
2023-01-22 00:01	server B			43	10	2513		
2023-01-22 00:01	server B			44	2	2508		
2023-01-22 00:01	server B	5G Func1					43	

© 2023 EXFO Inc.

File example for telecom tests data source

Timestamps	dimensions	features
	kpi type	value
2023-01-22 00:00	kpi1	12
2023-01-22 00:00	kpi2	2
2023-01-22 00:01	kpi1	11
2023-01-22 00:01	kpi2	7
2023-01-22 00:01	kpi1	10
2023-01-22 00:01	kpi2	12

File example for network delay tests data source

Timestamps	dimensions	features
	ping host	delay
2023-01-22 00:00	12.234.0.14	51
2023-01-22 00:00	12.234.0.15	64
2023-01-22 00:01	12.234.0.14	47
2023-01-22 00:01	12.234.0.15	68

Figure 6. File inputs used in the pre-processing pipeline.

Challenges

The pre-processing pipeline takes multiple files of data (from different sources) and converts them into a timeseries format before training the model. Because of the changing nature of the data sources, the features and dimensionality aren't constant, thus requiring special care and maintenance.

The goals of the pre-processing pipeline are:

- Merge multiple sources of data.
- Convert the format of data to make it trainable.
- Clean data sources that don't contain enough information to be valuable for the model.
- Limit cardinality of certain fields.
- Reduce the overall dataset (in our example by ~10x).

Figure 6 shows a sample of file inputs used in the pre-processing pipeline.

In this pre-processing step, code optimizations were used to reduce the quite significant time for dataset loading, dimension calculation, flattening and saving.

These functions are all part of the pandas library, such as `pandas.groupby, pandas.concat, etc.` [https://pandas.pydata.org/docs/reference/index.html]

However, most pandas functions are single threaded, so they only execute on a single core. Intel has developed and open-sourced Modin*, which substitutes many pandas functions into multi-threaded workers, allowing them to scale out as dataframes grow in size. The following bullet points show details of optimizations:

- Modifying the Python code can be as simple as importing the Modin library and replacing select function calls with their Modin equivalents.
- It is essential to always install the most recent versions. During our testing, the environment settings were using an outdated version (0.17.0) which, when upgraded to Modin 0.20.1 showed a 10% improvement.

The results came in less than the expected: between two and 10 times, which inspired further analysis of the code structure. When profiling the code using Intel® Granulate™ gProfiler and comparing pandas vs. Modin, it was clear that most functions still defaulted back to pandas.

Solution / Optimizations

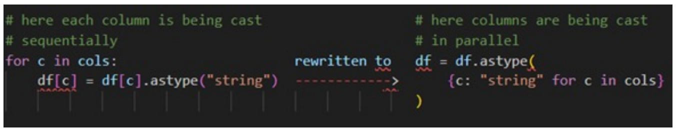
Researchers at EXFO and Intel jointly worked to analyze and refactor the original code, while in parallel also improved some Modin functions (now available in Modin from versions 0.24.1 on):

- **Load dataframes directly in Modin all at once**

  Instead of reading Parquet files one by one at the loading stage, it now passes the directory with the required files directly to '`pd.read _ parquet(dir _ with _ files)`'. This allows efficient reading of all files in one go for parallel processing.

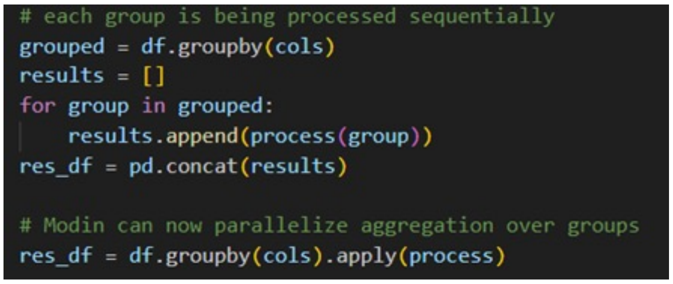
- **Avoid heavy Modin function calls in loops**

  Remove as many for-loops as possible. Loops are processed sequentially and thus cannot be parallelized by Modin. It's better to make one big function call instead of multiple smaller ones. Example:



```
# here each column is being cast                          # here columns are being cast
# sequentially                                            # in parallel
for c in cols:                              rewritten to  df = df.astype(
    df[c] = df[c].astype("string")          ------------>     {c: "string" for c in cols}
                                                          )
```

  Similarly, a sequential groupby aggregation was transformed into a native '`groupby.apply()`' call inside of '`flattenToColumnsNames`' function



```
# each group is being processed sequentially
grouped = df.groupby(cols)
results = []
for group in grouped:
    results.append(process(group))
res_df = pd.concat(results)

# Modin can now parallelize aggregation over groups
res_df = df.groupby(cols).apply(process)
```

- **Provision sufficient memory and disk space**

  For the smallest dataset (7 days) a minimum of 64 GB memory was needed to allow ~32 GB for `Ray /dev/shm tmpfs      31G     0      31G     0%      /dev/shm`

- **Optimize use of multithreading**

  To configure the software to use as many threads as there are datafiles to be loaded.

