NVIDIA TensorRT Migration Guide

Release 6.x and 7.x for NVIDIA DriveOS

NVIDIA Corporation

Jun 25, 2026

Contents

This guide covers the API and tooling changes required to migrate NVIDIA TensorRT workloads on NVIDIA DriveOS 6.x and 7.x. It addresses two migration paths: TensorRT 10.x to 11.x (primary) and TensorRT 8.x to 10.x (appendix).

If you are unfamiliar with these changes, refer to the TensorRT sample code for clarification.

Chapter 1. Migration Guide Revision History

Table 1.1: Document Revision History

Date	Summary of Change
March 31, 2026	Initial draft for NVIDIA TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5
April 1, 2026	Start of review
June 21, 2026	End of review
June 22, 2026	Approval review

Table 1.2: Document Changes

Date	Summary of Change
April 10, 2026	Reorganized the TensorRT 8.x to 10.x migration material under the appendix (with accompanying content updates) to continue support for those still on TensorRT 8.x. Added the new TensorRT 10.x to 11.x migration guide, which includes the C++ API, Python API, trtexec, and Safety Runtime migration chapters.
April 20, 2026	Added additional chapters: *Migrating IEngineInspector Usage from TensorRT 10.x to 11.x*, *Migrating TensorRT from 10.x to 11.x on NVIDIA DriveOS*, and *Migrating TensorRT from 10.x to 11.x on Jetson/JetPack*. Updated the *trtexec migration page* with additional deprecated flags (`--useCudaGraph`, `--useSpinWait`, `--noDataTransfers`, `--separateProfileRun`) and their replacements.
May 11, 2026	Removed a stale “NVIDIA TensorRT 11.x is currently in development” attention block from the **TensorRT 10.x to 11.x** overview, **C++ API**, **trtexec**, **Python API**, and **Safety Runtime** migration chapters.

1.1. How to Use This Guide

Choose the starting point that matches your current TensorRT version:

Migrating from TensorRT 10.x to 11.x
Start here if you are already on TensorRT 10.x and need to prepare for TensorRT 11.x.

Appendix: Migrating from TensorRT 8.x to 10.x
Start here if you are still on TensorRT 8.x. After completing the 8.x-to-10.x migration, continue to the 10.x-to-11.x section.

Chapter 2. Migrating from TensorRT 10.x to 11.x

TensorRT 11.x introduces several breaking changes that affect how you build engines, run inference, and use plugins. The sections below summarize the key changes. For language-specific and tool-specific migration details, select the page that matches your integration:

Python Integration
: Migrate Python inference and engine-build code, and review Python APIs removed in 11.x with their replacements.

C++ Integration
: Migrate C++ inference and engine-build code, and review C++ APIs and plugins removed in 11.x with their replacements.

trtexec Command-Line Usage
: Migrate `trtexec` command lines, and review removed or deprecated flag replacements.

Safety Runtime
: Migrate safety runtime code paths, including build-time safety scope validation and kernel checker changes.

NVIDIA DriveOS
: Platform-specific migration guidance for DRIVE Thor and DRIVE Orin, including version compatibility and DLA considerations.

Jetson / JetPack
: Platform-specific migration guidance for Jetson Orin and Jetson Thor, including JetPack dependencies and cross-compilation.

IEngineInspector JSON Output
: Migrate code that parses `IEngineInspector` JSON output, including the replacement of `Bindings` with `I/O Tensors` and the split of the `Format/DataType` field.

2.1. Strongly Typed Networks Are Now Required

TensorRT 11.x removes all *weak typing* APIs. In TensorRT 10.x, you could enable reduced-precision types on the model level by setting builder flags such as `BuilderFlag::kFP16` or `BuilderFlag::kINT8`, and TensorRT would automatically consider lower-precision kernels. While this made it simple to get higher throughput, it could compromise accuracy and did not provide adequate control to model authors.

Similarly, implicit quantization is not supported in TensorRT 11.x. To quantize a model, you must add quantize and dequantize layers to the network.

The most straightforward way to get results similar to using the `trtexec --fp16` flag or `BuilderFlag::kFP16` is to use ModelOpt's AutoCast feature on your model. Refer to the ModelOpt AutoCast documentation for more information; however, for a basic conversion from FP32 layers to FP16, run:

```
python -m modelopt.onnx.autocast --onnx_path model.onnx --output_path model_
↪fp16.onnx
```

For quantizing a network, run:

```
python -m modelopt.onnx.quantization --onnx_path model.onnx --calibration_data
↪data.npz
```

For more control or other advanced quantization topics (such as quantization-aware training), refer to the ModelOpt Quantization documentation.

You can also manually control a layer's I/O types by using `INetworkDefinition::addCast` or adding `addCast` nodes to the ONNX graph.

Similarly, you can apply quantization to a layer in your network using the `INetworkDefinition::addQuantize` and `INetworkDefinition::addDequantize` APIs, or by adding `QuantizeLinear` and `DequantizeLinear` nodes to the ONNX graph manually. This approach works, but determining the layers that must run at high precision and calibrating the quantization scales for optimal accuracy can be challenging. For most models, use ModelOpt to apply quantization effectively.

2.2. V3 Is the Recommended Plugin Interface

The V2 plugin interfaces (`IPluginV2`, `IPluginV2Ext`, `IPluginV2IOExt`, `IPluginV2DynamicExt`, `IPluginCreator`, `IPluginV2Layer`, and `INetworkDefinition::addPluginV2()`) remain present in TensorRT 11.0 but are deprecated; new plugins should use `IPluginV3` with `IPluginCreatorV3One`, and existing V2 plugins should be migrated. TensorRT 11.x removes the following deprecated methods on the V2 interfaces:

- `IPluginV2Ext::isOutputBroadcastAcrossBatch` and `IPluginV2Ext::canBroadcastInputAcrossBatch` (also inherited by `IPluginV2DynamicExt`) - implicit-batch broadcast helpers; no replacement is needed because implicit batch is unsupported since 10.0.
- The legacy `IPluginRegistry` overloads that took `IPluginCreator`: `registerCreator(IPluginCreator&, ...)`, `deregisterCreator(IPluginCreator const&)`, `getPluginCreator(...)`, and `getPluginCreatorList(...)`. Use the `IPluginCreatorInterface`-typed overloads instead.

Key differences between V2 and V3:

- **Capability-based design**: `IPluginV3` uses separate capability interfaces (`IPluginV3OneCore`, `IPluginV3OneBuild`, `IPluginV3OneRuntime`) instead of a single monolithic class.
- **Data-dependent output shapes**: V3 supports declaring size tensors using `IExprBuilder::declareSizeTensor()`, enabling plugins with data-dependent output shapes.
- **Shape inputs**: `addPluginV3()` accepts both data inputs and shape inputs, unlike `addPluginV2()`, which only accepts data inputs.
- **Phase-aware creation**: `IPluginCreatorV3One::createPlugin()` receives a `TensorRTPhase` parameter, allowing different behavior during build vs. runtime.

2.3. V2 API Methods Replaced with Updated Versions

Several API methods have been replaced with updated versions that use `int64_t` instead of `size_t`, accept additional parameters, or support asynchronous operation:

Category	Removed	Replacement
Device memory size	`getDeviceMemorySize()`	`getDeviceMemorySizeV2()` (returns `int64_t`)
Device memory (profile)	`getDeviceMemorySizeForProfile()`	`getDeviceMemorySizeForProfileV2()` (returns `int64_t`)
Weight streaming budget	`setWeightStreamingBudget()` / `getWeightStreamingBudget()`	`setWeightStreamingBudgetV2()` / `getWeightStreamingBudgetV2()`
Profile tensor values	`getProfileTensorValues()`	`getProfileTensorValuesV2()` (returns `int64_t` values)
Deserialization	`deserializeCudaEngine(IStreamReader&)`	`deserializeCudaEngine(IStreamReaderV2&)`

2.4. Other Removals

- **Tactic sources**: The `kCUBLAS`, `kCUBLAS_LT`, and `kCUDNN` values of the `TacticSource` enum have been removed. TensorRT no longer uses these external libraries for core operations.
- **Implicit batch**: `allInputShapesSpecified()` has been removed.
- **Built-in plugins removed**: 16 plugins deprecated before TensorRT 10 have been removed, including the NMS family (`BatchedNMS_TRT`, `BatchedNMSDynamic_TRT`, `EfficientNMS_ONNX_TRT`, `NMS_TRT`, `NMSDynamic_TRT`), activations (`Clip_TRT`, `CustomGeluPluginDynamic`, `LReLU_TRT`), and `BatchTilePlugin_TRT`, `CoordConvAC`, `Normalize_TRT`, `Proposal`, `SingleStepLSTMPlugin`, `SpecialSlice_TRT`, `Split`. Refer to *Removed C++ Plugins and Replacements* for replacements.
- **BERT plugin family deprecated**: The OSS BERT plugin classes (`bertQKVToContextPlugin` / `CustomQKVToContextPluginDynamic` versions 1 through 4, and `CustomEmbLayerNormPluginDynamic`) are deprecated in 11.0.0 and scheduled for removal in a future release. Use the native attention path (`addAttentionV2`) for QKV-to-context, and standard `IGatherLayer` + `addNormalizationV2()` for embedding + layer normalization. Refer to *Deprecated BERT Plugins* for per-plugin replacements.
- **ONNX parser**: `supportsModel()` and `parseWithWeightDescriptors()` have been removed.
- **Layer overloads**: Older overloads of `addTopK`, `addNonZero`, and `addNMS` that did not accept an `indicesType` parameter have been removed. Use the versions that accept a `DataType` parameter for the indices output type.
- **Multi-Device preview flag**: The `PreviewFeature::kMULTIDEVICE_RUNTIME_10_16` value has been removed. `setPreviewFeature()` is no longer required to access Multi-Device Inference features.

Note

If you are migrating from TensorRT 8.x, refer to the *appendix* first, then return here for the 10.x-to-11.x changes.

Chapter 3. Migrating Python Code from TensorRT 10.x to 11.x

This page describes how to update Python code when you migrate from TensorRT 10.x to 11.x: paired examples for strong typing, explicit quantization, plugin migration, and updated runtime APIs, followed by lists of Python APIs added and removed in 11.x.

Note

The Python API is not supported on QNX.

Warning

Engine files are executable artifacts that contain compiled CUDA tactics. The `deserialize_cuda_engine` examples in *Migrating from Weak Typing to Strong Typing* assume the engine file was built by you on a trusted host or received over a trusted, authenticated channel. Never deserialize an engine file from an untrusted source. Refer to Engine Deserialization and the Trust Boundary for the full trust-boundary model.

3.1. Migrating from Weak Typing to Strong Typing

TensorRT 11.x removes all precision-enabling builder flags such as `BuilderFlag.FP16` and `BuilderFlag.INT8`. Use ModelOpt AutoCast to convert your ONNX model to mixed precision before building.

3.1.1. Before (TensorRT 10.x)

```
import tensorrt as trt

logger = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
network = builder.create_network()
config = builder.create_builder_config()
```

(continues on next page)

(continued from previous page)

```
# Weak typing: TensorRT automatically considers FP16 kernels
config.set_flag(trt.BuilderFlag.FP16)

parser = trt.OnnxParser(network, logger)
with open("model.onnx", "rb") as f:
    parser.parse(f.read())

engine_bytes = builder.build_serialized_network(network, config)
```

In TensorRT 11.x, the `BuilderFlag.FP16` flag has been removed along with all other weak typing flags. Use ModelOpt AutoCast to convert your FP32 ONNX model to mixed precision before building, then build with a strongly typed network.

3.1.2. After (TensorRT 11.x)

```
import tensorrt as trt
import modelopt.onnx.autocast as autocast
import onnx

# Step 1: Convert FP32 model to mixed-precision FP16 using ModelOpt AutoCast
converted_model = autocast.convert_to_mixed_precision(
    onnx_path="model.onnx",
    low_precision_type="fp16",
    keep_io_types=True,
)
onnx.save(converted_model, "model_fp16.onnx")

# Step 2: Build with strongly typed network (strong typing is always on in 11.x)
logger = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
network = builder.create_network()
config = builder.create_builder_config()

parser = trt.OnnxParser(network, logger)
with open("model_fp16.onnx", "rb") as f:
    parser.parse(f.read())

engine_bytes = builder.build_serialized_network(network, config)
```

3.1.3. Summary of Changes

- Removed `config.set_flag(trt.BuilderFlag.FP16)` and all other precision-enabling builder flags
- Added a preprocessing step using ModelOpt AutoCast to convert the model to mixed precision
- The `STRONGLY_TYPED` network definition creation flag is no longer needed - all networks are strongly typed by default

3.2. Migrating INT8 Calibration to Explicit Quantization

TensorRT 11.x removes `IInt8Calibrator` and all its subclasses. Use ModelOpt or manual Q/DQ nodes for explicit quantization instead of runtime calibration.

3.2.1. Before (TensorRT 10.x)

```python
import tensorrt as trt

class MyCalibrator(trt.IInt8EntropyCalibrator2):
    def __init__(self, data_loader):
        super().__init__()
        self.data_loader = data_loader
        self.batch_iter = iter(data_loader)

    def get_batch_size(self):
        return 1

    def get_batch(self, names):
        try:
            batch = next(self.batch_iter)
            # Copy batch to device and return device pointers
            return [int(batch.data_ptr())]
        except StopIteration:
            return None

    def read_calibration_cache(self):
        return None

    def write_calibration_cache(self, cache):
        pass

logger = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
network = builder.create_network()
config = builder.create_builder_config()

# Implicit quantization via calibrator
config.set_flag(trt.BuilderFlag.INT8)
config.int8_calibrator = MyCalibrator(data_loader)

parser = trt.OnnxParser(network, logger)
with open("model.onnx", "rb") as f:
    parser.parse(f.read())

engine_bytes = builder.build_serialized_network(network, config)
```

In TensorRT 11.x, `IInt8Calibrator` and all its subclasses have been removed. Use ModelOpt or manual Q/DQ nodes for explicit quantization.

3.2.2. After (TensorRT 11.x)

```python
import tensorrt as trt

# Step 1: Quantize the model offline using ModelOpt (recommended)
# python -m modelopt.onnx.quantization --onnx_path model.onnx --calibration_data data.npz

# Step 2: Build the quantized model (Q/DQ nodes are already in the ONNX graph)
logger = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
network = builder.create_network()
config = builder.create_builder_config()

parser = trt.OnnxParser(network, logger)
with open("model_quantized.onnx", "rb") as f:
    parser.parse(f.read())

engine_bytes = builder.build_serialized_network(network, config)
```

3.2.3. Summary of Changes

- Removed the IInt8Calibrator subclass and all calibration-related code
- Removed `config.set_flag(trt.BuilderFlag.INT8)` and `config.int8_calibrator` assignment
- Quantization is now applied to the model itself (using Q/DQ nodes) before building, not during building

3.3. Migrating Plugins from V2 to V3

The V2 plugin interfaces have been deprecated since TensorRT 10 and remain present in TensorRT 11.x; no V2 classes are removed in this release, but several deprecated methods on them have been removed (the implicit-batch broadcast helpers on `IPluginV2Ext` / `IPluginV2DynamicExt` and the legacy `IPluginRegistry` overloads that took `IPluginCreator`). New plugins should use the `IPluginV3` interface with `IPluginCreatorV3One`, and existing V2 plugins should be migrated. Refer to the samples/python/sample_plugin_v2_to_v3_migration sample that demonstrates how to migrate V2 plugins to V3.

See also

Side-by-Side V2 → V3 API Mapping (in the *TensorRT Developer Guide*, *Plugins API Migration* chapter)
Method-by-method mapping table grouped by lifecycle phase (core, build, runtime, serialization, network attachment).

Known Migration Issues (in the *TensorRT Developer Guide*, *Plugins API Migration* chapter)
Known issues encountered when porting V2 plugins, including the empty `PluginField` initializer crash and the strongly-typed network requirement.

Performance: Resolving V2 → V3 Regressions (in the *TensorRT Developer Guide*, *Plugins API Migration* chapter)
Checklist for resolving performance regressions after migrating a plugin from `IPluginV2DynamicExt` to `IPluginV3`.

