**Source: Samsung Electronics Co., Ltd. (Rapporteur)**

**Title: [FS\_AI4Media] Permanent Document v0.2.0**

**Version: 0.2.0**

**Agenda Item: 9.8**

**Document for: Agreement**

# 1 Introduction

During SA4#117-e the New Study Item on “Artificial Intelligence (AI) and Machine Learning (ML) for Media” in S4-220226 was agreed and afterwards approved in by SA#95e in SP-220328.

The objective of this study item are primarily to identify the media service architectures and relevant service flows, model operation configurations, data components including available data formats, and the data traffic characteristics in AI/ML for media related services. Key performance indicators and performance metrics are also identified.

The concrete objectives are as follows:

* List and describe the use cases for media-based AI/ML scenarios, based on those defined in TR 22.874.
* Describe the media service architecture and relevant service flows for the scenarios, identifying for each use case the impacts on the architecture, including any potential gaps with existing 5G media service architectures. Also describe the model operation configurations for each use case, including split AI/ML operations, identifying where certain AI/ML operations occur.
* Identify and document the available data formats and suitable protocols for the exchange of different data components of various AI/ML models, such as model data, metadata, media data, and intermediate data necessary for such model operation configurations. Also investigate the data traffic characteristics of these data components for delivery over 5G system, including whether there are any needs and potentials for data rate reduction.
* Identify and study key performance indicators for such scenarios, based on the initial considerations in TS 22.261, with additional emphasis on the use cases, model operation configurations and data components as identified in earlier objectives, focusing on objective performance metrics considering the KPIs identified.
* Identify potential areas for normative work as the next phase and communicate/align with SA2 as well as other potential 3GPP WGs on relevant aspects related to the study.

# 2 Definition of terms, symbols and abbreviations

## 2.1 Terms

For the purposes of the present document, the terms given in 3GPP TR 21.905 [1] and the following apply. A term defined in the present document takes precedence over the definition of the same term, if any, in 3GPP TR 21.905 [1].

**AI/ML model:** a trained AI/ML model.

**AI/ML model subset:** an elementary element of an AI/ML model that can be inferred independently.

**AI/ML model composition:** the composition of an AI/ML Model into one or more AI/ML model subsets.

**AI/ML model split points:** the points in a DNN AI/ML model where it is split into multiple AI/ML model subsets.

**AI/ML inference endpoint:** an AI/ML endpoint that infers an AI/ML model, or a part of it.

**Split AI/ML model:** an AI/ML model composed of AI/ML subsets that is distributed to, and inferred on different inference endpoints.

# 3 Media-based AI/ML use cases and scenarios

TR 22.874 [1] has identified a set of use cases for AI/ML with the following key operations:

* AI/ML operation splitting between AI/ML endpoints;
* AI/ML model/data distribution and sharing over 5G system;
* Distributed/Federated Learning over 5G system.

These operations have been identified as they require exchange of ML and media data over 5G, and in some cases may have some requirements on the QoS for proper operation.

Of interest to SA4 are the media-based use cases, which are identified in this clause.

## 3.1 Object Recognition in Image and Video

In this set of use cases, images and video streams are processed to identify and recognize objects and extract some metadata, such as bounding boxes, object labels, movement counters, etc.

The specific scenarios that are considered are the following:

* Delivery of trained ML model(s) for object recognition to the UE in 5GS, including the selection of models for different tasks or environments. This scenario involves the key operation of AI/ML model/data distribution.
* Split inference of trained ML model(s) for object recognition between multiple endpoints, typically between the network and UE. Split points may depend on various factors including UE capabilities, network conditions, and model characteristics. Model characteristics include split inference with a task-specific model head running on the UE for object recognition. For example, in one UE, the task is to recognize pedestrians, whereas in another it is to recognize traffic signs. The core of the network model as well as the input image/video are the same, but the tasks (and their required task-specific models) in the UEs are different. This scenario involves both AI/ML operation splitting, and AI/ML model/data distribution.

Distributed online training of image and video recognition models based on input from different UEs. Depending on the configuration of the ML training framework, different data may need to be delivered between the UEs and the network. Typically a shared model in the network is calibrated continuously based on the training results from all UEs. This scenario involves all the three key operations related to AI/ML model distribution, splitting, and distributed/federated learning.