Changes made in Modin to make all this work can be seen in Appendix A.

Results

By substituting pandas with Modin libraries (including some fixes) up to 4.10 times total performance gain was achieved over the original pandas code (see Figure 7).

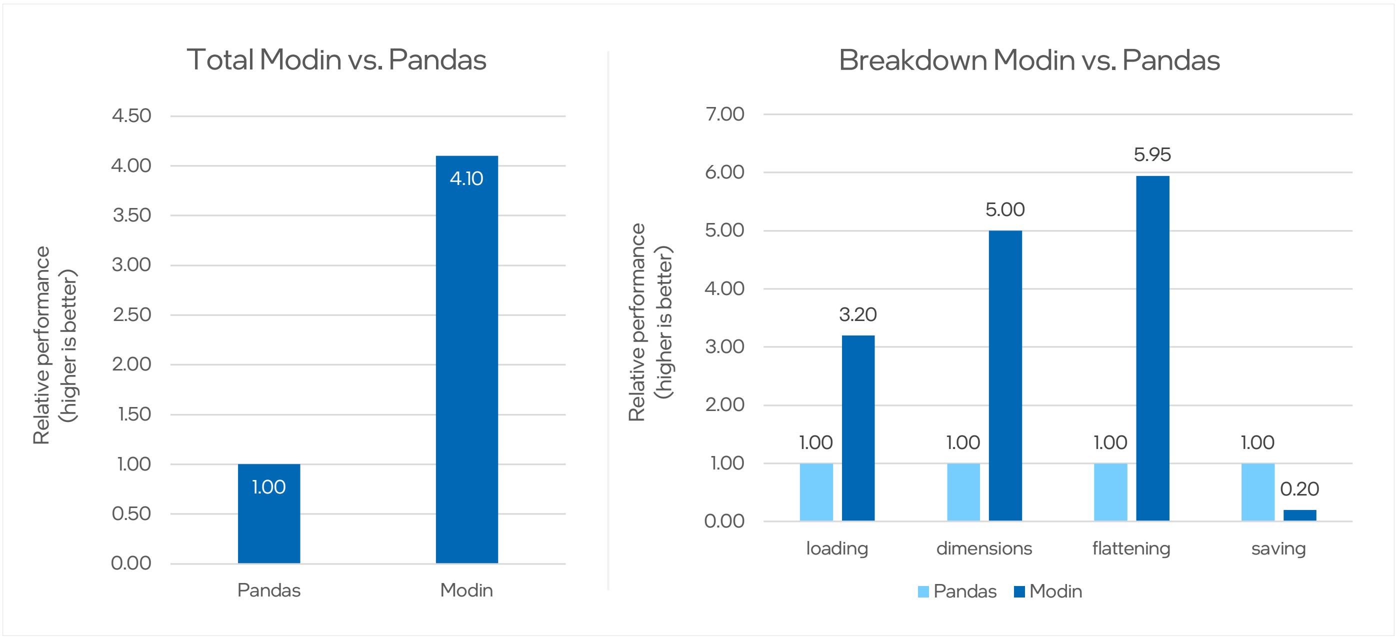


Figure 7. Relative performance to execute pre-processing stage for largest dataset (higher is better).

It is important to understand, that the benefits of Modin's multiprocessing will increase with the number and size of datasets to process. Figure 8 shows how Modin outperforms pandas with larger datasets:

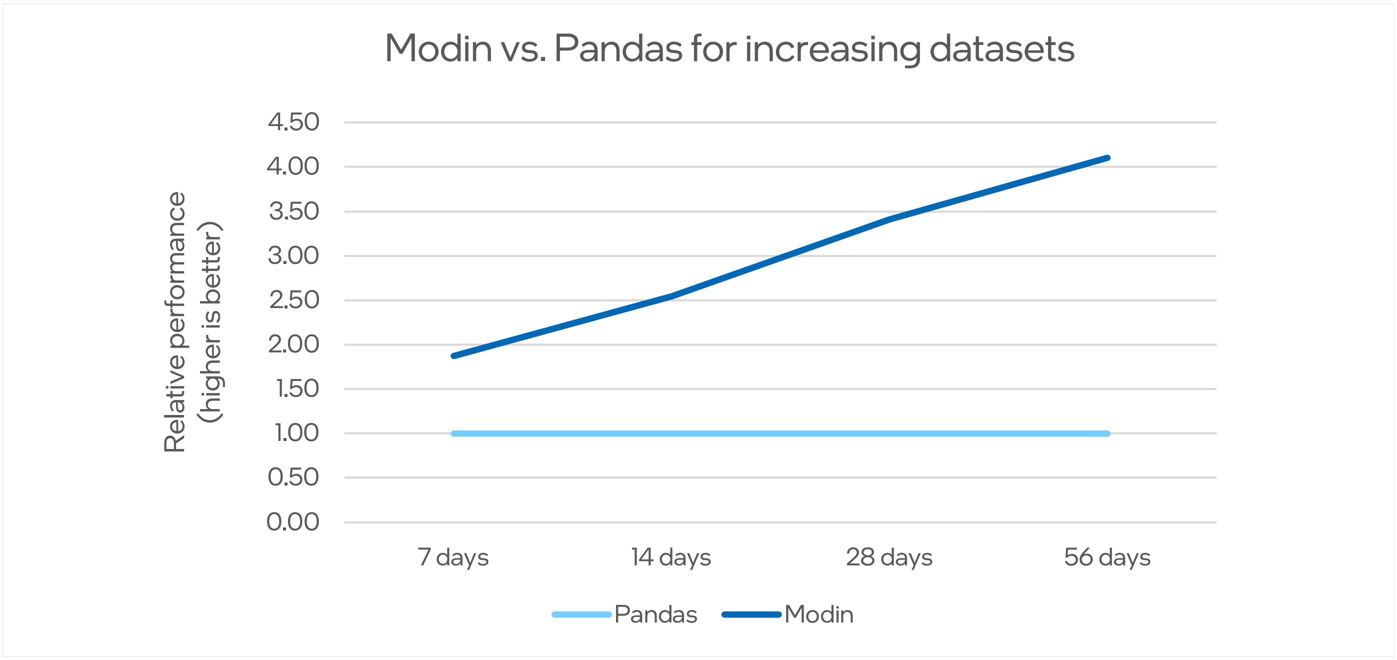


Figure 8. Relative performance to execute pre-processing stage for increasing datasets (higher is better).

Lastly, this workload was run again on three different memory-optimized AWS EC2 instances using the three most recent Intel Xeon Scalable processor generations (see Figure 9). These tests measured the gen-over-gen runtime performance:

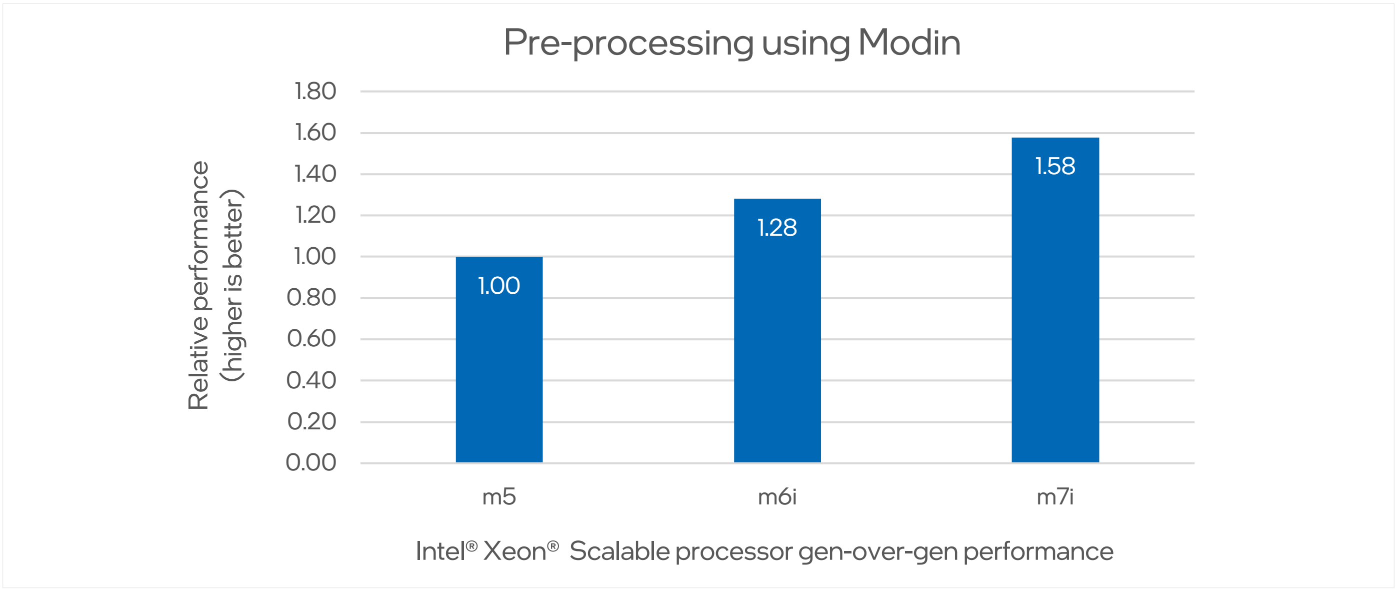


Figure 9. Shows pre-processing stage performance using Modin (higher is better) across cloud instances powered by 2nd Gen Intel® Xeon® Scalable processors, 3rd Gen Intel® Xeon® Scalable processors, 4th Gen Intel® Xeon® Scalable processors (from left to right).

The m6i.16xlarge instance[2] is 28% faster than m5.16xlarge[1] and m7i.16xlarge instance[3] is 58% faster than m5.16xlarge[1] to complete.

The test results show substantial performance improvements on the various generations of Intel Xeon Scalable processors. Considering equal EC2 rates for m5 and m6i and only marginal (5%) increased cost for m7i, the resulting TCO benefit is significant.