3.3.1. Before (TensorRT 10.x)

```python
import tensorrt as trt

class MyPluginV2(trt.IPluginV2DynamicExt):
    def __init__(self):
        super().__init__()
        self.plugin_namespace = ""
        self.plugin_type = "MyPlugin"
        self.plugin_version = "1"
        self.num_outputs = 1

    def get_output_dimensions(self, output_index, inputs, exprBuilder):
        return inputs[0]  # Same shape as input

    def configure_plugin(self, inp, out):
        pass

    def supports_format_combination(self, pos, in_out, num_inputs):
        return in_out[pos].format == trt.TensorFormat.LINEAR and \
               in_out[pos].type == trt.DataType.FLOAT

    def enqueue(self, input_desc, output_desc, inputs, outputs, workspace,
↪stream):
        # Kernel execution
        pass

    def clone(self):
        return MyPluginV2()

    def serialize(self):
        return b""

# Creator
class MyPluginCreatorV2(trt.IPluginCreator):
    def __init__(self):
        super().__init__()
        self.name = "MyPlugin"
        self.plugin_namespace = ""
        self.plugin_version = "1"
        self.field_names = trt.PluginFieldCollection([])

    def create_plugin(self, name, fc):
        return MyPluginV2()

    def deserialize_plugin(self, name, data):
```

(continues on next page)

(continued from previous page)

```
        return MyPluginV2()

# Usage
trt.get_plugin_registry().register_creator(MyPluginCreatorV2())
plugin = MyPluginV2()
layer = network.add_plugin_v2([input_tensor], plugin)
```

3.3.2. After (TensorRT 11.x)

```
import tensorrt as trt

class MyPluginV3(trt.IPluginV3, trt.IPluginV3OneCore,
                 trt.IPluginV3OneBuild, trt.IPluginV3OneRuntime):
    def __init__(self):
        trt.IPluginV3.__init__(self)
        trt.IPluginV3OneCore.__init__(self)
        trt.IPluginV3OneBuild.__init__(self)
        trt.IPluginV3OneRuntime.__init__(self)
        self.num_outputs = 1
        self.plugin_namespace = ""
        self.plugin_name = "MyPlugin"
        self.plugin_version = "1"

    def get_capability_interface(self, type):
        return self

    def get_output_data_types(self, input_types):
        return [input_types[0]]  # Same type as input

    def get_output_shapes(self, inputs, shape_inputs, exprBuilder):
        # Return a list of DimsExprs, one per output
        output = trt.DimsExprs(len(inputs[0]))
        for i in range(len(inputs[0])):
            output[i] = inputs[0][i]
        return [output]

    def supports_format_combination(self, pos, in_out, num_inputs):
        return in_out[pos].format == trt.TensorFormat.LINEAR and \
              in_out[pos].type == trt.DataType.FLOAT

    def configure_plugin(self, inp, out):
        pass

    def on_shape_change(self, inp, out):
        return 0

    def enqueue(self, input_desc, output_desc, inputs, outputs, workspace,
↪stream):
        # Kernel execution
        pass
```

(continues on next page)

(continued from previous page)

```
    def get_fields_to_serialize(self):
        return trt.PluginFieldCollection([])

    def get_valid_tactics(self):
        return []

    def set_tactic(self, tactic):
        return 0

    def clone(self):
        cloned_plugin = MyPluginV3()
        cloned_plugin.__dict__.update(self.__dict__)
        return cloned_plugin

    def attach_to_context(self, context):
        return self.clone()

# Creator
class MyPluginCreatorV3(trt.IPluginCreatorV3One):
    def __init__(self):
        trt.IPluginCreatorV3One.__init__(self)
        self.name = "MyPlugin"
        self.plugin_namespace = ""
        self.plugin_version = "1"
        self.field_names = trt.PluginFieldCollection([])

    def create_plugin(self, name, fc, phase):
        return MyPluginV3()

# Usage
trt.get_plugin_registry().register_creator(MyPluginCreatorV3())
plugin = MyPluginV3()
layer = network.add_plugin_v3([input_tensor], [], plugin)
```

3.3.3. Summary of Changes

- Plugin class now inherits from `IPluginV3`, `IPluginV3OneCore`, `IPluginV3OneBuild`, and `IPluginV3OneRuntime` instead of `IPluginV2DynamicExt`
- Added `get_capability_interface()` to return the appropriate capability for each phase
- `get_output_dimensions()` replaced by `get_output_shapes()`, which receives both data inputs and shape inputs and returns a list of `DimsExprs`
- `get_output_data_types()` is a new required method
- `serialize()` replaced by `get_fields_to_serialize()`, which returns a `PluginFieldCollection`
- Creator inherits from `IPluginCreatorV3One` instead of `IPluginCreator`; `create_plugin()` receives an additional `phase` parameter and `deserialize_plugin()` is no longer needed

- `network.add_plugin_v2()` replaced by `network.add_plugin_v3()`, which accepts an additional list of shape inputs

3.3.4. Known Issues When Migrating Plugins

- **Empty PluginField initializers can crash V3 dispatch.** When a plugin advertises a `PluginField` whose `data` is empty (`b""` or a zero-length array), the V3 creator dispatch path can crash during build or deserialization. Populate every entry with a non-empty sentinel value, even when it is unused at runtime:

```
import numpy as np
import tensorrt as trt

# Bad: empty initializer
fields = trt.PluginFieldCollection([
    trt.PluginField("flag", b"", trt.PluginFieldType.INT32),
])

# Good: non-empty sentinel keeps the dispatch path safe
fields = trt.PluginFieldCollection([
    trt.PluginField("flag", np.array([0], dtype=np.int32), trt.
↪PluginFieldType.INT32),
])
```

- **Use strongly-typed networks with IPluginV3.** Mixing `IPluginV3` plugins with weakly-typed networks can hit fusion paths that were not exercised by `IPluginV2DynamicExt` and trigger crashes. In TensorRT 11.0.0 all precision-enabling builder flags (`BuilderFlag.FP16`, `INT8`, `BF16`, `FP8`, `INT4`, `FP4`) have been removed, so any network you build is strongly typed by default; no action required for fresh 11.x builds. Authors back-porting V3 plugins to a 10.x build for evaluation must explicitly opt in with `builder.create_network(int(trt.NetworkDefinitionCreationFlag.STRONGLY_TYPED))`.

3.4. Migrating Weight Streaming APIs

The weight streaming API has been updated in TensorRT 11.x. The `minimum_weight_streaming_budget` property has been removed; compute a budget from `streamable_weights_size` and available device memory instead.

3.4.1. Before (TensorRT 10.x)

```
import tensorrt as trt
import pycuda.driver as cuda

engine = runtime.deserialize_cuda_engine(engine_bytes)

# Old API
budget = engine.get_minimum_weight_streaming_budget
engine.weight_streaming_budget = budget
current = engine.weight_streaming_budget
```

3.4.2. After (TensorRT 11.x)

```
import tensorrt as trt
import pycuda.driver as cuda

engine = runtime.deserialize_cuda_engine(engine_bytes)

# V2 API
free, total = cuda.mem_get_info()
weights_size = engine.streamable_weights_size
budget = min(free // 2, weights_size // 2)
engine.weight_streaming_budget_v2 = budget
current = engine.weight_streaming_budget_v2
```

3.4.3. Summary of Changes

- `weight_streaming_budget` replaced by `weight_streaming_budget_v2`
- `get_minimum_weight_streaming_budget` removed - compute your own budget based on available memory and `streamable_weights_size`

3.5. Migrating Memory Management APIs

TensorRT 11.x replaces `device_memory_size` with `device_memory_size_v2` (which returns `int64`) and removes `create_execution_context_without_device_memory()`.

3.5.1. Before (TensorRT 10.x)

```
engine = runtime.deserialize_cuda_engine(engine_bytes)

# Old API
mem_size = engine.device_memory_size
context = engine.create_execution_context_without_device_memory()
```

3.5.2. After (TensorRT 11.x)

```
engine = runtime.deserialize_cuda_engine(engine_bytes)

# V2 API (returns int64)
mem_size = engine.device_memory_size_v2
context = engine.create_execution_context()
```

3.5.3. Summary of Changes

- `device_memory_size` replaced by `device_memory_size_v2` (returns `int64` instead of `size_t`)

- `create_execution_context_without_device_memory()` removed - use `create_execution_context()` with appropriate runtime configuration

3.6. Removed Python APIs and Replacements

Warning

The APIs listed below have been **removed** in TensorRT 11.x and will cause runtime errors if called. Review each entry for its replacement before upgrading.

The following Python APIs have been removed. Each entry shows the removed API and its replacement or migration path.

`BuilderFlag.FP16`
: Strong typing with ModelOpt AutoCast

`BuilderFlag.INT8`
: Explicit quantization with Q/DQ nodes

`BuilderFlag.FP8`
: Explicit quantization with Q/DQ nodes

`BuilderFlag.BF16`
: Strong typing with ModelOpt AutoCast

`BuilderFlag.INT4`
: Explicit quantization with Q/DQ nodes

`BuilderFlag.FP4`
: Explicit quantization with Q/DQ nodes

`BuilderFlag.OBEY_PRECISION_CONSTRAINTS`
: Strong typing (always enforced)

`BuilderFlag.PREFER_PRECISION_CONSTRAINTS`
: Strong typing (always enforced)

`BuilderFlag.DIRECT_IO`
: Removed (unneeded)

`IBuilderConfig.int8_calibrator`
: Explicit quantization with Q/DQ nodes

`IBuilderConfig.set_calibration_profile()`
: Explicit quantization with Q/DQ nodes

`IBuilderConfig.get_calibration_profile()`
: Explicit quantization with Q/DQ nodes

`IBuilderConfig.set_quantization_flag()`
: Explicit quantization with Q/DQ nodes

`IBuilderConfig.get_quantization_flag()`
: Explicit quantization with Q/DQ nodes

`IBuilderConfig.quantization_flags`
: Explicit quantization with Q/DQ nodes

IBuilderConfig.clear_quantization_flag()
Explicit quantization with Q/DQ nodes

ICudaEngine.device_memory_size
`ICudaEngine.device_memory_size_v2`

ICudaEngine.device_memory_size_for_profile()
`ICudaEngine.device_memory_size_for_profile_v2()`

ICudaEngine.has_implicit_batch_dimension
Removed (always `False`)

ICudaEngine.weight_streaming_budget
`ICudaEngine.weight_streaming_budget_v2`

ICudaEngine.minimum_weight_streaming_budget
Compute from `streamable_weights_size` and available memory

ICudaEngine.create_execution_context_without_device_memory()
`ICudaEngine.create_execution_context()`

ICudaEngine.get_profile_tensor_values()
`ICudaEngine.get_profile_tensor_values_v2()`

IExecutionContext.all_input_shapes_specified
Removed (always `True`)

IExecutionContext.device_memory
`IExecutionContext.device_memory_v2`

IInt8Calibrator (all subclasses)
Explicit quantization with Q/DQ nodes

ILayer.precision
Strong typing (set types on tensors directly)

ILayer.precision_is_set
Removed

ILayer.reset_precision()
Removed

ILayer.set_output_type()
Strong typing (set types on tensors directly)

ILayer.output_type_is_set()
Removed

ILayer.reset_output_type()
Removed

INormalizationLayer.compute_precision
Removed (use strong typing)

INetworkDefinition.add_plugin_v2()
`INetworkDefinition.add_plugin_v3()`

INetworkDefinition.add_normalization()
`INetworkDefinition.add_normalization_v2()`

IPluginV2DynamicExt
`IPluginV3`

IPluginCreator
: IPluginCreatorV3One

IPluginRegistry.register_creator() (old overload)
: IPluginRegistry.register_creator() (accepts IPluginCreatorInterface)

IPluginRegistry.plugin_creator_list
: IPluginRegistry.all_creators

IPluginRegistry.get_plugin_creator()
: IPluginRegistry.get_creator()

IRefitter.set_dynamic_range()
: Explicit quantization with Q/DQ nodes

IRefitter.get_dynamic_range_min()
: Explicit quantization with Q/DQ nodes

IRefitter.get_dynamic_range_max()
: Explicit quantization with Q/DQ nodes

IRefitter.get_tensors_with_dynamic_range()
: Explicit quantization with Q/DQ nodes

ITensor.set_type()
: Strong typing (type determined by network construction)

ITensor.dynamic_range
: Explicit quantization with Q/DQ nodes

ITensor.is_dynamic_range_set
: Removed

ITensor.reset_dynamic_range()
: Removed

TacticSource.CUBLAS
: Removed

TacticSource.CUBLAS_LT
: Removed

TacticSource.CUDNN
: Removed

Chapter 4. Migrating C++ Code from TensorRT 10.x to 11.x

This page describes how to update C++ code when you migrate from TensorRT 10.x to 11.x: paired examples for strongly typed networks, explicit quantization, plugin migration, and updated runtime APIs, followed by lists of C++ APIs added and removed in 11.x.

Warning

Engine files are executable artifacts that contain compiled CUDA tactics. The `deserializeCudaEngine` examples in *Migrating from Weak Typing to Strong Typing* assume the engine file was built by you on a trusted host or received over a trusted, authenticated channel. Never deserialize an engine file from an untrusted source. Refer to Engine Deserialization and the Trust Boundary for the full trust-boundary model.

4.1. Migrating from Weak Typing to Strong Typing

TensorRT 11.x removes all precision-enabling builder flags such as `BuilderFlag::kFP16` and `BuilderFlag::kINT8`. Use ModelOpt AutoCast to convert your ONNX model to mixed precision before building.

4.1.1. Before (TensorRT 10.x)

```
auto builder = SampleUniquePtr<nvinfer1::IBuilder>
↪(nvinfer1::createInferBuilder(logger));
auto network = SampleUniquePtr<nvinfer1::INetworkDefinition>(builder->
↪createNetworkV2(0));
auto config = SampleUniquePtr<nvinfer1::IBuilderConfig>(builder->
↪createBuilderConfig());

// Weak typing: TensorRT automatically considers FP16 kernels
config->setFlag(BuilderFlag::kFP16);

auto parser = SampleUniquePtr<nvonnxparser::IParser>(
    nvonnxparser::createParser(*network, logger));
```

(continues on next page)

(continued from previous page)

```
parser->parseFromFile("model.onnx", static_cast<int>
↪(nvinfer1::ILogger::Severity::kWARNING));

auto plan = SampleUniquePtr<IHostMemory>(builder->
↪buildSerializedNetwork(*network, *config));
```

In TensorRT 11.x, `BuilderFlag::kFP16` and all other precision-enabling builder flags have been removed. Use ModelOpt AutoCast to convert the ONNX model to mixed precision before building.

4.1.2. After (TensorRT 11.x)

```
// Step 1: Convert model to mixed precision offline using ModelOpt:
// python -m modelopt.onnx.autocast --onnx_path model.onnx --output_path model_
↪fp16.onnx

// Step 2: Build with strongly typed network (always on in 11.x)
auto builder = SampleUniquePtr<nvinfer1::IBuilder>
↪(nvinfer1::createInferBuilder(logger));
auto network = SampleUniquePtr<nvinfer1::INetworkDefinition>(builder->
↪createNetworkV2(0));
auto config = SampleUniquePtr<nvinfer1::IBuilderConfig>(builder->
↪createBuilderConfig());

// No precision flags needed - the model itself specifies types

auto parser = SampleUniquePtr<nvonnxparser::IParser>(
    nvonnxparser::createParser(*network, logger));
parser->parseFromFile("model_fp16.onnx", static_cast<int>
↪(nvinfer1::ILogger::Severity::kWARNING));

auto plan = SampleUniquePtr<IHostMemory>(builder->
↪buildSerializedNetwork(*network, *config));
```

4.1.3. Summary of Changes

- Removed `config->setFlag(BuilderFlag::kFP16)` and all other precision flags (`kINT8`, `kFP8`, `kBF16`, `kINT4`, `kFP4`)
- Added an offline preprocessing step using ModelOpt AutoCast to produce a mixed-precision ONNX model
- No code changes needed for the build path itself beyond removing the flag

4.2. Migrating INT8 Calibration to Explicit Quantization

TensorRT 11.x removes `IInt8Calibrator` and all its subclasses, along with `setInt8Calibrator()`. Use ModelOpt or manual Q/DQ nodes for explicit quantization instead.

4.2.1. Before (TensorRT 10.x)

```
class MyCalibrator : public nvinfer1::IInt8EntropyCalibrator2
{
public:
    int32_t getBatchSize() const noexcept override { return 1; }

    bool getBatch(void* bindings[], char const* names[], int32_t nbBindings)
↪noexcept override
    {
        // Fill bindings with calibration data
        if (mCurrentBatch >= mNumBatches)
            return false;
        // ... copy data to GPU
        mCurrentBatch++;
        return true;
    }

    void const* readCalibrationCache(size_t& length) noexcept override {
↪return nullptr; }
    void writeCalibrationCache(void const* ptr, size_t length) noexcept
↪override {}

private:
    int32_t mCurrentBatch{0};
    int32_t mNumBatches{100};
};

// Usage
auto config = SampleUniquePtr<nvinfer1::IBuilderConfig>(builder->
↪createBuilderConfig());
config->setFlag(BuilderFlag::kINT8);

MyCalibrator calibrator;
config->setInt8Calibrator(&calibrator);

auto plan = SampleUniquePtr<IHostMemory>(builder->
↪buildSerializedNetwork(*network, *config));
```

In TensorRT 11.x, `IInt8Calibrator` and all subclasses have been removed along with `setInt8Calibrator()`. Use ModelOpt or manual Q/DQ nodes.

4.2.2. After (TensorRT 11.x)

```
// Step 1: Quantize the model offline using ModelOpt:
//   python -m modelopt.onnx.quantization --onnx_path model.onnx --calibration_
↪data data.npz
//
// Alternatively, add QuantizeLinear/DequantizeLinear nodes to the ONNX graph
↪manually,
// or use the INetworkDefinition::addQuantize() and addDequantize() APIs.