## 3.2 Video Quality Enhancement in Streaming

In this use case, the sender and receiver apply parts of an autoencoder DNN model to enhance the quality of a video stream. This is depicted in figure 3.2-1:



**Figure 3.2-1**

The sender is typically represented by various media functions in the network, which processes the high-fidelity video using the down-scaling part of a pre-trained DNN model to generate a metadata stream that is streamed together with a lower fidelity encoding of the video. The receiver (UE) runs an inference algorithm (e.g. the up-scaling part of DNN model) on the received metadata and video stream to produce a high-quality video for rendering.

The main scenario in this use case is about streaming intermediate model output data from the network for processing on the UE, involving AI/ML data distribution and operation splitting.

## 3.3 Crowd-Sourcing Media Capture

A set of users attending a live concert and capturing the event on their UEs, use a shared (or a set of shared) DNN model(s) to process and improve their respective captured video and/or audio. Audio and video data may be captured in a noisy environment or an environment with poor lighting conditions. Multiple tasks may then be performed on the processed video and/or audio for media content analysis, e.g. to extract lyrics, annotate the video, improve audio and video quality, translate language, anonymize a face, etc.

This use case involves two different scenarios based on either a device inference or a network inference.

### 3.3.1 Device inference

The main scenario is to improve the media capture of each UE by using an up-to-date model adapted to the context event.

This scenario may involve the distribution of multiple models to a large number of UEs in a short period of time. The UEs are heterogeneous, running with different types of operating systems (e.g., Android or iOS), supporting different AI/ML engines/frameworks or having different GPU/CPU/NPU and RAM capabilities available for running the AI/ML service on the UE. This will need the distribution of a huge amount of various AI/ML models adapted to the different device capabilities. Depending on each user’s UE, the UE may request the download of a set of DNN models for device inference.

Moving or changing the environment (localization, energy, processing unit, memory, etc.) may need AI/ML model updates, where the DNN models stored in the network may be adapted or updated during the service.

The AI/ML application may optimize the end-to-end latency (e.g., to achieve latency below 1s) or the expected accuracy level of the inference result (e.g., to achieve image recognition precision of 99%) by modifying the model. The desired latency and/or accuracy level can therefore impact the size of the AI/ML model to be distributed. This can be done by:

* optimizing the model accuracy and latency for on-device execution. The model accuracy and execution latency are known, and the optimization may result in bandwidth saving.
* compressing the model for reducing the bandwidth usage and improving the delivery latency. This may affect the accuracy of the model.

If an uncompressed model is sent, accuracy is not affected but delivery latency would depend on the size of the model and the network bandwidth.

The distribution of the AI/ML models for a large number of UEs at the same time may also need to serve the models from different endpoints (e.g., cloud, edge, or other UEs), and may use several or different communication links (e.g. unicast, multicast or broadcast).

### 3.3.2 Network inference

The main scenario may be the sharing of the input media from multiple sources for network inference, as well as the selection of suitable DNN models according to the UE and/or task.

This scenario requests the UE to upload the media data for network inference. Similarly, to the UE inference, DNN models stored in the network may be adapted or updated during the service for network inferences.

## 3.4 NLP on Speech

This set of use cases covers a wide range of speech processing use cases, e.g. to perform automatic speech recognition, voice translation, voice commands, speech synthesis, etc.

The AI/ML models for NLP are improved with distributed/federated training using multiple UEs. As more users make use of the service, the quality and accuracy of the models improves. The results of the local training of the models by the UEs are shared with the network.

The main scenario here is about UE downloading a partially trained model identified with its training state for local training, and then sharing the results with the network for distributed/federated learning.

# 4 Media service architecture for AI/ML

## 4.1 AI/ML model composition

Figure 4.1-1 depicts an AI/ML model composed of different AI/ML subsets based on various split points. Several compositions of the same AI/ML model are represented with AI/ML subsets (M0, M1), (M’0, M’1), or (M “0, M “1, M “2) with split points highlighted in red. The same AI/ML subset may be used in different compositions depending on the configurations of the model composition (e.g. M’0 and M ’00 according to figure 4.1-1).