Model training and interpretation

The use of machine learning models makes it possible to discover patterns inside multiple sources of data and possibly learn about relationships between them. Because live telecom network systems are very dynamic and always changing, models need to be updated frequently. The process of retraining machine learning models is time consuming and requires thorough validation.

This is why the training and validation process has been automated in a pipeline, which is triggered each time a model needs to be created or updated. As a part of validating these models, SHapley Additive exPlanations (SHAP)[4] was used to interpret the models. The use of SHAP values provides a standardized and consistent method for understanding and comparing model behaviors, thus identifying if the models perform as expected.

Gradient boosting on decision trees is one of the most accurate and efficient machine learning algorithms for classification and regression. There are many implementations of gradient boosting, but the most popular are the Intel® Optimization for XGBoost* and LightGBM frameworks. Although these frameworks provide good performance out of the box, their runtime can still be improved.

Solution / Optimizations

This section describes how we improved EXFO's LightGBM model runtimes up to 26% with the Intel® oneAPI Data Analytics Library (oneDAL).

We can use Model Builders to convert a LightGBM model to oneDAL for faster predictions. Only minimal code changes are required:

```
import daal4py

d4p _ model = daal4py.mb.convert _ model(model)

      shapA = d4p _ model.predict(df, pred _
contribs=True)[:,:-1]

       dfShap = pandas.DataFrame(shapA,
columns=featureList)
```

Other important considerations:

- Setting number of cores/threads to max available (e.g. 16)
- LGBM parameters (or XGBoost equivalents):
  - 'num_leaves': 100
  - 'max_depth': 10
    The max_depth parameter controls the maximum depth of the trees. That is, the maximum number of consecutive splits. Deeper trees can capture more complex patterns in the training data but are also prone to overfitting. Moreover, the oneDAL model performs most efficiently when the maximum depth "max_depth " and number of leaves "max_leaves" are related as $2^{max\_depth} \approxeq$ max_ leaves.

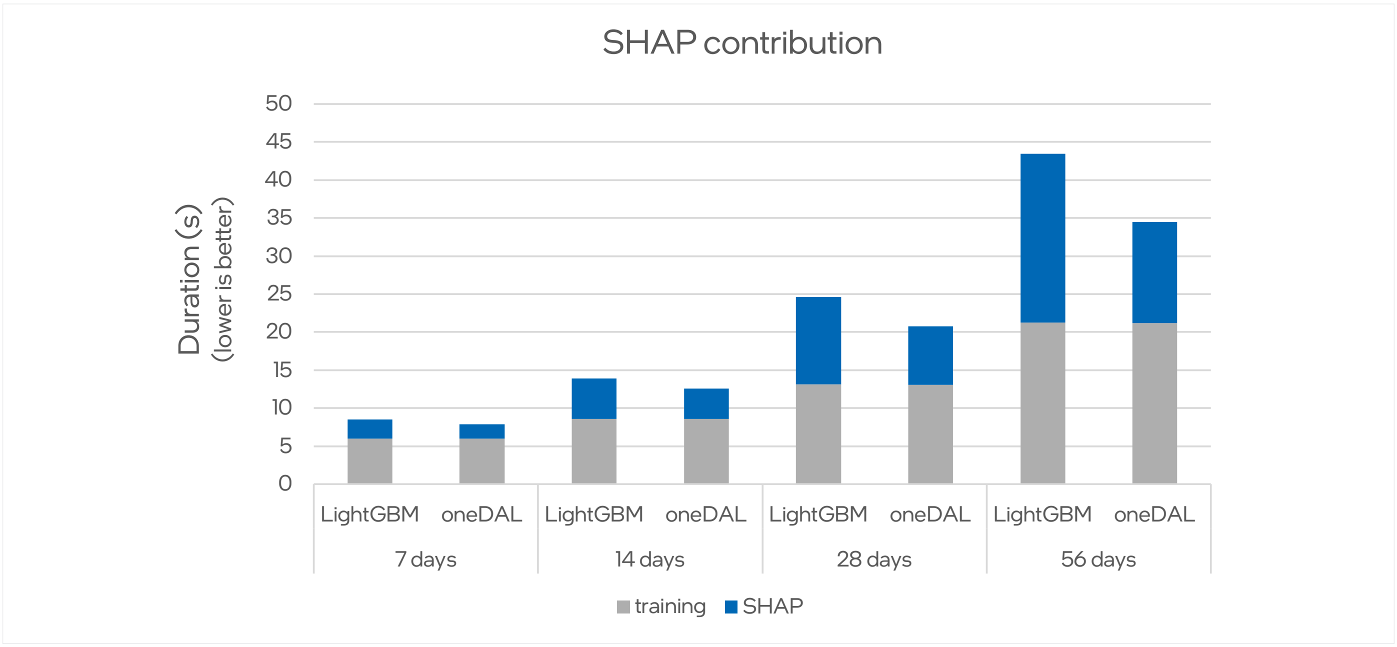


Figure 10. Training and SHAP computation with and without oneDAL optimization using different datasets.