```

(continues on next page)

(continued from previous page)

```
// Step 2: Build the pre-quantized model
auto builder = SampleUniquePtr<nvinfer1::IBuilder>
↪(nvinfer1::createInferBuilder(logger));
auto network = SampleUniquePtr<nvinfer1::INetworkDefinition>(builder->
↪createNetworkV2(0));
auto config = SampleUniquePtr<nvinfer1::IBuilderConfig>(builder->
↪createBuilderConfig());

auto parser = SampleUniquePtr<nvonnxparser::IParser>(
    nvonnxparser::createParser(*network, logger));
parser->parseFromFile("model_quantized.onnx",
    static_cast<int>(nvinfer1::ILogger::Severity::kWARNING));

auto plan = SampleUniquePtr<IHostMemory>(builder->
↪buildSerializedNetwork(*network, *config));
```

4.2.3. Summary of Changes

- Removed the `IInt8Calibrator` subclass entirely
- Removed `config->setFlag(BuilderFlag::kINT8)` and `config->setInt8Calibrator()`
- Quantization is applied to the model offline (Q/DQ nodes in the ONNX graph) or using the `addQuantize()` / `addDequantize()` network definition APIs

4.3. Migrating Plugins from `IPluginV2DynamicExt` to `IPluginV3`

The following example shows a complete plugin migration from V2 to V3 using a NonZero plugin that computes the indices of non-zero elements. This demonstrates V3's support for data-dependent output shapes, which was not possible with V2.

See also

Side-by-Side V2 → V3 API Mapping (in the *TensorRT Developer Guide*, *Plugins API Migration* chapter)
Method-by-method mapping table grouped by lifecycle phase (core, build, runtime, serialization, network attachment).

Known Migration Issues (in the *TensorRT Developer Guide*, *Plugins API Migration* chapter)
Known issues encountered when porting V2 plugins, including the empty `PluginField` initializer crash and the strongly-typed network requirement.

Performance: Resolving V2 → V3 Regressions (in the *TensorRT Developer Guide*, *Plugins API Migration* chapter)
Checklist for resolving performance regressions after migrating a plugin from `IPluginV2DynamicExt` to `IPluginV3`.

4.3.1. Before (TensorRT 10.x - IPluginV2DynamicExt)

```
class NonZeroPluginV2 : public nvinfer1::IPluginV2DynamicExt
{
public:
    // IPluginV2 core methods
    char const* getPluginType() const noexcept override { return "NonZeroPlugin"; }
    char const* getPluginVersion() const noexcept override { return "1"; }
    int32_t getNbOutputs() const noexcept override { return 1; }

    // Output dimensions - limited to expressions of input dimensions only
    DimsExprs getOutputDimensions(int32_t outputIndex, DimsExprs const* inputs,
        int32_t nbInputs, IExprBuilder& exprBuilder) noexcept override
    {
        // Cannot express data-dependent shapes - must use an upper bound
        DimsExprs output;
        output.nbDims = 2;
        output.d[0] = exprBuilder.operation(DimensionOperation::kPROD,
            *inputs[0].d[0], *inputs[0].d[1]); // Upper bound: R * C
        output.d[1] = exprBuilder.constant(2);
        return output;
    }

    bool supportsFormatCombination(int32_t pos, PluginTensorDesc const* inOut,
        int32_t nbInputs, int32_t nbOutputs) noexcept override
    {
        return inOut[pos].format == TensorFormat::kLINEAR;
    }

    void configurePlugin(DynamicPluginTensorDesc const* in, int32_t nbInputs,
        DynamicPluginTensorDesc const* out, int32_t nbOutputs) noexcept override {}

    int32_t enqueue(PluginTensorDesc const* inputDesc, PluginTensorDesc const* outputDesc,
        void const* const* inputs, void* const* outputs,
        void* workspace, cudaStream_t stream) noexcept override
    {
        // Execute kernel
        return 0;
    }

    size_t getWorkspaceSize(PluginTensorDesc const* inputs, int32_t nbInputs,
        PluginTensorDesc const* outputs, int32_t nbOutputs) const noexcept override
    {
        return 0;
    }

    // Serialization
```

(continues on next page)

(continued from previous page)

```
    size_t getSerializationSize() const noexcept override { return sizeof(bool); }
    void serialize(void* buffer) const noexcept override
    {
        *reinterpret_cast<bool*>(buffer) = mRowOrder;
    }

    IPluginV2DynamicExt* clone() const noexcept override
    {
        return new NonZeroPluginV2(mRowOrder);
    }

    // ... other required IPluginV2 methods (destroy, setPluginNamespace, etc.)

private:
    bool mRowOrder{true};
};

// V2 Plugin Creator
class NonZeroCreatorV2 : public nvinfer1::IPluginCreator
{
public:
    char const* getPluginName() const noexcept override { return "NonZeroPlugin"; }
    char const* getPluginVersion() const noexcept override { return "1"; }
    PluginFieldCollection const* getFieldNames() noexcept override { return &mFC; }

    IPluginV2* createPlugin(char const* name, PluginFieldCollection const* fc) noexcept override
    {
        return new NonZeroPluginV2(/*rowOrder=*/true);
    }

    IPluginV2* deserializePlugin(char const* name, void const* data,
        size_t length) noexcept override
    {
        bool rowOrder = *reinterpret_cast<bool const*>(data);
        return new NonZeroPluginV2(rowOrder);
    }

    // ... other required methods

private:
    PluginFieldCollection mFC{};
};

// Usage
NonZeroPluginV2 plugin(/*rowOrder=*/true);
auto* layer = network->addPluginV2(&inputTensor, 1, plugin);
```

4.3.2. After (TensorRT 11.x - IPluginV3)

```
class NonZeroPlugin : public IPluginV3, public IPluginV3OneCore,
                  public IPluginV3OneBuild, public IPluginV3OneRuntime
{
public:
    NonZeroPlugin(bool rowOrder) : mRowOrder(rowOrder) {}

    // IPluginV3 - return the appropriate capability interface
    IPluginCapability* getCapabilityInterface(PluginCapabilityType type)
↪noexcept override
    {
        if (type == PluginCapabilityType::kBUILD)
            return static_cast<IPluginV3OneBuild*>(this);
        if (type == PluginCapabilityType::kRUNTIME)
            return static_cast<IPluginV3OneRuntime*>(this);
        return static_cast<IPluginV3OneCore*>(this);
    }

    // IPluginV3OneCore
    AsciiChar const* getPluginName() const noexcept override { return
↪"NonZeroPlugin"; }
    AsciiChar const* getPluginVersion() const noexcept override { return "1";
↪}
    AsciiChar const* getPluginNamespace() const noexcept override { return "
↪"; }

    // IPluginV3OneBuild
    int32_t getNbOutputs() const noexcept override { return 2; } // data +
↪size tensor

    int32_t getOutputDataTypes(DataType* outputTypes, int32_t nbOutputs,
        DataType const* inputTypes, int32_t nbInputs) const noexcept override
    {
        outputTypes[0] = DataType::kINT32; // non-zero indices
        outputTypes[1] = DataType::kINT64; // size tensor
        return 0;
    }

    // Output shapes - V3 supports data-dependent shapes via declareSizeTensor
    int32_t getOutputShapes(DimsExprs const* inputs, int32_t nbInputs,
        DimsExprs const* shapeInputs, int32_t nbShapeInputs,
        DimsExprs* outputs, int32_t nbOutputs,
        IExprBuilder& exprBuilder) noexcept override
    {
        auto upperBound = exprBuilder.operation(DimensionOperation::kPROD,
            *inputs[0].d[0], *inputs[0].d[1]);
        auto optValue = exprBuilder.operation(DimensionOperation::kFLOOR_DIV,
            *upperBound, *exprBuilder.constant(2));

        // Declare a size tensor - enables data-dependent output shapes
        auto numNonZero = exprBuilder.declareSizeTensor(1, *optValue,
↪*upperBound);
```

(continues on next page)

(continued from previous page)

```
46
47          outputs[0].nbDims = 2;
48          outputs[0].d[0] = numNonZero; // Data-dependent dimension
49          outputs[0].d[1] = exprBuilder.constant(2);
50
51          outputs[1].nbDims = 0; // Size tensor is a scalar
52          return 0;
53      }
54
55      bool supportsFormatCombination(int32_t pos, DynamicPluginTensorDesc
    ↪const* inOut,
56          int32_t nbInputs, int32_t nbOutputs) noexcept override
57      {
58          return inOut[pos].desc.format == TensorFormat::kLINEAR;
59      }
60
61      int32_t configurePlugin(DynamicPluginTensorDesc const* in, int32_t
    ↪nbInputs,
62          DynamicPluginTensorDesc const* out, int32_t nbOutputs) noexcept
    ↪override
63      {
64          return 0;
65      }
66
67      size_t getWorkspaceSize(DynamicPluginTensorDesc const* inputs, int32_t
    ↪nbInputs,
68          DynamicPluginTensorDesc const* outputs, int32_t nbOutputs) const
    ↪noexcept override
69      {
70          return 0;
71      }
72
73      // IPluginV3OneRuntime
74      int32_t enqueue(PluginTensorDesc const* inputDesc, PluginTensorDesc
    ↪const* outputDesc,
75          void const* const* inputs, void* const* outputs,
76          void* workspace, cudaStream_t stream) noexcept override
77      {
78          // Execute kernel - same as V2
79          return 0;
80      }
81
82      int32_t onShapeChange(PluginTensorDesc const* in, int32_t nbInputs,
83          PluginTensorDesc const* out, int32_t nbOutputs) noexcept override
84      {
85          return 0;
86      }
87
88      // Serialization - uses PluginFieldCollection instead of raw bytes
89      PluginFieldCollection const* getFieldsToSerialize() noexcept override
90      {
91          mDataToSerialize.clear();
```

(continues on next page)

(continued from previous page)

```
        mDataToSerialize.emplace_back("rowOrder", &mRowOrder,
↪PluginFieldType::kINT32, 1);
        mFCToSerialize.nbFields = mDataToSerialize.size();
        mFCToSerialize.fields = mDataToSerialize.data();
        return &mFCToSerialize;
    }

    IPluginV3* attachToContext(IPluginResourceContext* context) noexcept
↪override
    {
        return clone();
    }

    IPluginV3* clone() noexcept override
    {
        return new NonZeroPlugin(mRowOrder);
    }

private:
    bool mRowOrder{true};
    std::vector<PluginField> mDataToSerialize;
    PluginFieldCollection mFCToSerialize{};
};

// V3 Plugin Creator
class NonZeroCreator : public nvinfer1::IPluginCreatorV3One
{
public:
    NonZeroCreator()
    {
        mPluginAttributes.emplace_back("rowOrder", nullptr,
↪PluginFieldType::kINT32, 1);
        mFC.nbFields = mPluginAttributes.size();
        mFC.fields = mPluginAttributes.data();
    }

    char const* getPluginName() const noexcept override { return
↪"NonZeroPlugin"; }
    char const* getPluginVersion() const noexcept override { return "1"; }
    char const* getPluginNamespace() const noexcept override { return ""; }
    PluginFieldCollection const* getFieldNames() noexcept override { return &
↪mFC; }

    // Phase-aware creation - no separate deserializePlugin needed
    IPluginV3* createPlugin(char const* name, PluginFieldCollection const*
↪fc,
        TensorRTPhase phase) noexcept override
    {
        bool rowOrder = true;
        for (int32_t i = 0; i < fc->nbFields; ++i)
        {
            if (std::string_view(fc->fields[i].name) == "rowOrder")
```

(continues on next page)

(continued from previous page)

```
                rowOrder = *static_cast<bool const*>(fc->fields[i].data);
        }
        return new NonZeroPlugin(rowOrder);
    }

private:
    PluginFieldCollection mFC{};
    std::vector<PluginField> mPluginAttributes;
};

// Usage - addPluginV3 accepts both data inputs and shape inputs
NonZeroPlugin plugin(/*rowOrder=*/true);
ITensor* inputs[] = {&inputTensor};
auto* layer = network->addPluginV3(inputs, 1, nullptr, 0, plugin);
```

4.3.3. Summary of Changes

- Plugin class inherits from `IPluginV3`, `IPluginV3OneCore`, `IPluginV3OneBuild`, and `IPluginV3OneRuntime` instead of `IPluginV2DynamicExt`
- Added `getCapabilityInterface()` to return the appropriate interface for each phase (`core`, `build`, `runtime`)
- `getOutputDimensions()` replaced by `getOutputShapes()`, which supports data-dependent output shapes using `exprBuilder.declareSizeTensor()`
- Added required `getOutputDataTypes()` method
- `serialize()` / `getSerializationSize()` replaced by `getFieldsToSerialize()`, which returns a `PluginFieldCollection` for structured serialization
- Added `onShapeChange()` and `attachToContext()` methods
- Creator inherits from `IPluginCreatorV3One` instead of `IPluginCreator`; `createPlugin()` takes a `TensorRTPhase` parameter, and `deserializePlugin()` is no longer needed - `createPlugin()` handles both build and runtime phases
- `addPluginV2(inputs, nbInputs, plugin)` replaced by `addPluginV3(inputs, nbInputs, shapeInputs, nbShapeInputs, plugin)`

4.3.4. Known Issues When Migrating Plugins

- **Empty PluginField initializers can crash V3 dispatch.** When a plugin advertises a `PluginField` with a `nullptr` `data` pointer and `length == 0`, the V3 creator dispatch path can dereference the pointer during build or deserialization. Populate every entry with a non-null sentinel buffer, even when the value is unused at runtime:

```
// Bad - empty initializer
mPluginAttributes.emplace_back("flag", nullptr, PluginFieldType::kINT32,
↪0);

// Good - non-null sentinel keeps the dispatch path safe
static int32_t kDummy = 0;
```

(continues on next page)

(continued from previous page)

```
mPluginAttributes.emplace_back("flag", &kDummy, PluginFieldType::kINT32,
↪1);
```

- **Use strongly-typed networks with IPluginV3.** Mixing `IPluginV3` plugins with weakly-typed networks can hit fusion paths that were not exercised by `IPluginV2DynamicExt` and trigger crashes. In TensorRT 11.0.0 all precision-enabling builder flags (`BuilderFlag::kFP16`, `kINT8`, `kBF16`, `kFP8`, `kINT4`, `kFP4`) have been removed, so any network you build is strongly typed by default; no action required for fresh 11.x builds. Authors backporting V3 plugins to a 10.x build for evaluation must explicitly opt in with `createNetworkV2(NetworkDefinitionCreationFlag::kSTRONGLY_TYPED)`.

4.4. Migrating Weight Streaming APIs

The weight streaming API has been updated in TensorRT 11.x. The `getMinimumWeightStreamingBudget()` method has been removed; compute a budget from `getStreamableWeightsSize()` and available device memory instead.

4.4.1. Before (TensorRT 10.x)

```
auto engine = SampleUniquePtr<ICudaEngine>(
runtime->deserializeCudaEngine(engineData, engineSize));

// Old API
int64_t minBudget = engine->getMinimumWeightStreamingBudget();
engine->setWeightStreamingBudget(minBudget);
int64_t currentBudget = engine->getWeightStreamingBudget();
```

4.4.2. After (TensorRT 11.x)

```
auto engine = SampleUniquePtr<ICudaEngine>(
runtime->deserializeCudaEngine(engineData, engineSize));

// V2 API
size_t freeMem, totalMem;
cudaMemGetInfo(&freeMem, &totalMem);
int64_t weightsSize = engine->getStreamableWeightsSize();
int64_t budget = std::min(static_cast<int64_t>(freeMem / 2), weightsSize /
↪2);
engine->setWeightStreamingBudgetV2(budget);
int64_t currentBudget = engine->getWeightStreamingBudgetV2();
```

4.4.3. Summary of Changes

- `setWeightStreamingBudget()` replaced by `setWeightStreamingBudgetV2()`
- `getWeightStreamingBudget()` replaced by `getWeightStreamingBudgetV2()`

- `getMinimumWeightStreamingBudget()` removed - compute a budget using `getStreamableWeightsSize()` and available device memory

4.5. Migrating Memory Management APIs

TensorRT 11.x replaces `getDeviceMemorySize()` with `getDeviceMemorySizeV2()` (which returns `int64_t`), removes `createExecutionContextWithoutDeviceMemory()`, and replaces `setDeviceMemory(void*)` with `setDeviceMemoryV2(void*, int64_t)`.

4.5.1. Before (TensorRT 10.x)

```
auto engine = SampleUniquePtr<ICudaEngine>(
runtime->deserializeCudaEngine(engineData, engineSize));

// Old APIs
size_t memSize = engine->getDeviceMemorySize();
auto context = SampleUniquePtr<IExecutionContext>(
    engine->createExecutionContextWithoutDeviceMemory());

void* deviceMem;
cudaMalloc(&deviceMem, memSize);
context->setDeviceMemory(deviceMem);
```

4.5.2. After (TensorRT 11.x)

```
auto engine = SampleUniquePtr<ICudaEngine>(
runtime->deserializeCudaEngine(engineData, engineSize));

// V2 APIs - int64_t sizes, explicit size parameter
int64_t memSize = engine->getDeviceMemorySizeV2();
auto context = SampleUniquePtr<IExecutionContext>(engine->
↪createExecutionContext());

void* deviceMem;
cudaMalloc(&deviceMem, memSize);
context->setDeviceMemoryV2(deviceMem, memSize);
```

4.5.3. Summary of Changes

- `getDeviceMemorySize()` replaced by `getDeviceMemorySizeV2()` (returns `int64_t` instead of `size_t`)
- `createExecutionContextWithoutDeviceMemory()` removed - use `createExecutionContext()`
- `setDeviceMemory(void*)` replaced by `setDeviceMemoryV2(void*, int64_t)`, which takes an explicit size parameter

4.6. Removed C++ APIs and Replacements

Warning

The APIs listed below have been **removed** in TensorRT 11.x and will cause compile-time errors if used. Review each entry for its replacement before upgrading.