In figure 4.1-1, (a) and (b) are examples of AI/ML inference endpoints running an AI/ML model M composed of two subsets M0, M1. A endpoint (network/UE) may run the AI/ML model subset M0 while downloading the other subset M1.

Examples (c) and (d) demonstrate AI/ML split models where M0, M’0 run on the UE while M1, M1’ run on the network respectively.



Figure 4.1-1 AI/ML model composition example

### 4.1.1 Split AI/ML model inference topologies

#### 4.1.1.1 UE as media data source

Figure 4.1.1.1-1 depicts examples of split AI/ML model inference topologies where the UE is the media data source, such as in the use-case of clause 3.1 (object recognition). Assuming that the necessary AI/ML model subsets are already available at each endpoint, figure 4.1.1.1-1 shows the data exchanged between the different split inference endpoints, including input media data, intermediate data and inference result.

This result can be a textual indication of the recognized object, an output score, a bounding box, enhanced media data, etc.



Figure 4.1.1.1-1: Split AI/ML model inference where the UE is the media data source

#### 4.1.1.2 Provider/network as media data source

Figure 4.1.1.2-1 depicts examples of split model topologies where the network or the AI/ML provider ingests the media data, such as in the use-case of clause 3.2 (video quality enhancement).



Figure 4.1.1.2-1: Split AI/ML Model inference where the network/ AI/ML service provider ingests the media data

## 4.2 Architectures and service flows

[

Considering the related use cases as documented in TR 22.874 and also as documented in the latest version of the Permanent Document (S4-220500), we can start from some basic scenarios for consideration of a basic architecture for AI/ML media services.

The basic starting scenarios are:

1. Delivery of a pre-trained AI/ML model from network to UE, typically at the start of an AI media service, but may also require updates during the service. At the most basic level AI/ML models can be delivered as a file (e.g. TensorFlow SavedModel, PDF5, ONNX file, NNEF file etc.) containing all the necessary information required for the UE to perform on device inference using the delivered model. For split scenarios, a (partial) AI model to be used in the UE may be delivered.
2. Split inference of a pre-trained AI/ML model(s) with two further sub-scenarios:
	1. Basic scenario with an inference in the network or in the UE.
	2. Split scenario with inferences between the network and the UE, where the intermediate data output from the network inference (resp. UE inference) is transferred to the UE (resp. network) to be used as the input for UE device inference (resp. network inference). Depending on the characteristics of the intermediate data, such as if the intermediate data is media content data, it may be practical to consider 5GMS architectures, procedures and/or protocols for the streaming delivery of such intermediate media data.

]

### 4.2.1 Complete/Basic AI/ML model distribution

[



**Figure 4.2.1-1: Service architecture for AI/ML model delivery with inference in the UE**

Figure 4.2.1-1 shows a simple service architecture for AI/ML model delivery, as described in scenario 1) of clause 4.2, with an inference of a pre-trained AI/ML model in the UE, as described in scenario 2a) of clause 4.2.

In the network:

* An AI model in the repository is selected for the AI media service by the network application, and sent to the delivery function for delivery to the UE.
* The AI model delivery function sends the AI model data to the UE via the 5GS. This delivery function may also contain functionalities related to QoS requests and monitoring.

In the UE:

* A UE application provides an AI media service using the AI model inference engine and AI model access function.
* The AI model access function receives the AI model data via the 5G system, and sends it to the AI model inference engine.
* The AI model inference engine performs inference by using the input data from the data source (e.g. a camera, or other media source) as the input into the AI model received from the AI model access function. The inference output data is sent to the data destination (e.g. a media player).

Depending on the exact service scenario, AI model updates may be necessary during the service, and different AI model data delivery pipelines may be considered for such purposes.

]

### 4.2.2 Split AI/ML operation

[



**Figure 4.2.2-1: Service architecture for split inference between the network and UE**

Figure 4.2.2-1 shows a simple service architecture for split inferences between the network and the UE, as described in scenario 2b) of clause 4.2.