Results

The training is used to discover pattern and relationship between metrics. As part of the training the application is calculating SHAP values to interpret the model outcomes.

For benchmarking, the datasets generated in the previous pre-processing steps were used, i.e. the same 7/14/28 and 56 days of data.

In running the original setup, the SHAP calculation time (see Figure 10) significantly contributes to the overall runtime and grows over-proportionally with larger data.

It was therefore beneficial to optimize the SHAP calculations, which are supported through the oneDAL libraries as part of [`scikit-learn-intelex`]

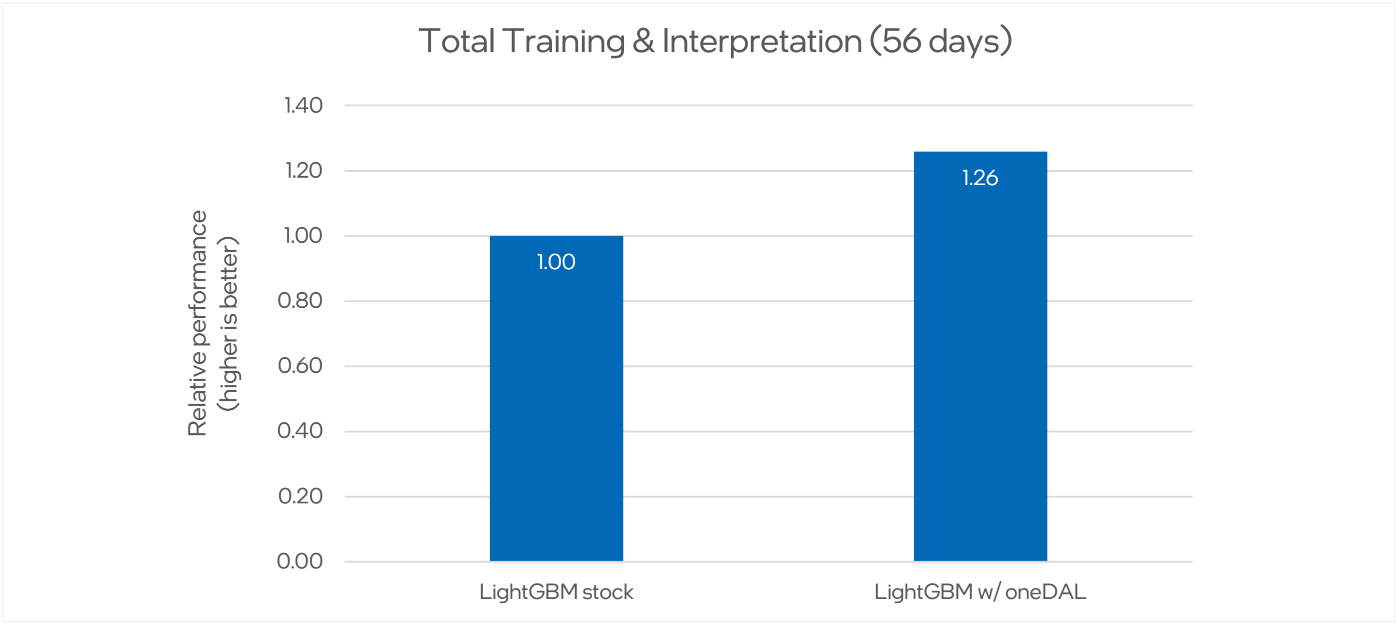


Figure 11. Relative performance gain for 56 days of data on m7i.16xlarge.

Using oneDAL optimized libraries, it is possible to significantly reduce the SHAP portion, leading to a total improvement of 26%.

For even larger datasets or to further reduce run time, the SHAP calculation also nicely scales with the number of threads assigned to the application:

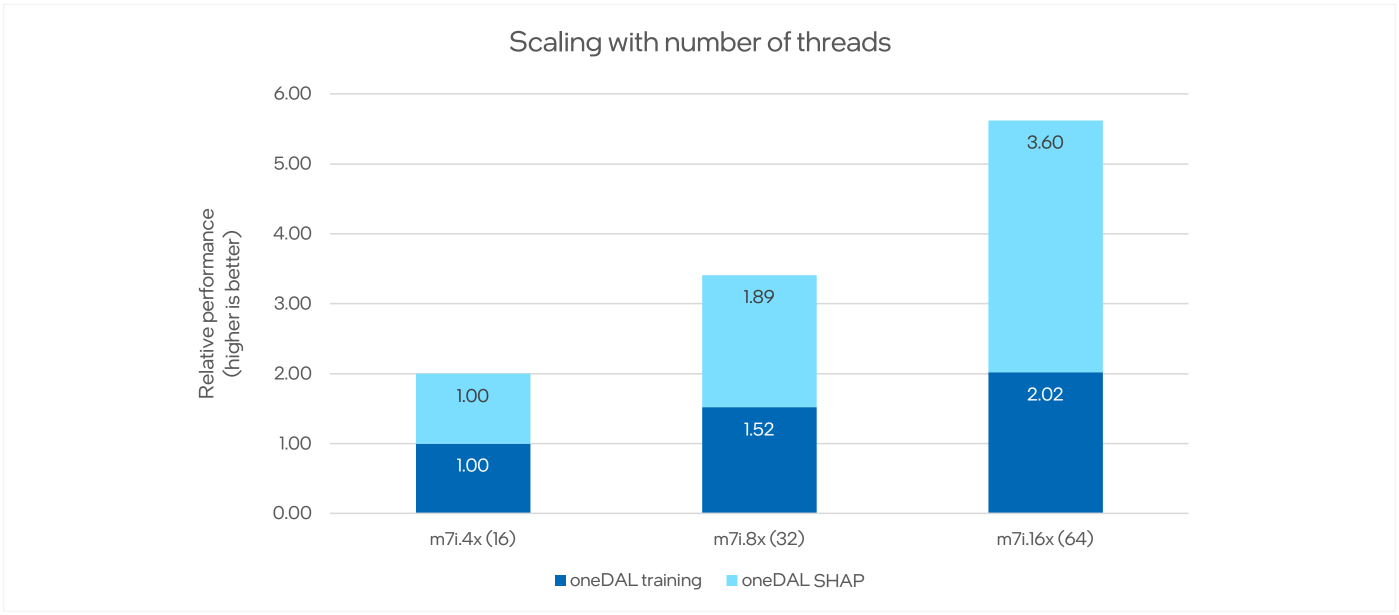


Figure 12. Relative performance for 56 days of data for different instance sizes (number of threads).

The results show how training and the SHAP interpretation task in particular scales with the number of threads using different instance sizes. So, it is possible to easily scale up for faster results or to train on larger datasets.

Make sure you configure your environment accordingly, i.e.:

```
set NB _ CPU _ CORES = int(os.getenv("NB _ CPU _
CORES", 64))
```

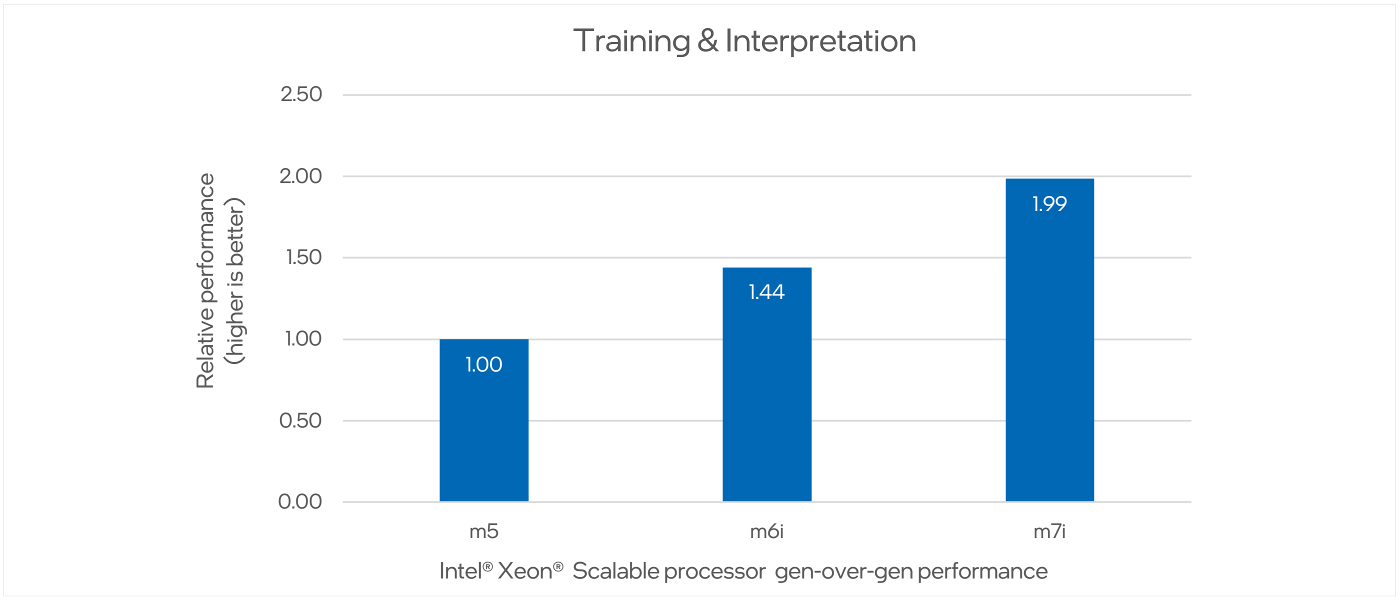


Figure 13. For the largest dataset (56 days) the tests show a gain of 44% and 99% respectively using cloud services based on newer Intel® Xeon® Scalable CPU generations.

And finally, it is again imperative to use latest generation Intel Xeon Scalable Processor family providing the best performance / TCO (see Figure 13).