`BuilderFlag::kFP16`
Strong typing with ModelOpt AutoCast

`BuilderFlag::kINT8`
Explicit quantization with Q/DQ nodes

`BuilderFlag::kFP8`
Explicit quantization with Q/DQ nodes

`BuilderFlag::kBF16`
Strong typing with ModelOpt AutoCast

`BuilderFlag::kINT4`
Explicit quantization with Q/DQ nodes

`BuilderFlag::kFP4`
Explicit quantization with Q/DQ nodes

`BuilderFlag::kOBEY_PRECISION_CONSTRAINTS`
Strong typing (always enforced)

`BuilderFlag::kPREFER_PRECISION_CONSTRAINTS`
Strong typing (always enforced)

`BuilderFlag::kDIRECT_IO`
Removed (not needed in 11.x)

`IAlgorithm`, `IAlgorithmContext`, `IAlgorithmIOInfo`, `IAlgorithmSelector`, `IAlgorithmVariant`
Use editable mode in `ITimingCache` instead.

`IBuilderConfig::setInt8Calibrator(IInt8Calibrator*)`
Explicit quantization with Q/DQ nodes

`IBuilder::platformHasFastFp16()`
Removed; use strongly typed networks instead of querying platform FP16 support. Third-party code that links against TensorRT 11.x - including the ONNX Runtime TensorRT Execution Provider - must migrate away from this API. ONNX Runtime 1.27+ supports TensorRT 11.x. Refer to the TensorRT 11.0.0 release notes (Known Issues) and TensorRT 11.1.0 release notes (Fixed Issues).

`IBuilder::platformHasFastInt8()`
Removed; use explicit quantization with Q/DQ nodes instead of querying platform INT8 support.

`IBuilderConfig::getInt8Calibrator()`
Removed

`IBuilderConfig::setCalibrationProfile(IOptimizationProfile const*)`
Removed

`IBuilderConfig::getCalibrationProfile()`
Removed

IBuilderConfig::setQuantizationFlags(QuantizationFlags)
Removed

IBuilderConfig::getQuantizationFlags()
Removed

IBuilderConfig::clearQuantizationFlag(QuantizationFlag)
Removed

IBuilderConfig::setQuantizationFlag(QuantizationFlag)
Removed

IBuilderConfig::getQuantizationFlag(QuantizationFlag)
Removed

ICudaEngine::createExecutionContextWithoutDeviceMemory()
`ICudaEngine::createExecutionContext()`

ICudaEngine::getDeviceMemorySize()
`ICudaEngine::getDeviceMemorySizeV2()`

ICudaEngine::getDeviceMemorySizeForProfile(int32_t)
`ICudaEngine::getDeviceMemorySizeForProfileV2(int32_t)`

ICudaEngine::getMinimumWeightStreamingBudget()
Compute from `getStreamableWeightsSize()`

ICudaEngine::getProfileTensorValues(char const*, int32_t, OptProfileSelector)
`ICudaEngine::getProfileTensorValuesV2()`

ICudaEngine::getWeightStreamingBudget()
`ICudaEngine::getWeightStreamingBudgetV2()`

ICudaEngine::hasImplicitBatchDimension()
Removed (always `false`)

ICudaEngine::setWeightStreamingBudget(int64_t)
`ICudaEngine::setWeightStreamingBudgetV2(int64_t)`

IExecutionContext::allInputShapesSpecified()
Removed (always `true`)

IExecutionContext::setDeviceMemory(void*)
`IExecutionContext::setDeviceMemoryV2(void*, int64_t)`

IGpuAllocator::allocate(uint64_t, uint64_t, AllocatorFlags)
`IGpuAllocator::allocateAsync(uint64_t, uint64_t, AllocatorFlags, cudaStream_t)`

IGpuAllocator::deallocate(void*)
`IGpuAllocator::deallocateAsync(void*, cudaStream_t)`

IInt8Calibrator (all subclasses)
Explicit quantization with Q/DQ nodes

ILayer::setPrecision(DataType)
Strong typing (set types on tensors directly)

ILayer::getPrecision()
Removed

ILayer::precisionIsSet()
Removed

ILayer::resetPrecision()
Removed

ILayer::setOutputType(int32_t, DataType)
Strong typing (set types on tensors directly)

ILayer::outputTypeIsSet(int32_t)
Removed

ILayer::resetOutputType(int32_t)
Removed

INetworkDefinition::addAttention(..., bool)
`INetworkDefinition::addAttentionV2(..., CausalMaskKind)`

INetworkDefinition::addNMS(ITensor&, ITensor&, ITensor&)
`INetworkDefinition::addNMS(..., DataType)` (4-arg version)

INetworkDefinition::addNonZero(ITensor&)
`INetworkDefinition::addNonZero(ITensor&, DataType)`

INetworkDefinition::addNormalization(...)
`INetworkDefinition::addNormalizationV2(...)`

INetworkDefinition::addPluginV2(ITensor* const*, int32_t, IPluginV2&)
`INetworkDefinition::addPluginV3(...)`

INetworkDefinition::addTopK(ITensor&, TopKOperation, int32_t, uint32_t)
`INetworkDefinition::addTopK(..., DataType)` (5-arg version)

INormalizationLayer::setComputePrecision(DataType)
Removed (use strong typing)

IOutputAllocator::reallocateOutput(char const*, void*, uint64_t, uint64_t)
`IOutputAllocator::reallocateOutputAsync(..., cudaStream_t)`

IPluginCreator
`IPluginCreatorV3One`

IPluginRegistry::deregisterCreator(IPluginCreator const&)
`IPluginRegistry::deregisterCreator(IPluginCreatorInterface const&)`

IPluginRegistry::getPluginCreator(...)
`IPluginRegistry::getCreator(...)`

IPluginRegistry::getPluginCreatorList(int32_t*)
`IPluginRegistry::getAllCreators(int32_t*)`

IPluginRegistry::registerCreator(IPluginCreator&, ...)
`IPluginRegistry::registerCreator(IPluginCreatorInterface&, ...)`

IPluginV2DynamicExt
`IPluginV3`

IPluginV2Ext
`IPluginV3`

IPluginV2IOExt
`IPluginV3`

IPluginV2Layer
`IPluginV3Layer`

IRefitter::setDynamicRange(char const*, float, float)
Explicit quantization with Q/DQ nodes

IRefitter::getDynamicRangeMin(char const*)
Removed

IRefitter::getDynamicRangeMax(char const*)
Removed

IRefitter::getTensorsWithDynamicRange()
Removed

IRuntime::deserializeCudaEngine(IStreamReader&)
`IRuntime::deserializeCudaEngine(IStreamReaderV2&)`

ITensor::setType(DataType)
Strong typing (type determined by network construction)

ITensor::setDynamicRange(float, float)
Explicit quantization with Q/DQ nodes

ITensor::dynamicRangeIsSet()
Removed

ITensor::resetDynamicRange()
Removed

ITensor::getDynamicRangeMin()
Removed

ITensor::getDynamicRangeMax()
Removed

ITensor::setBroadcastAcrossBatch(bool)
Removed (implicit batch not supported)

ITensor::getBroadcastAcrossBatch()
Removed (implicit batch not supported)

TacticSource::kCUBLAS
Removed

TacticSource::kCUBLAS_LT
Removed

TacticSource::kCUDNN
Removed

DetectionOutputParameters
Removed

NMSParameters
Removed

CodeTypeSSD
Removed

4.7. Removed C++ Plugins and Replacements

Warning

The plugins listed below have been **removed** in TensorRT 11.x. Using them will cause compilation or linker errors. Review each entry for its replacement before upgrading.

`BatchedNMS_TRT`
: Use `INetworkDefinition::addNMS()`

`BatchedNMSDynamic_TRT`
: Use `INetworkDefinition::addNMS()`

`BatchTilePlugin_TRT`
: Implement with standard TensorRT layers

`Clip_TRT`
: Use `INetworkDefinition::addActivation()` with `kCLIP`

`CoordConvAC`
: Implement with standard TensorRT layers (concatenate coordinate channels with `IConcatenationLayer`, then apply convolution)

`CustomGeluPluginDynamic`
: Use `INetworkDefinition::addActivation()` with `kGELU_ERF` or `kGELU_TANH`

`EfficientNMS_ONNX_TRT`
: Use `INetworkDefinition::addNMS()`

`LReLU_TRT`
: Use `INetworkDefinition::addActivation()` with `kLEAKY_RELU`

`NMS_TRT`
: Use `INetworkDefinition::addNMS()`

`NMSDynamic_TRT`
: Use `INetworkDefinition::addNMS()`

`Normalize_TRT`
: Use `INetworkDefinition::addNormalizationV2()`

`Proposal`
: Implement with standard TensorRT layers

`SingleStepLSTMPlugin`
: Use `INetworkDefinition::addLoop()` or standard RNN decomposition

`SpecialSlice_TRT`
: Use `INetworkDefinition::addSlice()`

`Split`
: Use `INetworkDefinition::addSlice()`

4.8. Deprecated BERT Plugins

The following OSS BERT plugin classes are deprecated in 11.0.0 and scheduled for removal in a future release. Migrate to the listed replacements before upgrading beyond 11.x.

`bertQKVToContextPlugin` / `CustomQKVToContextPluginDynamic`
: Refer to Migrate to IAttention for more information.

Chapter 5. Migrating `trtexec` Usage from TensorRT 10.x to 11.x

This page describes how to update `trtexec` usage when migrating from TensorRT 10.x to 11.x: before/after command examples and lists of removed or deprecated options.

5.1. Migrating Precision Flags

TensorRT 11.x removes all precision flags (`--fp16`, `--int8`, `--bf16`, `--fp8`, `--int4`, `--best`) because all networks are strongly typed. Use ModelOpt AutoCast to produce a mixed-precision model before building.

5.1.1. Before (TensorRT 10.x)

```
trtexec \
    --onnx=model.onnx \
    --saveEngine=engine.plan \
    --fp16
```

In TensorRT 11.x, the `--fp16` flag and all other precision flags have been removed because all networks are strongly typed. Use ModelOpt AutoCast to produce a mixed-precision model, then build it.

5.1.2. After (TensorRT 11.x)

```
# Step 1: Convert model to mixed precision offline
python -m modelopt.onnx.autocast --onnx_path model.onnx --output_path model_
↪fp16.onnx

# Step 2: Build with trtexec (strong typing is always on)
trtexec \
  --onnx=model_fp16.onnx \
  --saveEngine=engine.plan
```

5.1.3. Summary of Changes

- Removed `--fp16` (and `--bf16`, `--fp8`, `--int4`, `--best`)
- Added an offline preprocessing step using ModelOpt AutoCast
- `--stronglyTyped` is no longer needed - it is always enabled

5.2. Migrating INT8 Calibration Workflow

TensorRT 11.x removes the `--int8` and `--calib` flags. Quantization is now applied to the model offline using ModelOpt before building.

5.2.1. Before (TensorRT 10.x)

```
trtexec \
    --onnx=model.onnx \
    --saveEngine=engine.plan \
    --int8 \
    --calib=calibration_cache.bin
```

5.2.2. After (TensorRT 11.x)

```
# Step 1: Quantize the model offline using ModelOpt
python -m modelopt.onnx.quantization \
    --onnx_path model.onnx \
    --calibration_data data.npz

# Step 2: Build with trtexec
trtexec \
    --onnx=model_quantized.onnx \
    --saveEngine=engine.plan
```

5.2.3. Summary of Changes

- Removed `--int8` and `--calib` flags
- Quantization is applied to the model offline, not during engine build

5.3. Removed `trtexec` Flags and Replacements

Warning

The flags listed below have been **removed** in TensorRT 11.x. Using them will cause `trtexec` to exit with an error. Review each entry for its replacement before upgrading.

--fp16
Use ModelOpt AutoCast to produce a mixed-precision model

--bf16
Use ModelOpt AutoCast to produce a mixed-precision model

--int8
Use ModelOpt quantization to produce a quantized model

--fp8
Use ModelOpt quantization to produce a quantized model

--int4
Use ModelOpt quantization to produce a quantized model

--best
Use ModelOpt AutoCast and/or quantization

--precisionConstraints
Removed (strong typing is always enforced)

--layerPrecisions
Removed (use strong typing in the model)

--layerOutputTypes
Removed (use strong typing in the model)

--calib
Use ModelOpt quantization offline

--calibProfile
Removed

5.4. Deprecated `trtexec` Flags and Replacements

The following `trtexec` flags have been deprecated in 11.x but are still accepted. Each entry shows the deprecated flag and its replacement.

--stronglyTyped
Always enabled; flag accepted but has no effect

--inputIOFormats (type+format)
Format-only specification (type portion ignored)

--outputIOFormats (type+format)
Format-only specification (type portion ignored)

--useCudaGraph
Enabled by default; flag accepted but has no effect. Use `--noCudaGraph` to disable CUDA graph.

--useSpinWait
Enabled by default; flag accepted but has no effect. Use `--noSpinWait` to disable spin wait for CUDA synchronizations.

--noDataTransfers
Data transfers are disabled by default; flag accepted but has no effect. Use `--includeDataTransfers` to include data transfer latencies in the benchmarking result.

--separateProfileRun

Always enabled; flag accepted but has no effect. When `--dumpProfile` or `--exportProfile` is set, `trtexec` runs end-to-end performance benchmarking first and then per-layer performance benchmarking as a separate run.

Chapter 6. Migrating Safety Runtime Code from TensorRT 10.x to 11.x

This page describes how to update safety runtime code when migrating from TensorRT 10.x to 11.x.

The most significant change is the removal of frontend safety scope validation. In 10.x, the builder performed a static pre-build check against a “Minimal Safety Scope” allowlist before engine building began. In 11.x, this check is removed and the build-time compiler is the sole authority for safety scope enforcement. As a result, safety build errors now appear as tactic failures with layer-level traceback rather than pre-build scope rejections.

This shift affects builder flag usage, the behavior of `isNetworkSupported()`, error handling patterns, and `trtexec` command-line options. The kernel checker tool also gains MLIR validation in this release. Each section below pairs 10.x and 11.x C++ snippets where applicable.

Important

This section is only applicable when using the TensorRT 11.x safety runtime, which is only available on NVIDIA DriveOS 7.x.

6.1. Adapting to Build-Time Safety Scope Validation

TensorRT 11.x removes pre-build safety scope validation and relies exclusively on build-time enforcement. In 10.x, setting `BuilderFlag::kSAFETY_SCOPE` triggered a pre-compile check against a static “Minimal Safety Scope” allowlist. In 11.x, this pre-build check is removed and the safety scope is now enforced entirely during engine building.

Update your code in three areas: builder configuration, build failure handling, and `isNetworkSupported()` usage.

6.1.1. `BuilderFlag::kSAFETY_SCOPE` is Deprecated

`BuilderFlag::kSAFETY_SCOPE` is retained for API compatibility but has no effect in TensorRT 11.x. Setting it no longer triggers pre-build validation. You can leave existing calls in place or remove them; either way, the flag is silently ignored.

6.1.1.1 Before (TensorRT 10.x)

```
config->setFlag(BuilderFlag::kSAFETY_SCOPE);
// Triggers static per-layer supportsSafety() checks before engine building.
// Returns an error at network definition time for out-of-scope layers.
auto serialized = builder->buildSerializedNetwork(*network, *config);
```

6.1.1.2 After (TensorRT 11.x)

```
// kSAFETY_SCOPE is now a no-op. Remove it or leave it - it has no effect.
// Safety validation occurs exclusively at build time.
auto serialized = builder->buildSerializedNetwork(*network, *config);
```

6.1.1.3 Summary of Changes

- `BuilderFlag::kSAFETY_SCOPE` is deprecated and ignored.
- Safety build failures now surface as build-time tactic errors (for example, `No tactic available`) with layer-level traceback, rather than pre-build scope errors.

6.1.2. `isNetworkSupported()` No Longer Validates Safety Scope

In TensorRT 10.x, `IBuilder::isNetworkSupported()` performed static pre-build safety scope checks when kSAFETY_SCOPE was set. In 11.x, the function is simplified: it checks only architectural constraint violations (for example, hybrid DLA/GPU mode conflicts and kSAFETY_SCOPE flag combinations). A `true` return value does not guarantee a successful build.