For the split inference scenario, additional components are required:

In the network:

* An AI model inference engine that receives both the network AI model subset(s), and input data, for network inference.
* An intermediate data delivery function receives the partial inference output (intermediate data) from the network inference engine, and sends it to the UE via the 5GS. This delivery function may also contain functionalities related to QoS requests and monitoring.

In the UE:

* An intermediate data access function receives the intermediate data from the network via the 5GS, and sends it to the UE inference engine for UE inference.
* The final inference output data is sent to the data destination (e.g. a media player).

Extra factors should be considered, including those such as:

* Configuration of the split inference between the network and UE. (e.g. definition and selection of the AI/ML model composition into “UE AI model subset” and “network AI model subset”)
* Resource allocation and management for network inference, including ingestion of network AI model data and media data
* Intermediate data delivery pipelines between the network and UE, in particular considering the use of 5GMS defined pipelines to stream intermediate data that is media content data.
* The functionalities of certain components in figure 1 and figure 2 may overlap, and depending on the use case a combined architecture may also be considered FFS.
* Certain components may also overlap with functions defined in 5GMS, clarifications FFS.

]

### 4.2.3 Distributed/federated learning

# 5 Data components for AI/ML-based media services

## 5.1 Model data

## 5.2 Intermediate data

## 5.3 Media data

## 5.4 Metadata

## 5.5 Existing formats for AI/ML models

### 5.5.6 ONNX format

The Open Neural Network Exchange (ONNX) format [2] is an open specification that was developed to facilitate the exchange of machine learning models between different AI frameworks. ONNX consists of the following components:

* A definition of an extensible computation graph model.
* Definitions of standard data types.
* Definitions of built-in operators.

The ONNX format is built around the Protocol Buffers (Protobuf) open-source cross-platform serialization format that was developed initially by Google.

The ONNX Graph is structured as a list of nodes that form an acyclic graph. Each node of the graph represents one of the built-in operators and its attributes. As an example, a node could be a Convolution operation, and its attributes would contain information regarding the padding and stride that must be used. Each edge of the graph represents input or output data tensors. The top-level ONNX construct is a ‘Model.’, and is represented in protocol buffers as the type onnx.ModelProto. It provides metadata that is necessary for the reader to determine if they are able to process the stored model. Each model must explicitly name the operator sets that it relies on for its functionality. Operator sets defines a set of operators and their versions. An operator is identified through its unique operator type (op\_type), which is a case-sensitive operator name.

Built-in operators include a large list of widely used operators such as the following:

* Math operators such as Abs
* DNN operators such as Conv and LSTM
* Activation operators such Sigmoid and Relu
* Pooling operators such as MaxPool
* Other operators such as error computation and data reformatting operators

The following provides an example of an ONNX model in protobuf format:

|  |
| --- |
| ir\_version: 5producer\_name: "skl2onnx"producer\_version: "1.11"domain: "ai.onnx"model\_version: 0graph { node { input: "X" output: "Y" name: "Pa\_Pad" op\_type: "Pad" attribute { name: "mode" s: "constant" type: STRING } attribute { name: "pads" ints: 0 ints: 1 ints: 0 ints: 1 type: INTS } attribute { name: "value" f: 1.5 type: FLOAT } domain: "" } name: "OnnxPad" input { name: "X" type { tensor\_type { elem\_type: 1 shape { dim { } dim { dim\_value: 2 } } } } } output { name: "Y" type { tensor\_type { elem\_type: 1 shape { dim { } dim { dim\_value: 4 } } } } }}opset\_import { domain: "" version: 10} |

### 5.5.6 NNEF format

The Neural Network Exchange Format (NNEF) [3] is a Khronos developed standard that defines a data format for facilitating the exchange of trained network models. The NNEF format enables the encapsulation of both the structure of the neural network model as well as the associated data. NNEF stores the data in structures that are independent of the training environment that was used for training the network, which will facilitate its consumption on any execution platform. NNEF offers itself as an intermediary between deep learning frameworks, which export into NNEF, and neural network accelerator libraries, which will import and compile the NNEF model for hardware-optimized inference.