Conclusion

With this paper we described important considerations for building telco AI analytics applications. This representative implementation of a data pipeline to ingest, convert, pre-process, train and interpret time series data streams can serve as a blueprint for many network analytics use cases.

While data sources and the desired analysis results may vary, the principles remain very similar and require some thorough understanding of the implications of choosing appropriate AI/ML software frameworks and libraries.

Sample diagram:

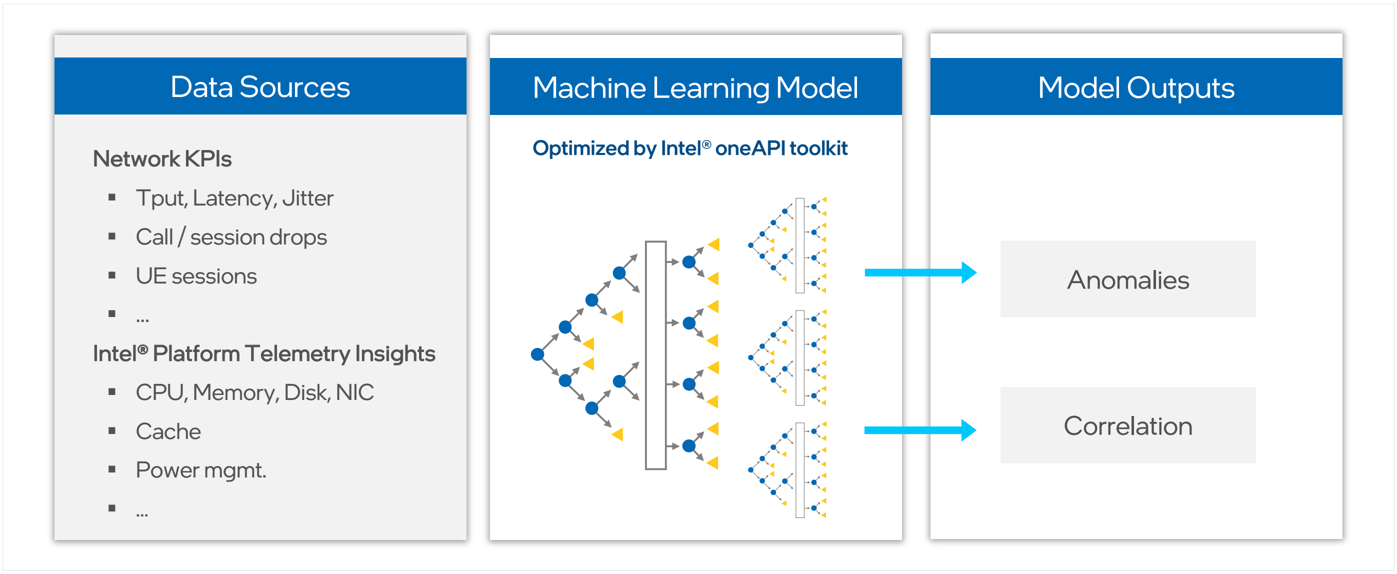


Figure 14. Typical Telco AI/ML pipeline.

For data ingestion and conversion, it is important to evaluate optimal data structures, including compression schemes and storage formats. In our example, we sourced data of approximately 1 TByte to ultimately create compact training data of a few hundreds of MBytes.

The obligatory and, often most compute-intensive, pre-processing stage will likely require multiprocessing capabilities to scale with increasing data volumes to create suitable datasets for training and inference.

Finally, even model training and interpretation can perfectly run on AI optimized CPUs, where optimized libraries will take advantage of specific instruction sets and hardware accelerated functions, such as Intel® Advanced Vector Extensions 512 (Intel® AVX-512), Intel® Advanced Matrix Extensions (Intel® AMX) and others.

oneAPI provides a full suite of libraries supporting all common ML and DL frameworks and can significantly improve the runtime on latest generation Intel Xeon Scalable processors but also GPUs and other accelerators.

By improving runtimes for the entire pipeline, customers will not only benefit from much faster time-to-result and/or scale out for larger data, but they can also reduce the overall footprint and ultimately power consumption of AI workloads. These improvements can result in orders of magnitude of savings considering the frequency of running these different tasks, i.e. permanent, hourly, daily etc.

The following picture summarizes the improvements made across the entire AI/ML pipeline:

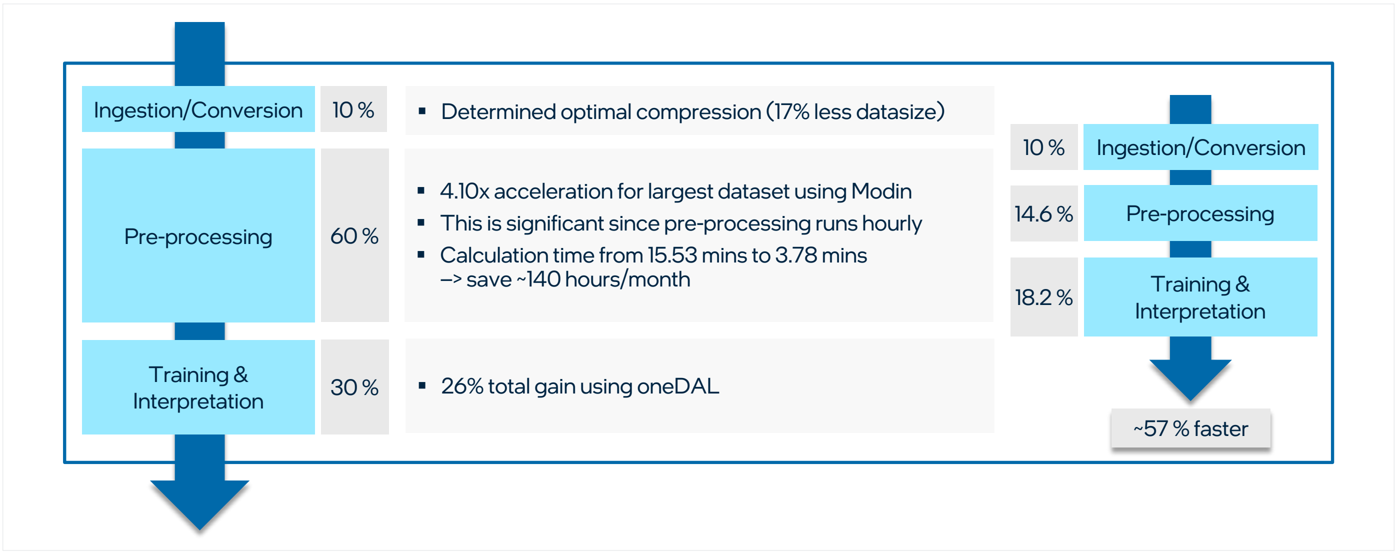

Ingestion/Conversion
10 %
Determined optimal compression (17% less datasize)
Pre-processing
60 %
4.10x acceleration for largest dataset using Modin
This is significant since pre-processing runs hourly
Calculation time from 15.53 mins to 3.78 mins –> save ~140 hours/month
Training & Interpretation
30 %
26% total gain using oneDAL
10 %
Ingestion/Conversion
14.6 %
Pre-processing
18.2 %
Training & Interpretation
~57 % faster

Figure 15. Final pipeline optimizations result in two times the total runtime improvement.

Learn More

EXFO Website

Intel® Network Builders

Intel® Xeon® Scalable Processors

Faster XGBoost*, LightGBM, and CatBoost Inference on the CPU

Intel® Optimization for XGBoost*

Intel® Distribution of Modin*

Video: Full-stack assurance overview and use cases

Disclaimers

Code and data provided by EXFO
All tests conducted by Intel in January 2024

Hardware and Software references[1,2,3]

All tests performed on AWS EC2 instances (region Ohio):

- 1-instance m5.16xlarge: 64 vcpu (2nd Gen Intel Xeon Scalable processor), 256 GB total memory, Ubuntu 22.04.3 LTS, GNU/Linux 6.2.0-1017-aws x86_64, (software/libraries see below)
- 2-instance m6i.16xlarge: 64 vcpu (3rd Gen Intel Xeon Scalable processor), 256 GB total memory, Ubuntu 22.04.3 LTS, GNU/Linux 6.2.0-1017-aws x86_64, (software/libraries see below)
- 3-instance m7i.16xlarge: 64 vcpu (4th Gen Intel Xeon Scalable processor), 256 GB total memory, Ubuntu 22.04.3 LTS, GNU/Linux 6.2.0-1017-aws x86_64, (software/libraries see below)

Additional software environment and versions

Python 3.9.18
Anaconda3-2023.03-1-Linux-x86_64
Numpy 1.26.2
Pandas 2.1.4
Pyarrow 14.0.2
Modin 0.26.0
Ray 2.9.0
XGBoost 2.03
LightGBM 4.2.0
Shap 0.44.0
scikit-learn 1.3.2
scikit-learn-intelex 2024.011
daal 2024.011
daal4py 2024.011

Appendix A: Modin changes

Changes made in Modin to make all this work (these changes were contributed to Modin and expected in release 0.24.0):

1. Initially, the newly introduced '`groupby.apply()`' call has failed since the aggregation function was pretty complex and required group's shape transformation. This was considered as a bug and has been fixed in Modin (modin-project#6506)
2. Next was the problem of low cardinality of the grouping columns. Modin used to process such cases quite inefficiently, which resulted in poor performance. The following improvement being introduced to Modin helped to reduce the total time spent in groupby by ~40% (modin-project#6535) for this workload.
3. Introduced a new more optimal implementation of '`.dropna()`' to Modin which helped to speed-up the 'dim detection' stage ~1.7x (modin-project#6472)
4. Improved performance of the '`.read_parquet()`' method when multiple files were given (there's no PR yet in Modin, so linking the issue modin-project#5723)
5. The workload also revealed some inefficient mechanisms of meta-data handling in Modin, these were fixed by: modin-project#6481, modin-project#6491, modin-project#6525

[4] https://shap.readthedocs.io/en/latest/index.html