6.1.2.1 Before (TensorRT 10.x)

```
// isNetworkSupported() checked static safety scope rules
// and could be used as a fast pre-build safety gate.
if (!builder->isNetworkSupported(*network, *config)) {
    // Network was outside the static Minimal Safety Scope.
    return false;
}
auto serialized = builder->buildSerializedNetwork(*network, *config);
```

6.1.2.2 After (TensorRT 11.x)

```
// isNetworkSupported() checks only architectural constraints (flag
↪compatibility,
// hybrid DLA/GPU mode). It does NOT validate safety scope.
// Use buildSerializedNetwork() for a definitive safety determination.
if (!builder->isNetworkSupported(*network, *config)) {
    // Network has an invalid configuration (e.g., incompatible builder flags).
    return false;
}
auto serialized = builder->buildSerializedNetwork(*network, *config);
if (!serialized) {
```

(continues on next page)

(continued from previous page)

```
    // Build failed; build-time safety checks rejected the network.
    // Inspect error log for layer-level traceback.
    return false;
}
```

6.1.2.3 Summary of Changes

- `isNetworkSupported()` no longer performs per-layer safety scope checks.
- If you relied on `isNetworkSupported()` as a definitive safety gate, switch to `buildSerializedNetwork()` for a complete build-time determination.
- Tensor volume limit and boolean tensor checks have been removed from `isNetworkSupported()` (these are now validated at build-time).

6.1.3. Interpreting Build-Time Safety Build Failures

Because all safety scope enforcement now occurs during engine building, build failures manifest as build-time errors rather than pre-build scope rejections. When a network is outside the certified scope, the builder reports a `No tactic available` error that traces back to the originating layer.

Review your error handling code to process these build-time messages:

```
auto serialized = builder->buildSerializedNetwork(*network, *config);
if (!serialized) {
    // Examine the error log for messages such as:
    //   "Autotuner: no tactics to implement operation: ... layers=[ONNX Layer: <LayerName>]"
    // Use this layer name to identify the unsupported operation and consult the
    // TensorRT Safety Delta Document for known limitations relative to standard scope.
}
```

6.1.4. Removed `trtexec` Flag `--restricted`

Warning

The `--restricted` flag has been **removed** in TensorRT 11.x. Using it will cause `trtexec` to exit with an error.

The `--restricted` `trtexec` flag, which previously enabled pre-build safety scope validation during engine builds, has been removed. Safety restrictions can no longer be applied during standard engine building. Remove `--restricted` from any build scripts or CI pipelines that reference it.

6.2. Kernel Checker Tool Improvements

Important

This section is only applicable when using the TensorRT 11.x safety runtime, which is only available on NVIDIA DriveOS 7.x.

The TensorRT kernel checker tool adds new checks to validate kernels in MLIR form, in addition to the existing checks that validate kernels in CUDA C++ form. Refer to the NVIDIA Deep Learning Inference SEooC 2.2 Safety Tools Manual V0.1 for more information.

6.3. Recommendation to Use Safety Proxy for Early Bring-Up

For safety workflows, begin developing with the TensorRT safety proxy runtime on x86 for early bring-up rather than with the TensorRT standard runtime. The standard and safety runtime APIs differ significantly, as described in the *Migrating Safety Runtime Code from TensorRT 8.x to 10.x* section.

6.4. Deprecated `trtexec_safe` Flags and Replacements

The following `trtexec_safe` flags have been deprecated in 11.x but are still accepted. Each entry shows the deprecated flag and its replacement.

`--useCudaGraph`
: Enabled by default; flag accepted but has no effect. A deprecation warning is issued. Use `--noCudaGraph` to disable CUDA graph usage.

`--separateProfileRun`
: Always enabled: flag accepted but has no effect. A deprecation warning is issued. This flag will be removed in a future release.

6.5. Adapting to Auxiliary CUDA Stream Support

Important

This section is only applicable when using the TensorRT 11.x safety runtime, which is only available on NVIDIA DriveOS 7.x.

The TensorRT 11.x safety runtime now supports auxiliary CUDA streams, lifting the TensorRT 10.x restriction that forced every safety engine to execute on a single stream. By default, the builder may produce engines that require one or more auxiliary streams; when it does, the application is responsible for allocating, registering, and destroying those streams at runtime.

Warning

Existing TensorRT 10.x safety application code that loads a serialized engine and calls `executeAsync()` without first registering auxiliary streams may fail at runtime if the engine was built with `maxAuxStreams > 0`.

There are two migration paths:

- **Preserve 10.x single-stream behavior** by setting `maxAuxStreams` to `0` at build time. No runtime code changes are required.
- **Adopt multi-stream execution** by querying the engine's auxiliary-stream count and registering streams before the first `executeAsync()` call.

Note

The standard (non-safety) runtime documents the equivalent APIs in Within-Inference Multi-Streaming . The key safety-specific differences are:

- The application must allocate and register auxiliary streams; the safety runtime does not auto-create them.
- The runtime APIs are on `nvinfer2::safe::ITRTGraph` (not `IExecutionContext`) and live in `NvInferSafeRuntime.h`.

6.5.1. Preserving 10.x Single-Stream Behavior

When `maxAuxStreams` is omitted at build time, the builder may select a non-zero value using internal heuristics. To guarantee single-stream behavior across releases, set the flag explicitly to `0` at build time:

```
auto config = builder->createBuilderConfig();
config->setMaxAuxStreams(0);  // produces a single-stream engine
// ... rest of build configuration ...
```

The same flag is available in `trtexec`:

```
trtexec --onnx=model.onnx --maxAuxStreams=0 ...
```

6.5.2. Adopting Auxiliary Streams

If your workflow allows multi-stream engines, the aux-stream-specific steps to add to your runtime code are:

1. Query the number of auxiliary streams the engine expects (after creating the safety graph from the serialized engine).
2. Allocate that many CUDA streams with the `cudaStreamNonBlocking` flag.
3. Register the streams using `setAuxStreams()` before the first call to `executeAsync()`. This is an INIT-phase operation (before any inference launches).
4. Destroy the streams after the final `sync()` call.

If `getNbAuxStreams()` returns `0`, the engine is single-stream and no auxiliary-stream handling is required - you can skip the allocation and registration steps.

The relevant API lives in namespace `nvinfer2::safe` in `NvInferSafeRuntime.h`:

```
// 1. Create the safety graph from the serialized engine.
nvinfer2::safe::ITRTGraph* graph = nullptr;
nvinfer2::safe::createTRTGraph(graph, planData, planSize, recorder, true);

// 2. Query the number of auxiliary streams this engine requires.
int32_t nbAuxStreams = 0;
graph->getNbAuxStreams(nbAuxStreams);

// 3. Allocate that many non-blocking streams. The application owns
//    the lifetime of these streams.
std::vector<cudaStream_t> auxStreams(nbAuxStreams);
for (int32_t i = 0; i < nbAuxStreams; ++i)
{
   cudaStreamCreateWithFlags(&auxStreams[i], cudaStreamNonBlocking);
}

// 4. Register the streams BEFORE the first executeAsync call.
//    The count must exactly match getNbAuxStreams().
graph->setAuxStreams(auxStreams.data(), nbAuxStreams);

// 5. Create the main inference stream.
cudaStream_t mainStream;
cudaStreamCreateWithFlags(&mainStream, cudaStreamNonBlocking);

// 6. Run inference.
graph->executeAsync(mainStream);

// 7. Wait for the full inference to complete.
graph->sync();

// 8. Cleanup — must occur AFTER the final sync().
for (auto s : auxStreams)
{
   cudaStreamDestroy(s);
}
cudaStreamDestroy(mainStream);
nvinfer2::safe::destroyTRTGraph(graph);
```

6.5.3. Using Debug Overlay Filesystem for Safe Engine Building on QNX Safety

Starting in TensorRT 11.0.1, components of the TensorRT safety builder are shipped in the debug overlay filesystem on QNX Safety. Specifically, the `timing_server` executable and certain dependent libraries are mounted in the debug overlay filesystem in the `/nvidia_overlay` directory rather than `/usr/libnvidia`, where they previously shipped. Other platforms, such as QNX Standard and x86, are unchanged.

To mount the debug overlay, specify `debug_sr` as the platform configuration table (PCT) argument to

`bind_partitions`, as shown in the *TensorRT 11.0.1 Installation Guide for DriveOS 7.2.5*:

```
./make/bind_partitions -b <board_variant> qnx -p debug_sr
```

The debug overlay is not needed for production runtime. The `libnvinfer_safe.so*` libraries are still available at `/usr/libnvidia`.

Similarly, the `--remoteAutoTuningConfig` flag to `trtexec` should refer to `/nvidia_overlay`:

```
--remoteAutoTuningConfig="ssh://root:root@${TARGET_IP}:22?remote_exec_path=/
↪nvidia_overlay/timing_server&remote_lib_path=/nvidia_overlay:/usr/libnvidia&
↪dump_remote_stdout=on&dump_remote_stderr=on&verbose=on"
```

Chapter 7. Migrating `IEngineInspector` Usage from TensorRT 10.x to 11.x

If your application parses the JSON output of `IEngineInspector::getLayerInformation()` or `IEngineInspector::getEngineInformation()`, the output schema has changed in TensorRT 11.x. Update your parsing logic as described below.

For general Engine Inspector usage and API details, refer to Engine Inspector.

7.1. Bindings Field Replaced by I/O Tensors

The `Bindings` field, which was a flat list of I/O tensor names as strings, has been removed. It is replaced by the new `I/O Tensors` field, which is a list of JSON objects. Each object contains structured information about the tensor, including its name, I/O mode, data type, dimensions, memory location, and whether it is a shape inference I/O.

7.1.1. Before (TensorRT 10.x)

```
{
  "Bindings": [
    "input_0",
    "output_0"
  ]
}
```

7.1.2. After (TensorRT 11.x)

```
{
   "I/O Tensors": [
     {
       "Name": "input_0",
       "IOMode": "Input",
       "DataType": "Float",
       "Dimensions": [1, 3, 224, 224],
```

(continues on next page)

(continued from previous page)

```
      "Location": "Device",
      "IsShapeInferenceIO": false,
      "ProfileInfo": [
      {
        "MinShape": [1,3,224,224],
        "OptShape": [1,3,224,224],
        "MaxShape": [1,3,224,224],
        "Format": "Row major linear FP32 format (kLINEAR)"
      }]
    },
    {
      "Name": "output_0",
      "IOMode": "Output",
      "DataType": "Float",
      "Dimensions": [1, 1000],
      "Location": "Device",
      "IsShapeInferenceIO": false,
      "ProfileInfo": [
      {
        "Format": "Row major linear FP32 format (kLINEAR)"
      }]
    }
  ]
}
```

7.2. Format / DataType Field Split

In layer-level information, the combined `Format/DataType` field has been removed and replaced with two separate fields: `Format` and `DataType`. This allows you to read the data type directly without parsing the format string.

7.2.1. TensorRT 10.x Output (Before)

```
{
  "Name": "Conv_0",
  "Inputs": [
    {
      "Name": "input_0",
      "Format/DataType": "Row major linear FP32"
    }
  ]
}
```

7.2.2. TensorRT 11.x Output (After)

```
{
   "Name": "Conv_0",
```

(continues on next page)

(continued from previous page)

```
  "Inputs": [
    {
      "Name": "input_0",
      "Datatype": "Float",
      "Format": "(Linear) Row major linear format"
    }
  ]
}
```

Chapter 8. Migrating TensorRT from 10.x to 11.x on NVIDIA DriveOS

This page describes platform-specific migration considerations for TensorRT on NVIDIA DriveOS when upgrading from TensorRT 10.x to 11.x. For API-level migration details, refer to the *C++*, *Python*, *trtexec*, and *Safety Runtime* migration pages.

Important

TensorRT 11.x is supported on DriveOS 7.2.5 and later. DriveOS 7.0 customers on DRIVE Orin remain on TensorRT 10.x, which is maintained on a separate branch. If you are running DriveOS 7.0 with TensorRT 10.x, no migration is required.

8.1. Platform Eligibility

Platform	TensorRT Version	Notes
DRIVE Thor (DriveOS 7.2.5+)	11.x	Full upgrade path. Follow the migration steps on this page.
DRIVE Orin (DriveOS 7.0)	10.x (maintained branch)	Hard-forked TensorRT 10.x branch. No upgrade to 11.x.
DRIVE Orin (DriveOS 7.3)	10.x	Out of scope for TensorRT 11.x in this release.

8.2. Who Should Migrate

You need to take action if your DriveOS application does any of the following:

- Uses implicit INT8 quantization (`BuilderFlag::kINT8` with `IInt8Calibrator`, or `--int8` with `trtexec`).
- Relies on weakly typed builder flags (`BuilderFlag::kFP16`, `BuilderFlag::kBF16`, `BuilderFlag::kFP8`, or equivalent flags such as `--fp16` with `trtexec`).

- Uses `IPluginV2DynamicExt` or `IPluginCreator`.
- References removed tactic sources (`kCUBLAS`, `kCUBLAS_LT`, `kCUDNN`).

If your application already uses strong typing, explicit quantization (Q/DQ nodes), and `IPluginV3`, the upgrade to TensorRT 11.x should require minimal code changes beyond recompilation.

8.3. Migration Checklist

1. **Migrate from weak typing to strong typing.** TensorRT 11.x removes all precision-enabling builder flags. Use ModelOpt AutoCast to convert ONNX models to mixed precision before building. Refer to the *Strongly Typed Networks* section for details.
2. **Migrate from implicit INT8 to explicit quantization.** The `IInt8Calibrator` class and all subclasses have been removed. Use ModelOpt Quantization to produce models with Q/DQ nodes. Refer to the *C++* or *Python* migration page for before/after examples.
3. **Migrate plugins from V2 to V3.** `IPluginV2DynamicExt` and `IPluginCreator` have been deprecated. All plugins must use `IPluginV3` with `IPluginCreatorV3One`. Refer to the *C++ plugin migration* section for a complete NonZero plugin example.
4. **Update V2 API calls to V2 replacements.** Several methods (`getDeviceMemorySize`, `setWeightStreamingBudget`, `setDeviceMemory`) have been replaced with V2 versions that use `int64_t` or accept additional parameters. Refer to the *V2 API replacements table*.
5. **Update safety runtime code (if applicable).** Pre-build safety scope validation has been removed. Build failures now surface as tactic errors with layer-level traceback. Refer to the *Safety Runtime migration* page.
6. **Remove ``–restricted`` from ``trtexec`` scripts.** The `--restricted` flag has been removed in TensorRT 11.x. Remove it from any build scripts or CI pipelines.
7. **Rebuild all engines.** Engines built with TensorRT 10.x using the `BuilderFlag::kVERSION_COMPATIBLE` setting are forward-compatible with the TensorRT 11.x runtime. However, to take advantage of 11.x optimizations and new features, rebuild engines with the 11.x builder.

8.4. Version Compatibility on DriveOS

TensorRT 11.x maintains forward compatibility with TensorRT 10.x engines:

- Engines built with TensorRT 10.x can run on the TensorRT 11.x runtime.
- Engines built with TensorRT 8.x or 9.x are **not** compatible with the TensorRT 11.x runtime. TensorRT 8.x uses CUDA 11.x, which is not supported by TensorRT 11.x.
- Backward compatibility is maintained within a major release. For example, a version-compatible engine built with TensorRT 11.5 runs on TensorRT 11.5 or any later 11.x release, but not on TensorRT 11.0 through 11.4, and not on TensorRT 10.x.

If your DriveOS deployment currently uses TensorRT 8.x, you must first migrate to TensorRT 10.x using the *8.x to 10.x appendix*, then follow this guide for the 10.x to 11.x migration.

8.5. CUDA Dependency Changes

TensorRT 11.x requires CUDA 13.3 or later. Verify that your DriveOS BSP includes a compatible CUDA toolkit before upgrading. TensorRT 11.x drops support for CUDA 11.x, which means TensorRT engines built against CUDA 11.x cannot run on the TensorRT 11.x runtime.

8.6. DLA Considerations

DLA is not supported in TensorRT 11.0; DLA support will be re-introduced in a later minor version update.

8.7. Recommendation for Production Deployments

Tip

For automotive customers already in production on non-safety platforms prior to the TensorRT 11.x release, NVIDIA recommends remaining on the TensorRT 10.x branch to ensure optimal stability and performance. Migrate to 11.x when your next development cycle begins.

For safety customers who must upgrade to 11.x, the TensorRT team provides mitigation plans for all removed APIs and will work directly with customers to ensure a smooth transition. Contact your NVIDIA representative for assistance.

Chapter 9. Migrating TensorRT from 10.x to 11.x on Jetson/JetPack

This page describes platform-specific migration considerations for TensorRT on NVIDIA Jetson platforms (JetPack) when upgrading from TensorRT 10.x to 11.x. For API-level migration details, refer to the *C++*, *Python*, and *trtexec* migration pages.

9.1. Platform Eligibility

Platform	TensorRT Version	Notes
Jetson Thor (JetPack 7.x)	11.x	Full upgrade path. Follow the migration steps on this page.
Jetson Orin (JetPack 6.x)	10.x or 11.x	Optional but recommended upgrade path.