The NNEF container consists of the following files:

* a textual file that describes the structure of the neural network
* a binary data file for each variable tensor. These files are structured hierarchically into sub-folders associated with the corresponding operation. Each tensor may have different representations, each matching a different quantized version.
* a quantization file that contains details about the quantization algorithm that is used for quantizing the exported tensors.

The NNEF network structure is described through a computational graph. The computational graph is a directed graph. The nodes of the graph may be data nodes or operation nodes. A directed edge from a data node to an operation node indicates the data is input to the operation. A directed edge from an operation node to a data node indicates the data node is an output.

Data nodes are tensors of different ranks and shapes and may be external, constant, variable, or intermediate/regular tensors. external, constant, and variable tensors all provide an explicit declaration of their shapes. Other tensors shapes will be determined based on the input and operation that is applied to them to produce that tensor. This is commonly known as shape propagation.

The NNEF operation nodes may have attributes that describe the exact computation that needs to be performed. Operations may be composed together to produce more compound operations. Primitive operations are operations that cannot be broken down into simpler operations.

The following is an excerpt from an NNEF graph representation of the VGG-16 network model:

|  |
| --- |
| version 1.0;graph VGG\_ILSVRC\_16\_layers(data) -> (prob){ variable\_15 = variable<scalar>(label = 'conv4\_1\_blob2', shape = [1, 512]); variable\_14 = variable<scalar>(label = 'conv4\_1\_blob1', shape = [512, 256, 3, 3]); variable\_13 = variable<scalar>(label = 'conv3\_3\_blob2', shape = [1, 256]); variable\_31 = variable<scalar>(label = 'fc8\_blob2', shape = [1, 1000]); variable\_30 = variable<scalar>(label = 'fc8\_blob1', shape = [1000, 4096]); variable\_29 = variable<scalar>(label = 'fc7\_blob2', shape = [1, 4096]); variable\_28 = variable<scalar>(label = 'fc7\_blob1', shape = [4096, 4096]); variable\_27 = variable<scalar>(label = 'fc6\_blob2', shape = [1, 4096]); variable\_26 = variable<scalar>(label = 'fc6\_blob1', shape = [4096, 25088]); variable\_25 = variable<scalar>(label = 'conv5\_3\_blob2', shape = [1, 512]); variable\_24 = variable<scalar>(label = 'conv5\_3\_blob1', shape = [512, 512, 3, 3]); variable\_23 = variable<scalar>(label = 'conv5\_2\_blob2', shape = [1, 512]); variable\_22 = variable<scalar>(label = 'conv5\_2\_blob1', shape = [512, 512, 3, 3]); variable\_21 = variable<scalar>(label = 'conv5\_1\_blob2', shape = [1, 512]); variable\_20 = variable<scalar>(label = 'conv5\_1\_blob1', shape = [512, 512, 3, 3]); variable\_19 = variable<scalar>(label = 'conv4\_3\_blob2', shape = [1, 512]); variable\_18 = variable<scalar>(label = 'conv4\_3\_blob1', shape = [512, 512, 3, 3]); variable\_17 = variable<scalar>(label = 'conv4\_2\_blob2', shape = [1, 512]); variable\_16 = variable<scalar>(label = 'conv4\_2\_blob1', shape = [512, 512, 3, 3]); variable\_12 = variable<scalar>(label = 'conv3\_3\_blob1', shape = [256, 256, 3, 3]); variable\_10 = variable<scalar>(label = 'conv3\_2\_blob1', shape = [256, 256, 3, 3]); variable\_9 = variable<scalar>(label = 'conv3\_1\_blob2', shape = [1, 256]); variable\_8 = variable<scalar>(label = 'conv3\_1\_blob1', shape = [256, 128, 3, 3]); variable\_6 = variable<scalar>(label = 'conv2\_2\_blob1', shape = [128, 128, 3, 3]); variable\_11 = variable<scalar>(label = 'conv3\_2\_blob2', shape = [1, 256]); variable\_5 = variable<scalar>(label = 'conv2\_1\_blob2', shape = [1, 128]); variable\_4 = variable<scalar>(label = 'conv2\_1\_blob1', shape = [128, 64, 3, 3]); variable\_2 = variable<scalar>(label = 'conv1\_2\_blob1', shape = [64, 64, 3, 3]); variable\_1 = variable<scalar>(label = 'conv1\_1\_blob2', shape = [1, 64]); variable\_7 = variable<scalar>(label = 'conv2\_2\_blob2', shape = [1, 128]); variable = variable<scalar>(label = 'conv1\_1\_blob1', shape = [64, 3, 3, 3]); variable\_3 = variable<scalar>(label = 'conv1\_2\_blob2', shape = [1, 64]); data = external<scalar>(shape = [10, 3, 224, 224]); conv = conv(data, variable, variable\_1, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu = relu(conv); conv\_1 = conv(relu, variable\_2, variable\_3, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_1 = relu(conv\_1); max\_pool = max\_pool(relu\_1, border = 'ignore', padding = [(0, 0), (0, 0), (0, 0), (0, 0)], size = [1, 1, 2, 2], stride = [1, 1, 2, 2]); conv\_2 = conv(max\_pool, variable\_4, variable\_5, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_2 = relu(conv\_2); conv\_3 = conv(relu\_2, variable\_6, variable\_7, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_3 = relu(conv\_3); max\_pool\_1 = max\_pool(relu\_3, border = 'ignore', padding = [(0, 0), (0, 0), (0, 0), (0, 0)], size = [1, 1, 2, 2], stride = [1, 1, 2, 2]); conv\_4 = conv(max\_pool\_1, variable\_8, variable\_9, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_4 = relu(conv\_4); conv\_5 = conv(relu\_4, variable\_10, variable\_11, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_5 = relu(conv\_5); conv\_6 = conv(relu\_5, variable\_12, variable\_13, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_6 = relu(conv\_6); max\_pool\_2 = max\_pool(relu\_6, border = 'ignore', padding = [(0, 0), (0, 0), (0, 0), (0, 0)], size = [1, 1, 2, 2], stride = [1, 1, 2, 2]); conv\_7 = conv(max\_pool\_2, variable\_14, variable\_15, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_7 = relu(conv\_7); conv\_8 = conv(relu\_7, variable\_16, variable\_17, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_8 = relu(conv\_8); conv\_9 = conv(relu\_8, variable\_18, variable\_19, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_9 = relu(conv\_9); max\_pool\_3 = max\_pool(relu\_9, border = 'ignore', padding = [(0, 0), (0, 0), (0, 0), (0, 0)], size = [1, 1, 2, 2], stride = [1, 1, 2, 2]); conv\_10 = conv(max\_pool\_3, variable\_20, variable\_21, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_10 = relu(conv\_10); conv\_11 = conv(relu\_10, variable\_22, variable\_23, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_11 = relu(conv\_11); conv\_12 = conv(relu\_11, variable\_24, variable\_25, border = 'constant', dilation = [1, 1], groups = 1, padding = [(1, 1), (1, 1)], stride = [1, 1]); relu\_12 = relu(conv\_12); max\_pool\_4 = max\_pool(relu\_12, border = 'ignore', padding = [(0, 0), (0, 0), (0, 0), (0, 0)], size = [1, 1, 2, 2], stride = [1, 1, 2, 2]); reshape = reshape(max\_pool\_4, shape = [10, -1]); linear = linear(reshape, variable\_26, variable\_27); relu\_13 = relu(linear); linear\_1 = linear(relu\_13, variable\_28, variable\_29); relu\_14 = relu(linear\_1); linear\_2 = linear(relu\_14, variable\_30, variable\_31); prob = softmax(linear\_2, axes = [1]);} |

# 6 Traffic characteristics

## 6.1 Complete/Basic AI/ML model distribution

## 6.2 Split AI/ML operation

## 6.3 Distributed/federated learning

# 7 KPIs

# 8 References

[1] 3GPP TR 22.874, Study on traffic characteristics and performance requirements for AI/ML model transfer in 5GS

[2] Open Neural Network Exchange (ONNX), <https://onnx.ai>

[3] The Khronos NNEF Working Group, “Neural Network Exchange Format”, https://www.khronos.org/registry/NNEF/specs/1.0/nnef-1.0.5.html