9.2. Who Should Migrate

You need to take action if your Jetson application does any of the following:

- Uses implicit INT8 quantization (`BuilderFlag::kINT8` with `IInt8Calibrator`, or `--int8` with `trtexec`).
- Relies on weakly typed builder flags (`BuilderFlag::kFP16`, `BuilderFlag::kBF16`, `BuilderFlag::kFP8`, or equivalent flags such as `--fp16` with `trtexec`).
- Uses `IPluginV2DynamicExt` or `IPluginCreator`.
- References removed tactic sources (`kCUBLAS`, `kCUBLAS_LT`, `kCUDNN`).

If your application already uses strong typing, explicit quantization (Q/DQ nodes), and `IPluginV3`, the upgrade to TensorRT 11.x should require minimal code changes beyond recompilation.

9.3. Migration Checklist

1. **Migrate from weak typing to strong typing.** TensorRT 11.x removes all precision-enabling builder flags. Use ModelOpt AutoCast to convert ONNX models to mixed precision before building. Refer to the *Strongly Typed Networks* section for details.
2. **Migrate from implicit INT8 to explicit quantization.** The `IInt8Calibrator` class and all subclasses have been removed. Use ModelOpt Quantization to produce models with Q/DQ nodes. Refer to the *C++* or *Python* migration page for before/after examples.
3. **Migrate plugins from V2 to V3.** `IPluginV2DynamicExt` and `IPluginCreator` have been deprecated. All plugins must use `IPluginV3` with `IPluginCreatorV3One`. Refer to the *C++ plugin migration* section for a complete NonZero plugin example.
4. **Update V2 API calls to V2 replacements.** Several methods (`getDeviceMemorySize`, `setWeightStreamingBudget`, `setDeviceMemory`) have been replaced with V2 versions that use `int64_t` or accept additional parameters. Refer to the *V2 API replacements table*.
5. **Rebuild all engines.** Engines built with TensorRT 10.x using the `BuilderFlag::kVERSION_COMPATIBLE` setting are forward-compatible with the TensorRT 11.x runtime. However, to take advantage of 11.x optimizations and new features, rebuild engines with the 11.x builder.

9.4. Version Compatibility on Jetson

TensorRT 11.x maintains forward compatibility with TensorRT 10.x engines:

- Engines built with TensorRT 10.x can run on the TensorRT 11.x runtime.
- Engines built with TensorRT 8.x or 9.x are **not** compatible with the TensorRT 11.x runtime. TensorRT 8.x uses CUDA 11.x, which is not supported by TensorRT 11.x.
- Backward compatibility is maintained within a major release. For example, a version-compatible engine built with TensorRT 11.5 runs on TensorRT 11.5 or any later 11.x release, but not on TensorRT 11.0 through 11.4, and not on TensorRT 10.x.

If your Jetson deployment currently uses TensorRT 8.x, you must first migrate to TensorRT 10.x using the *8.x to 10.x appendix*, then follow this guide for the 10.x to 11.x migration.

9.5. CUDA and JetPack Dependency Changes

TensorRT 11.x requires CUDA 13.2 Update 1 or later. Verify that your JetPack version includes a compatible CUDA toolkit before upgrading. Refer to the JetPack release notes for supported CUDA versions.

TensorRT 11.x drops support for CUDA 11.x. Engines built against CUDA 11.x cannot run on the TensorRT 11.x runtime.

9.6. DLA Considerations

DLA is not supported in TensorRT 11.0; DLA support will be re-introduced in a later minor version update.

9.7. Cross-Compilation

If you cross-compile TensorRT applications for Jetson on an x86 host, ensure that both the host TensorRT SDK and the target JetPack installations are upgraded to 11.x. Mixing TensorRT 10.x headers on the host with 11.x libraries on the target (or vice versa) will cause compilation or linker errors due to removed APIs.

9.8. Recommendation for Edge Deployments

Tip

For Jetson applications in production, NVIDIA recommends testing the TensorRT 11.x migration in a staging environment before deploying to production devices. Verify model accuracy, inference latency, and memory usage against your TensorRT 10.x baseline.

Customers still using TensorRT 10.x should begin migrating to strong typing, explicit quantization, and `IPluginV3` to reduce the scope of changes when upgrading. The TensorRT team provides mitigation plans for all removed APIs.

Chapter 10. Appendix: Migrating from TensorRT 8.x to 10.x

This appendix explains how to migrate from NVIDIA TensorRT 8.x to 10.x.

If you are unfamiliar with these changes, refer to our sample code for clarification.

10.1. How to Use This Appendix

Select the migration surface that matches what you ship:

Python Integration
: Migrate Python inference and engine-build flows, and review Python APIs added and removed in 10.x.

C++ Integration
: Migrate C++ inference and engine-build flows, including 64-bit dimension updates, and review C++ APIs and plugins removed with replacements.

trtexec Command-Line Usage
: Migrate `trtexec` command lines, and review removed or deprecated flag replacements.

Safety Runtime
: Migrate safety runtime code paths, and review removed safety C++ APIs with replacements.

10.1.1. Suggested Reading Order

1. Start with *Python Integration* or *C++ Integration* for application code paths (read both only if you maintain both stacks).
2. Add *trtexec Command-Line Usage* when your workflow depends on CLI flags or scripts.
3. Read *Safety Runtime* when you target the safety runtime instead of the standard runtime.

Attention

TensorRT 10.x safety proxy runtime support on QNX Standard (without QNX Safety support) is available in limited, alpha-quality capacity in this release. Full functional safety runtime support, including QNX Safety, is planned for a future sideband drop on top of NVIDIA DriveOS 7.2.5.

10.1.2. Next Steps

After completing the 8.x-to-10.x migration, continue to *Migrating from TensorRT 10.x to 11.x* for the next set of API and tooling changes.

Chapter 11. Appendix: Migrating Python Code from TensorRT 8.x to 10.x

This page describes how to update Python code when you migrate from TensorRT 8.x to 10.x: paired examples for the name-based tensor API (buffers and I/O), `enqueueV3` execution, and `build_serialized_network` for engine build, followed by lists of Python APIs added and removed in 10.x.

Important

The Python API is not supported on QNX. If you are targeting QNX (Standard or Safety), use the C++ API; refer to *Appendix: Migrating C++ Code from TensorRT 8.x to 10.x*.

Attention

DriveOS 7.2.5 scope: DLA-related rename entries documented later on this page (such as `EngineCapability.kSAFE_DLA` → `EngineCapability.DLA_STANDALONE`) are listed for API completeness only. **DLA is not supported in TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5.** For current DriveOS guidance, refer to *Migrating TensorRT from 10.x to 11.x on NVIDIA DriveOS*.

11.1. Migrating I/O Buffer Allocation to Named Tensors

TensorRT 10.x replaces the binding-based API with a name-based tensor API. Use `num_io_tensors` and `get_tensor_name()` to iterate over I/O tensors, and `get_tensor_mode()` to check whether a tensor is an input or output.

Before (TensorRT 8.x)

```
def allocate_buffers(self, engine):
'''
Allocates all buffers required for an engine; that is, host and device inputs
```

(continues on next page)

(continued from previous page)

```
    and outputs.
'''
inputs = []
outputs = []
bindings = []
stream = cuda.Stream()

# binding is the name of input/output
for binding in the engine:
    size = trt.volume(engine.get_binding_shape(binding)) * engine.max_batch_size
    dtype = trt.nptype(engine.get_binding_dtype(binding))

    # Allocate host and device buffers
    host_mem = cuda.pagelocked_empty(size, dtype) # page-locked memory buffer (will not be swapped to disk)
    device_mem = cuda.mem_alloc(host_mem.nbytes)

    # Append the device buffer address to device bindings.
    # When cast to int, it is a linear index into the context's memory (like memory address).
    bindings.append(int(device_mem))

    # Append to the appropriate input/output list.
    if engine.binding_is_input(binding):
        inputs.append(self.HostDeviceMem(host_mem, device_mem))
    else:
        outputs.append(self.HostDeviceMem(host_mem, device_mem))

return inputs, outputs, bindings, stream
```

After (TensorRT 10.x)

```
def allocate_buffers(self, engine):
'''
Allocates all buffers required for an engine; that is, host and device inputs and outputs.
'''
inputs = []
outputs = []
bindings = []
stream = cuda.Stream()

for i in range(engine.num_io_tensors):
    tensor_name = engine.get_tensor_name(i)
    size = trt.volume(engine.get_tensor_shape(tensor_name))
    dtype = trt.nptype(engine.get_tensor_dtype(tensor_name))

    # Allocate host and device buffers
    host_mem = cuda.pagelocked_empty(size, dtype) # page-locked memory buffer (will not be swapped to disk)
```

(continues on next page)

(continued from previous page)

```
    device_mem = cuda.mem_alloc(host_mem.nbytes)

    # Append the device buffer address to device bindings.
    # When cast to int, it is a linear index into the context's memory (like
↪memory address).
    bindings.append(int(device_mem))

    # Append to the appropriate input/output list.
    if engine.get_tensor_mode(tensor_name) == trt.TensorIOMode.INPUT:
        inputs.append(self.HostDeviceMem(host_mem, device_mem))
    else:
        outputs.append(self.HostDeviceMem(host_mem, device_mem))

return inputs, outputs, bindings, stream
```

11.1.1. Summary of Changes

- Changed from binding-based iteration to name-based API using `num_io_tensors` and `get_tensor_name()`
- Replaced `binding_is_input()` with `get_tensor_mode()` to check I/O mode

Note

The `HostDeviceMem` helper class used in the examples above is a simple container that pairs host (CPU) and device (GPU) memory allocations. It is not part of the TensorRT API. A minimal implementation is:

```
from collections import namedtuple

HostDeviceMem = namedtuple("HostDeviceMem", ["host", "device"])
```

11.2. Migrating from enqueueV2 to enqueueV3 (Python)

The examples below show TensorRT 8.x first, then TensorRT 10.x, for the same inference task. In TensorRT 10.x, enqueueV3 replaces enqueueV2: call `set_tensor_address` for each I/O tensor before `execute_async_v3`, as shown in the After tab.

Before (TensorRT 8.x)

```
# Allocate device memory for inputs.
d_inputs = [cuda.mem_alloc(input_nbytes) for binding in range(input_num)]

# Allocate device memory for outputs.
h_output = cuda.pagelocked_empty(output_nbytes, dtype=np.float32)
d_output = cuda.mem_alloc(h_output.nbytes)
```

(continues on next page)

(continued from previous page)

```

# Transfer data from host to device.
cuda.memcpy_htod_async(d_inputs[0], input_a, stream)
cuda.memcpy_htod_async(d_inputs[1], input_b, stream)
cuda.memcpy_htod_async(d_inputs[2], input_c, stream)

# Run inference
context.execute_async_v2(bindings=[int(d_inp) for d_inp in d_inputs] + [int(d_
↪output)], stream_handle=stream.handle)

# Synchronize the stream
stream.synchronize()
```

After (TensorRT 10.x)

```
# Allocate device memory for inputs.
d_inputs = [cuda.mem_alloc(input_nbytes) for binding in range(input_num)]

# Allocate device memory for outputs.
h_output = cuda.pagelocked_empty(output_nbytes, dtype=np.float32)
d_output = cuda.mem_alloc(h_output.nbytes)

# Transfer data from host to device.
cuda.memcpy_htod_async(d_inputs[0], input_a, stream)
cuda.memcpy_htod_async(d_inputs[1], input_b, stream)
cuda.memcpy_htod_async(d_inputs[2], input_c, stream)

# Setup tensor address
bindings = [int(d_inputs[i]) for i in range(3)] + [int(d_output)]

for i in range(engine.num_io_tensors):
    context.set_tensor_address(engine.get_tensor_name(i), bindings[i])

# Run inference
context.execute_async_v3(stream_handle=stream.handle)

# Synchronize the stream
stream.synchronize()
```

11.2.1. Summary of Changes

- Added explicit tensor address setup using `set_tensor_address()` with tensor names
- Changed from `execute_async_v2()` to `execute_async_v3()`
- The bindings parameter is no longer passed to `execute_async_v3()`; tensor addresses must be set beforehand

11.3. Migrating Engine Builds to `build_serialized_network`

The examples below show TensorRT 8.x first, then TensorRT 10.x, for the same engine build path. In TensorRT 10.x, `build_serialized_network()` is the standard build path and is always available. Omit the `build_engine()` / `serialize()` fallback from the 8.x sample, and check for a `None` return instead of catching `AttributeError`, as shown in the After tab.

Before (TensorRT 8.x)

```
engine_bytes = None
try:
    engine_bytes = self.builder.build_serialized_network(self.network, self.
↪config)
except AttributeError:
    engine = self.builder.build_engine(self.network, self.config)
    engine_bytes = engine.serialize()
    del engine
assert engine_bytes
```

After (TensorRT 10.x)

```
engine_bytes = self.builder.build_serialized_network(self.network, self.
↪config)
if engine_bytes is None:
    log.error("Failed to create engine")
    sys.exit(1)
```

11.3.1. Summary of Changes

- The fallback to `build_engine()` and `serialize()` is no longer needed in TensorRT 10.x
- `build_serialized_network()` is now the standard method and always available
- Error handling should check for `None` return value instead of catching `AttributeError`

11.4. Python APIs Added in 10.x

The following Python APIs have been added in TensorRT 10.x to support new features and improved functionality.

11.4.1. Types

- `APILanguage`
- `ExecutionContextAllocationStrategy`
- `IGpuAsyncAllocator`

- `InterfaceInfo`
- `IPluginResource`
- `IPluginV3`
- `IStreamReader`
- `IVersionedInterface`

11.4.2. Methods and Properties

- `ICudaEngine.is_debug_tensor()`
- `ICudaEngine.minimum_weight_streaming_budget`
- `ICudaEngine.streamable_weights_size`
- `ICudaEngine.weight_streaming_budget`
- `IExecutionContext.get_debug_listener()`
- `IExecutionContext.get_debug_state()`
- `IExecutionContext.set_all_tensors_debug_state()`
- `IExecutionContext.set_debug_listener()`
- `IExecutionContext.set_tensor_debug_state()`
- `IExecutionContext.update_device_memory_size_for_shapes()`
- `IGpuAllocator.allocate_async()`
- `IGpuAllocator.deallocate_async()`
- `INetworkDefinition.add_plugin_v3()`
- `INetworkDefinition.is_debug_tensor()`
- `INetworkDefinition.mark_debug()`
- `INetworkDefinition.unmark_debug()`
- `IPluginRegistry.acquire_plugin_resource()`
- `IPluginRegistry.all_creators`
- `IPluginRegistry.deregister_creator()`
- `IPluginRegistry.get_creator()`
- `IPluginRegistry.register_creator()`
- `IPluginRegistry.release_plugin_resource()`

11.5. Removed Python APIs and Replacements

Warning

The APIs listed below have been **removed** in TensorRT 10.x and will cause runtime errors if called. Review each entry for its replacement before upgrading.

Note

The `EngineCapability.kSAFE_DLA` → `EngineCapability.DLA_STANDALONE` rename in the table below is documented for API completeness only. Refer to the **DriveOS 7.2.5 scope** attention notice at the top of this page.

Removed API	Replacement
`BuilderFlag.ENABLE_TACTIC_HEURISTIC`	Builder optimization level 2.
`BuilderFlag.STRICT_TYPES`	Use `BuilderFlag.DIRECT_IO`, `BuilderFlag.PREFER_PRECISION_CONSTRAINTS`, and `BuilderFlag.REJECT_EMPTY_ALGORITHMS`.
`EngineCapability.DEFAULT`	`EngineCapability.STANDARD`
`EngineCapability.kSAFE_DLA`	`EngineCapability.DLA_STANDALONE`
`EngineCapability.SAFE_GPU`	`EngineCapability.SAFETY`
`IAlgorithmIOInfo.tensor_format`	Strides, data type, and vectorization information are sufficient to identify tensor formats uniquely.
`IBuilder.max_batch_size`	Implicit batch support was removed.
`IBuilderConfig.max_workspace_size`	`IBuilderConfig.set_memory_pool_limit()` or `get_memory_pool_limit()` with `MemoryPoolType.WORKSPACE`
`IBuilderConfig.min_timing_iterations`	`IBuilderConfig.avg_timing_iterations`
`ICudaEngine.binding_is_input()`	`ICudaEngine.get_tensor_mode()`
`ICudaEngine.get_binding_bytes_per_compo`	`ICudaEngine.get_tensor_bytes_per_component()`
`ICudaEngine.get_binding_components_per`	`ICudaEngine.get_tensor_components_per_element()`
`ICudaEngine.get_binding_dtype()`	`ICudaEngine.get_tensor_dtype()`
`ICudaEngine.get_binding_format()`	`ICudaEngine.get_tensor_format()`
`ICudaEngine.get_binding_format_desc()`	`ICudaEngine.get_tensor_format_desc()`
`ICudaEngine.get_binding_index()`	No name-based equivalent replacement.
`ICudaEngine.get_binding_name()`	No name-based equivalent replacement.
`ICudaEngine.get_binding_shape()`	`ICudaEngine.get_tensor_shape()`
`ICudaEngine.get_binding_vectorized_dim`	`ICudaEngine.get_tensor_vectorized_dim()`
`ICudaEngine.get_location()`	`ITensor.location`
`ICudaEngine.get_profile_shape()`	`ICudaEngine.get_tensor_profile_shape()`
`ICudaEngine.get_profile_shape_input()`	`ICudaEngine.get_tensor_profile_values()`
`ICudaEngine.has_implicit_batch_dimensi`	Implicit batch is no longer supported.
`ICudaEngine.is_execution_binding()`	No name-based equivalent replacement.
`ICudaEngine.is_shape_binding()`	`ICudaEngine.is_shape_inference_io()`

continues on next page

Table 11.1 – continued from previous page

Removed API	Replacement
ICudaEngine.max_batch_size()	Implicit batch is no longer supported.
ICudaEngine.num_bindings()	ICudaEngine.num_io_tensors()
IExecutionContext.get_binding_shape()	IExecutionContext.get_tensor_shape()
IExecutionContext.get_strides()	IExecutionContext.get_tensor_strides()
IExecutionContext.set_binding_shape()	IExecutionContext.set_input_shape()
IFullyConnectedLayer	IMatrixMultiplyLayer
INetworkDefinition.add_convolution()	INetworkDefinition.add_convolution_nd()
INetworkDefinition.add_deconvolution()	INetworkDefinition.add_deconvolution_nd()
INetworkDefinition.add_fully_connected()	INetworkDefinition.add_matrix_multiply()
INetworkDefinition.add_padding()	INetworkDefinition.add_padding_nd()
INetworkDefinition.add_pooling()	INetworkDefinition.add_pooling_nd()
INetworkDefinition.add_rnn_v2()	INetworkDefinition.add_loop()
INetworkDefinition.has_explicit_precision	Explicit precision support was removed in 10.0.
INetworkDefinition.has_implicit_batch_dimension	Implicit batch support was removed.
IRNNv2Layer	ILoop
NetworkDefinitionCreationFlag.EXPLICIT_BATCH	Support was removed in 10.0.
NetworkDefinitionCreationFlag.EXPLICIT_PRECISION	Support was removed in 10.0.
PaddingMode.CAFFE_ROUND_DOWN	Caffe support was removed.
PaddingMode.CAFFE_ROUND_UP	Caffe support was removed.
PreviewFeature.DISABLE_EXTERNAL_TACTIC.	External tactics are always disabled for core code.
PreviewFeature.FASTER_DYNAMIC_SHAPES_0	This flag is on by default.
ProfilingVerbosity.DEFAULT	ProfilingVerbosity.LAYER_NAMES_ONLY
ProfilingVerbosity.VERBOSE	ProfilingVerbosity.DETAILED
ResizeMode	Use InterpolationMode. Alias was removed.
SampleMode.DEFAULT	SampleMode.STRICT_BOUNDS
SliceMode	Use SampleMode. Alias was removed.

Chapter 12. Appendix: Migrating C++ Code from TensorRT 8.x to 10.x

This page describes how to update C++ code when you migrate from TensorRT 8.x to 10.x: paired examples for `enqueueV3` and the name-based tensor API, guidance on 64-bit dimensions, and lists of C++ APIs added and removed in 10.x.

Attention

DriveOS 7.2.5 scope: DLA-related rename entries documented later on this page (such as `EngineCapability::kSAFE_DLA` → `EngineCapability::kDLA_STANDALONE`) are listed for API completeness only. **DLA is not supported in TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5.** For current DriveOS guidance, refer to *Migrating TensorRT from 10.x to 11.x on NVIDIA DriveOS*.

12.1. Migrating from enqueueV2 to enqueueV3 (C++)

The examples below show TensorRT 8.x first, then TensorRT 10.x, for the same inference task. In TensorRT 10.x, `enqueueV3` replaces `enqueueV2`: call `setTensorAddress` for each I/O tensor (using names from `getIOTensorName`) before `enqueueV3`, as shown in the After tab.

Before (TensorRT 8.x)

```
// Create RAII buffer manager object.
samplesCommon::BufferManager buffers(mEngine);

auto context = SampleUniquePtr<nvinfer1::IExecutionContext>(mEngine->
↪createExecutionContext());
if (!context)
{
    return false;
}

// Pick a random digit to try to infer.
```

(continues on next page)

(continued from previous page)

```
srand(time(NULL));
int32_t const digit = rand() % 10;

// Read the input data into the managed buffers.
// There should be just 1 input tensor.
ASSERT(mParams.inputTensorNames.size() == 1);

if (!processInput(buffers, mParams.inputTensorNames[0], digit))
{
    return false;
}
// Create a CUDA stream to execute this inference.
cudaStream_t stream;
CHECK(cudaStreamCreate(&stream));

// Asynchronously copy data from host input buffers to device input
buffers.copyInputToDeviceAsync(stream);

// Asynchronously enqueue the inference work
if (!context->enqueueV2(buffers.getDeviceBindings().data(), stream, nullptr))
{
    return false;
}
// Asynchronously copy data from device output buffers to host output buffers.
buffers.copyOutputToHostAsync(stream);

// Wait for the work in the stream to complete.
CHECK(cudaStreamSynchronize(stream));

// Release stream.
CHECK(cudaStreamDestroy(stream));
```

After (TensorRT 10.x)

```
// Create RAII buffer manager object.
samplesCommon::BufferManager buffers(mEngine);

auto context = SampleUniquePtr<nvinfer1::IExecutionContext>(mEngine->
↪createExecutionContext());
if (!context)
{
    return false;
}

for (int32_t i = 0, e = mEngine->getNbIOTensors(); i < e; i++)
{
    auto const name = mEngine->getIOTensorName(i);
    context->setTensorAddress(name, buffers.getDeviceBuffer(name));
}

// Pick a random digit to try to infer.
```

(continues on next page)

(continued from previous page)

```
srand(time(NULL));
int32_t const digit = rand() % 10;

// Read the input data into the managed buffers.
// There should be just 1 input tensor.
ASSERT(mParams.inputTensorNames.size() == 1);

if (!processInput(buffers, mParams.inputTensorNames[0], digit))
{
    return false;
}
// Create a CUDA stream to execute this inference.
cudaStream_t stream;
CHECK(cudaStreamCreate(&stream));

// Asynchronously copy data from host input buffers to device input
buffers.copyInputToDeviceAsync(stream);

// Asynchronously enqueue the inference work
if (!context->enqueueV3(stream))
{
    return false;
}

// Asynchronously copy data from device output buffers to host output buffers.
buffers.copyOutputToHostAsync(stream);

// Wait for the work in the stream to complete.
CHECK(cudaStreamSynchronize(stream));

// Release stream.
CHECK(cudaStreamDestroy(stream));
```

12.1.1. Summary of Changes

- Added explicit tensor address setup using `setTensorAddress()` with tensor names from `getIOTensorName()`
- Changed from `enqueueV2()` to `enqueueV3()`
- The bindings parameter is no longer passed to `enqueueV3()`; tensor addresses must be set beforehand using `setTensorAddress()`

12.2. Migrating Dims and IShapeLayer to 64-Bit Types

Warning

This is a **breaking ABI change**. Code that bitwise copies to or from `Dims` must be updated for the wider type.

TensorRT 10.x changes the dimension type from `int32_t` to `int64_t`. The dimensions held by Dims changed from `int32_t` to `int64_t`. However, in TensorRT 10.x, TensorRT will generally reject networks that use dimensions exceeding the range of `int32_t`. The tensor type returned by `IShapeLayer` is now `DataType::kINT64`. Use `ICastLayer` to cast the result to the tensor of type `DataType::kINT32` if 32-bit dimensions are required.

Inspect code that bitwise copies to and from Dims to ensure it is correct for `int64_t` dimensions.

12.3. C++ APIs Added in 10.x

The following C++ APIs have been added in TensorRT 10.x to support new features and improved functionality.

12.3.1. Enums

- `ActivationType::kGELU_ERF`
- `ActivationType::kGELU_TANH`
- `BuilderFlag::kREFIT_IDENTICAL`
- `BuilderFlag::kSTRIP_PLAN`
- `BuilderFlag::kWEIGHT_STREAMING`
- `BuilderFlag::kSTRICT_NANS`
- `Datatype::kINT4`
- `LayerType::kPLUGIN_V3`

12.3.2. Types

- `APILanguage`
- `Dims64`
- `ExecutionContextAllocationStrategy`
- `IGpuAsyncAllocator`
- `InterfaceInfo`
- `IPluginResource`

- IPluginV3
- IStreamReader
- IVersionedInterface

12.3.3. Methods and Properties

- getInferLibBuildVersion
- getInferLibMajorVersion
- getInferLibMinorVersion
- getInferLibPatchVersion
- IBuilderConfig::setMaxNbTactics
- IBuilderConfig::getMaxNbTactics
- ICudaEngine::createRefitter
- IcudaEngine::getMinimumWeightStreamingBudget
- IcudaEngine::getStreamableWeightsSize
- ICudaEngine::getWeightStreamingBudget
- IcudaEngine::isDebugTensor
- ICudaEngine::setWeightStreamingBudget
- IExecutionContext::getDebugListener
- IExecutionContext::getTensorDebugState
- IExecutionContext::setAllTensorsDebugState
- IExecutionContext::setDebugListener
- IExecutionContext::setOuputTensorAddress
- IExecutionContext::setTensorDebugState
- IExecutionContext::updateDeviceMemorySizeForShapes
- IGpuAllocator::allocateAsync
- IGpuAllocator::deallocateAsync
- INetworkDefinition::addPluginV3
- INetworkDefinition::isDebugTensor
- INetworkDefinition::markDebug
- INetworkDefinition::unmarkDebug
- IPluginRegistry::acquirePluginResource
- IPluginRegistry::deregisterCreator
- IPluginRegistry::getAllCreators
- IPluginRegistry::getCreator
- IPluginRegistry::registerCreator
- IPluginRegistry::releasePluginResource

12.4. Removed C++ APIs and Replacements

Warning

The APIs listed below have been **removed** in TensorRT 10.x and will cause compilation or linker errors. Review each entry for its replacement before upgrading.

Note

The `EngineCapability::kSAFE_DLA` → `EngineCapability::kDLA_STANDALONE` rename in the table below is documented for API completeness only. Refer to the **DriveOS 7.2.5 scope** attention notice at the top of this page.

Removed API	Replacement
`BuilderFlag::kENABLE_TACTIC_HEURISTIC`	Builder optimization level 2.
`BuilderFlag::kSTRICT_TYPES`[1]	Use `kREJECT_EMPTY_ALGORITHMS`, `kDIRECT_IO`, and `kPREFER_PRECISION_CONSTRAINTS`.
`EngineCapability::kDEFAULT`	`EngineCapability::kSTANDARD`
`EngineCapability::kSAFE_DLA`	`EngineCapability::kDLA_STANDALONE`
`EngineCapability::kSAFE_GPU`	`EngineCapability::kSAFETY`
`IAlgorithm::getAlgorithmIOInfo()`	`IAlgorithm::getAlgorithmIOInfoByIndex()`
`IAlgorithmIOInfo::getTensorFormat()`	Strides, data type, and vectorization information are sufficient to identify tensor formats uniquely.
`IBuilder::buildEngineWithConfig()`	`IBuilder::buildSerializedNetwork()`
`IBuilder::destroy()`	`delete ObjectName`
`IBuilder::getMaxBatchSize()`	Implicit batch support was removed.
`IBuilder::setMaxBatchSize()`	Implicit batch support was removed.
`IBuilderConfig::destroy()`	`delete ObjectName`
`IBuilderConfig::getMaxWorkspaceSize()`	`IBuilderConfig::getMemoryPoolLimit()` with `MemoryPoolType::kWORKSPACE`
`IBuilderConfig::getMinTimingIterations`	`IBuilderConfig::getAvgTimingIterations()`
`IBuilderConfig::setMaxWorkspaceSize()`	`IBuilderConfig::setMemoryPoolLimit()` with `MemoryPoolType::kWORKSPACE`
`IBuilderConfig::setMinTimingIterations`	`IBuilderConfig::setAvgTimingIterations()`
`IConvolutionLayer::getDilation()`	`IConvolutionLayer::getDilationNd()`
`IConvolutionLayer::getKernelSize()`	`IConvolutionLayer::getKernelSizeNd()`

continues on next page

Table 12.1 – continued from previous page

Removed API	Replacement
IConvolutionLayer::getPadding()	IConvolutionLayer::getPaddingNd()
IConvolutionLayer::getStride()	IConvolutionLayer::getStrideNd()
IConvolutionLayer::setDilation()	IConvolutionLayer::setDilationNd()
IConvolutionLayer::setKernelSize()	IConvolutionLayer::setKernelSizeNd()
IConvolutionLayer::setPadding()	IConvolutionLayer::setPaddingNd()
IConvolutionLayer::setStride()	IConvolutionLayer::setStrideNd()
ICudaEngine::bindingIsInput()	ICudaEngine::getTensorIOMode()
ICudaEngine::destroy()	delete ObjectName
ICudaEngine::getBindingBytesPerComponer	ICudaEngine::getTensorBytesPerComponent()
ICudaEngine::getBindingComponentsPerEl	ICudaEngine::getTensorComponentsPerElement()
ICudaEngine::getBindingDataType()	ICudaEngine::getTensorDataType()
ICudaEngine::getBindingDimensions()	ICudaEngine::getTensorShape()
ICudaEngine::getBindingFormat()	ICudaEngine::getTensorFormat()
ICudaEngine::getBindingFormatDesc()	ICudaEngine::getTensorFormatDesc()
ICudaEngine::getBindingIndex()	Name-based methods.
ICudaEngine::getBindingName()	Name-based methods.
ICudaEngine::getBindingVectorizedDim()	ICudaEngine::getTensorVectorizedDim()
ICudaEngine::getLocation()	ITensor::getLocation()
ICudaEngine::getMaxBatchSize()	Implicit batch support was removed.
ICudaEngine::getNbBindings()	ICudaEngine::getNbIOTensors()
ICudaEngine::getProfileDimensions()	ICudaEngine::getProfileShape()
ICudaEngine::getProfileShapeValues()	ICudaEngine::getShapeValues()
ICudaEngine::hasImplicitBatchDimension	Implicit batch support was removed.
ICudaEngine::isExecutionBinding()	No name-based equivalent replacement.
ICudaEngine::isShapeBinding()	ICudaEngine::isShapeInferenceIO()
IDeconvolutionLayer::getKernelSize()	IDeconvolutionLayer::getKernelSizeNd()
IDeconvolutionLayer::getPadding()	IDeconvolutionLayer::getPaddingNd()
IDeconvolutionLayer::getStride()	IDeconvolutionLayer::getStrideNd()
IDeconvolutionLayer::setKernelSize()	IDeconvolutionLayer::setKernelSizeNd()
IDeconvolutionLayer::setPadding()	IDeconvolutionLayer::setPaddingNd()
IDeconvolutionLayer::setStride()	IDeconvolutionLayer::setStrideNd()
IExecutionContext::destroy()	delete ObjectName
IExecutionContext::enqueue()	IExecutionContext::enqueueV3()

continues on next page

Table 12.1 – continued from previous page

Removed API	Replacement
IExecutionContext::enqueueV2()	IExecutionContext::enqueueV3()
IExecutionContext::execute()	IExecutionContext::executeV2()
IExecutionContext::getBindingDimension:	IExecutionContext::getTensorShape()
IExecutionContext::getShapeBinding()	IExecutionContext::getTensorAddress() or getOutputTensorAddress()
IExecutionContext::getStrides()	IExecutionContext::getTensorStrides()
IExecutionContext::setBindingDimension:	IExecutionContext::setInputShape()
IExecutionContext::setInputShapeBindin	IExecutionContext::setInputTensorAddress() or setTensorAddress()
IExecutionContext::setOptimizationProf:	IExecutionContext::setOptimizationProfileAsync()
IFullyConnectedLayer	IMatrixMultiplyLayer
IGpuAllocator::free()	IGpuAllocator::deallocate()
IHostMemory::destroy()	delete ObjectName
INetworkDefinition::addConvolution()	INetworkDefinition::addConvolutionNd()
INetworkDefinition::addDeconvolution()	INetworkDefinition::addDeconvolutionNd()
INetworkDefinition::addFullyConnected(	INetworkDefinition::addMatrixMultiply()
INetworkDefinition::addPadding()	INetworkDefinition::addPaddingNd()
INetworkDefinition::addPooling()	INetworkDefinition::addPoolingNd()
INetworkDefinition::addRNNv2()	INetworkDefinition::addLoop()
INetworkDefinition::destroy()	delete ObjectName
INetworkDefinition::hasExplicitPrecisi	Explicit precision support was removed in 10.0.
INetworkDefinition::hasImplicitBatchDi	Implicit batch support was removed.
IOnnxConfig::destroy()	delete ObjectName
IPaddingLayer::getPostPadding()	IPaddingLayer::getPostPaddingNd()
IPaddingLayer::getPrePadding()	IPaddingLayer::getPrePaddingNd()
IPaddingLayer::setPostPadding()	IPaddingLayer::setPostPaddingNd()
IPaddingLayer::setPrePadding()	IPaddingLayer::setPrePaddingNd()
IPoolingLayer::getPadding()	IPoolingLayer::getPaddingNd()
IPoolingLayer::getStride()	IPoolingLayer::getStrideNd()
IPoolingLayer::getWindowSize()	IPoolingLayer::getWindowSizeNd()
IPoolingLayer::setPadding()	IPoolingLayer::setPaddingNd()
IPoolingLayer::setStride()	IPoolingLayer::setStrideNd()
IPoolingLayer::setWindowSize()	IPoolingLayer::setWindowSizeNd()

continues on next page

Table 12.1 – continued from previous page

Removed API	Replacement
IRefitter::destroy()	delete ObjectName
IResizeLayer::getAlignCorners()	IResizeLayer::getAlignCornersNd()
IResizeLayer::setAlignCorners()	IResizeLayer::setAlignCornersNd()
IRuntime::deserializeCudaEngine(void const* blob, std::size_t size, IPluginFactory* pluginFactory)	Use deserializeCudaEngine with two parameters.
IRuntime::destroy()	delete ObjectName
IRNNv2Layer	ILoop
kNV_TENSORRT_VERSION_IMPL[2]	NV_TENSORRT_VERSION_INT(major, minor, patch) macro: ((major) *10000L + (minor) *100L + (patch) *1L)
NetworkDefinitionCreationFlag::kEXPLICIT_BATCH	Support was removed in 10.0.
NetworkDefinitionCreationFlag::kEXPLICIT_PRECISION	Support was removed in 10.0.
NV_TENSORRT_SONAME_MAJOR	NV_TENSORRT_MAJOR
NV_TENSORRT_SONAME_MINOR	NV_TENSORRT_MINOR
NV_TENSORRT_SONAME_PATCH	NV_TENSORRT_PATCH
getBuilderSafePluginRegistry()	API will not be implemented.
PaddingMode::kCAFFE_ROUND_DOWN	Caffe support was removed.
PaddingMode::kCAFFE_ROUND_UP	Caffe support was removed.
PreviewFeature::kDISABLE_EXTERNAL_TACT	External tactics are always disabled for core code.
PreviewFeature::kFASTER_DYNAMIC_SHAPES	This flag is on by default.
ProfilingVerbosity::kDEFAULT	ProfilingVerbosity::kLAYER_NAMES_ONLY
ProfilingVerbosity::kVERBOSE	ProfilingVerbosity::kDETAILED
ResizeMode	Use InterpolationMode. Alias was removed.
RNNDirection	RNN-related data structures were removed.
RNNGateType	RNN-related data structures were removed.
RNNInputMode	RNN-related data structures were removed.
RNNOperation	RNN-related data structures were removed.
SampleMode::kDEFAULT	SampleMode::kSTRICT_BOUNDS
SliceMode	Use SampleMode. Alias was removed.

[1] When removing enum members (for all enums in this list), we will change the enumeration in the enum to have sequential numbers.

[2] TensorRT version encoding was changed to accommodate a two-digit minor version.

12.5. Removed C++ Plugins and Replacements

Warning

The plugins listed below have been **removed** in TensorRT 10.x. Using them will cause compilation or linker errors. Review each entry for its replacement before upgrading.

Removed Plugin	Replacement
`createAnchorGeneratorPlugin()`	`GridAnchorPluginCreator::createPlugin()`
`createBatchedNMSPlugin()`	`BatchedNMSPluginCreator::createPlugin()`
`createInstanceNormalizationPlugin()`	`InstanceNormalizationPluginCreator::createPlugin()`
`createNMSPlugin()`	`NMSPluginCreator::createPlugin()`
`createNormalizePlugin()`	`NormalizePluginCreator::createPlugin()`
`createPriorBoxPlugin()`	`PriorBoxPluginCreator::createPlugin()`
`createRegionPlugin()`	`RegionPluginCreator::createPlugin()`
`createReorgPlugin()`	`ReorgPluginCreator::createPlugin()`
`createRPNROIPlugin()`	`RPROIPluginCreator::createPlugin()`
`createSplitPlugin()`	`INetworkDefinition::addSlice()`
`struct Quadruple`	Related plugins were removed.

Chapter 13. Appendix: Migrating `trtexec` Usage from TensorRT 8.x to 10.x

This page describes how to update `trtexec` usage when migrating from TensorRT 8.x to 10.x: before/after command examples and lists of removed or deprecated options.

13.1. Migrating `--workspace` and `--minTiming` Options

The examples below show TensorRT 8.x first, then TensorRT 10.x, for the same `trtexec` invocation pattern.

Before (TensorRT 8.x)

```
trtexec \
    --onnx=/path/to/model.onnx \
    --saveEngine=/path/to/engine.trt \
    --optShapes=input:$INPUT_SHAPE \

    --workspace=1024 \
    --minTiming=1
```

After (TensorRT 10.x)

```
trtexec \
    --onnx=/path/to/model.onnx \
    --saveEngine=/path/to/engine.trt \
    --optShapes=input:$INPUT_SHAPE \

    --memPoolSize=workspace:1024
```

13.1.1. Summary of Changes

- `--workspace` flag replaced with `--memPoolSize=workspace:<size>`
- `--minTiming` flag removed (use `--avgTiming` instead)

13.2. Removed `trtexec` Flags and Replacements

Warning

The flags listed below have been **removed** in TensorRT 10.x. Using them will cause `trtexec` to exit with an error. Review each entry for its replacement before upgrading.

Removed Flag	Replacement
`--deploy`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--output`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--model`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--uff`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--uffInput`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--uffNHWC`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--batch`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--maxBatch`	TensorRT 10.x does not support Caffe input, UFF input, and implicit batch dimension mode.
`--minTiming`	`--avgTiming`
`--preview=features`	`disableExternalTacticSourcesForCore0805` or `fasterDynamicShapes0805`
`--workspace=N`	`--memPoolSize=poolspec`
`--explicitPrecision`	Removed (no replacement).
`--nativeInstanceNorm`	Removed (no replacement).
`--heuristic`	`--builderOptimizationLevel=<N>` (where <N> can be 0, 1, or 2)
`--buildOnly`	`--skipInference`

continues on next page

Table 13.1 – continued from previous page

Removed Flag	Replacement
`--nvtxMode`	`--profilingVerbosity`

13.3. Deprecated `trtexec` Flags and Replacements

The following `trtexec` flags have been deprecated. Each entry shows the deprecated flag and its replacement.

Deprecated Flag	Replacement
`--sparsity=force`	Use `polygraphy surgeon prune` to rewrite the weights to a sparsity pattern and then run `--sparsity=enable`.
`--plugins`	`--staticPlugins`
`--preview=profileSharing086`	Enabled by default and has no effect.
`--profilingVerbosity=defaul`	`--profilingVerbosity=layer_names_only`
`--profilingVerbosity=verbos`	`--profilingVerbosity=detailed`
`--streams`	`--infStreams`
`--weightless`	`--stripWeights`

Chapter 14. Appendix: Migrating Safety Runtime Code from TensorRT 8.x to 10.x

This page describes how to update safety runtime code when migrating from TensorRT 8.x to 10.x. The safety runtime targets safety-critical applications with clearer error handling and memory management. The numbered walkthrough below pairs 8.x and 10.x C++ snippets for the same tasks, and the final section lists removed safety APIs and replacements.

14.1. Migrating from TensorRT 8.x Safety Runtime to 10.x

The numbered steps below show TensorRT 8.x first, then TensorRT 10.x, for loading the engine, querying I/O tensors, execution context setup, tensor addresses, and inference.

1. Load the engine from the user's local file system into a memory buffer.
2. Create an `IRuntime` object to deserialize the CUDA engine. Use the `IRuntime` object's `deserializeCudaEngine` method to retrieve the `ICudaEngine` object from the memory buffer.

   Before (TensorRT 8.x)

```
using namespace nvinfer1::safe;
auto infer = createInferRuntime(logger);
infer->setErrorRecorder(recorder.get());
auto engine = infer->deserializeCudaEngine(enginebuffer.data(),
↪enginebufferSize);
```

   The `infer` object (an `IRuntime` instance) provides the primary C++ entry point for TensorRT. It deserializes a serialized engine buffer and produces an `ICudaEngine` object.

   After (TensorRT 10.x)

```
using namespace nvinfer2::safe;
ITRTGraph *graph = nullptr;
ErrorCode code = createTRTGraph(/* ITRTGraph *& */graph,
/*Engine buffer*/ buffer.data(),
```

(continues on next page)

(continued from previous page)

```
/*Engine buffer size*/ buffer.size(),
/*ISafeRecorder& */ recorder,
/*trtManagedScratch*/ true,
/*ISafeMemAllocator* */ allocator);
```

`ITRTGraph` represents a neural network graph. The `createTRTGraph` function takes the engine data buffer and constructs the graph object from the serialized engine file.

- `recorder` – a reference to an `ISafeRecorder` object (a derived class of `IErrorRecorder`). A sample implementation is provided.
- `trtManagedScratch` – when `true` (default), the memory allocator (user-supplied or built-in) allocates scratch memory, equivalent to `createExecutionContext` in TensorRT 8.x. When `false` (equivalent to `createExecutionContextWithoutScratch`), the user must supply and manage scratch memory directly.
- `allocator` – an optional custom memory allocator implementing the `ISafeMemAllocator` interface. If `nullptr` (default), the built-in allocator is used.

3. Get the input/output tensor attributes from the engine.

Before (TensorRT 8.x)

```
int32_t nb = engine->getNbIOTensors();
char const* inputName = engine->getIOTensorName(inputIndex);
char const* outputName = engine->getIOTensorName(outputIndex);
Dims inputDims = engine->getTensorShape(inputName);
Dims outputDims = engine->getTensorShape(outputName);
DataType inputType = engine->getTensorDataType(inputName);
DataType outputType = engine->getTensorDataType(outputName);
```

To calculate the tensor volume, you might also need `getTensorVectorizedDim`, `getTensorBytesPerComponent`, and `getTensorComponentsPerElement`.

After (TensorRT 10.x)

```
int64_t nb;
ErrorCode code = graph->getNbIOTensors(nb);
char const* inputName;
code = graph->getIOTensorName(inputName, inputIndex);
char const* outputName;
code = graph->getIOTensorName(outputName, outputIndex);
TensorDescriptor inputDescriptor;
code = graph->getIOTensorDescriptor(inputDescriptor, inputName);
TensorDescriptor outputDescriptor;
code = graph->getIOTensorDescriptor(outputDescriptor, outputName);
```

In TensorRT 10.x, the `TensorDescriptor` convenience class contains all the information needed to construct an `IOTensor` outside of TensorRT and pass its address back to TensorRT.

```
struct TensorDescriptor
{
    //! Name of the IO Tensor.
    AsciiChar const* tensorName{nullptr};
```

(continues on next page)

(continued from previous page)

```
    //! Static shape of the IO Tensor.
    //! \note The static shape will depend on the IO Profile selected.
    PhysicalDims shape{-1, {}};
    //! Stride vector for each element of the IO Tensor.
    PhysicalDims stride{-1, {}};
    //! DataType enum for the IO Tensor.
    //! \warning The actual type of the tensor data must correspond to this DataType.
    DataType dataType{DataType::kFLOAT};
    //! The size of the tensor data type in bytes (4 for float and int32, 2 for half, 1 for int8)
    uint64_t bytesPerComponent{0U};
    //! The vector length (in scalars) for a vectorized tensor, 1 if the tensor is not vectorized.
    uint64_t componentsPerVector{1U};
    //! The dimension index that the tensor is vectorized along, -1 if the tensor is not vectorized.
    int64_t vectorizedDim{-1};
    //! Total size in bytes for the IO Tensor.
    uint64_t sizeInBytes{0U};
    //! Enum to denote whether the Tensor is for input or output.
    TensorIOMode ioMode{TensorIOMode::kNONE};
    //! Enum to denote whether the Tensor memory is allocated on the GPU, CPU, or CPU_PINNED.
    MemoryPlacement memPlacement{MemoryPlacement::kNONE};
    //! The order that the dimensions are laid out in memory.
    PhysicalDims strideOrder{-1, {}};
    //! The original user-specified shape of the tensor, provided as a reference if the tensor is
    //! vectorized. When a tensor is vectorized by size T along some dimension with size K, the
    //! vectorized dimension is split into two dimensions of sizes ceil(K/T) and T.
    Dims userShape{-1, {}};
};
```

The `sizeInBytes` field directly provides the memory required for the tensor.

The descriptor captures all properties needed to query, allocate, read, and write IO tensors. The physical tensor shape and layout chosen by the compiler can differ from the original user-specified shape due to padding or tiling for vectorized layouts. For example, a user-specified shape `[1,10,3,4]` with a `TensorFormat` of `NHWC8` tiles by 8 elements along the C dimension, producing the following descriptor fields:

User-specified properties (`NHWC8`):

- `userShape`: `[1,10,3,4]` (in NCHW order)
- `componentsPerVector`: 8
- `vectorizedDim`: 1 (C dimension)

Compiler-determined layout:

- `shape`: `[1,2,8,3,4]` (C dimension split into two dimensions)
- `stride`: `[192,8,1,64,16]`

- `strideOrder`: `[4,1,0,3,2]` (tiled C dimensions change fastest)

4. Allocate device/host buffers for the input/output tensors of the engine.
5. Create an `ExecutionContext` from the engine with or without scratch memory. If without scratch memory, the user also needs to allocate and set memory for the context.

Before (TensorRT 8.x)

```
auto context = engine->createExecutionContext();
```

If without `scratchMemory`:

```
auto context = engine->createExecutionContextWithoutDeviceMemory();
size_t size = engine->getDeviceMemorySize();
void* mem;
cudaError_t code = cudaMalloc(&mem, size);
context->setDeviceMemory(mem);
```

After (TensorRT 10.x)

There is no need to call `createExecutionContext` because the `ITRTGraph` object carries its own context.

If `trtManagedScratch` was set to `false` when creating the graph, scratch memory must be supplied manually:

```
size_t size;
code = graph->getScratchMemorySize(size);
void* mem;
cudaError_t code = cudaMalloc(&mem, size);
code = graph->setScratchMemory(mem);
```

6. Preprocess the input data with the host buffer and copy it to the device buffer.
7. Set the input/output tensor addresses of the context with the device buffer addresses accordingly.

Before (TensorRT 8.x)

```
bool result = context->setInputTensorAddress(inputName,inDataPtr);
result = context->setOutputTensorAddress(outputName, outDataPtr);
```

After (TensorRT 10.x)

```
TypedArray inputData(inDataPtr, inSize);
code = graph->setInputTensorAddress(inputName, inputData);
TypedArray outputData(outDataPtr, outSize);
graph->setOutputTensorAddress(outputName, outputData);
```

In TensorRT 10.x, IO tensor memory must be strongly typed and wrapped in a `TypedArray` object.

8. Call enqueue to start inference.

Before (TensorRT 8.x)

```
result = context->enqueueV3(stream);
cudaStreamSynchronize(stream);
```

After (TensorRT 10.x)

```
code = graph->executeAsync(stream);
code = graph->sync();
```

In TensorRT 10.x, CUDA stream synchronization is handled by the `sync()` API, which also performs additional finalization.

9. Postprocess the output device buffer to retrieve the inference output.

14.2. Removed Safety C++ APIs and Replacements

Warning

The APIs listed below have been **removed** in TensorRT 10.x and will cause compilation or linker errors. Review each entry for its replacement before upgrading.

The naming pattern follows the general TensorRT migration from binding-based APIs to name-based (tensor) APIs.

Removed API	Replacement
`safe::ICudaEngine::bindingIsInput()`	`safe::ICudaEngine::tensorIOMode()`
`safe::ICudaEngine::getBindingBytesPerC`	`safe::ICudaEngine::getTensorBytesPerComponent()`
`safe::ICudaEngine::getBindingComponent`	`safe::ICudaEngine::getTensorComponentsPerElement()`
`safe::ICudaEngine::getBindingDataType(`	`safe::ICudaEngine::getTensorDataType()`
`safe::ICudaEngine::getBindingDimension`	`safe::ICudaEngine::getTensorShape()`
`safe::ICudaEngine::getBindingIndex()`	`safe::` name-based methods
`safe::ICudaEngine::getBindingName()`	`safe::` name-based methods
`safe::ICudaEngine::getBindingVectorize`	`safe::ICudaEngine::getTensorVectorizedDim()`
`safe::ICudaEngine::getNbBindings()`	`safe::ICudaEngine::getNbIOTensors()`
`safe::ICudaEngine::getTensorFormat()`	`safe::ICudaEngine::getBindingFormat()`
`safe::IExecutionContext::enqueueV2()`	`safe::IExecutionContext::enqueueV3()`
`safe::IExecutionContext::getStrides()`	`safe::IExecutionContext::getTensorStrides()`

See also

TensorRT C++ API Documentation
Full C++ API documentation.

TensorRT Python API Documentation
Full Python API documentation.

Version Compatibility **(in the *TensorRT Developer Guide*)**
Engine compatibility across TensorRT versions and hardware.

Troubleshooting **(in the *TensorRT Developer Guide*)**
Common errors and debugging guidance.

Upgrading TensorRT
Package-level upgrade instructions (pip, Debian, RPM, tar, zip) to complement API-level migration.

TensorRT GitHub: Compare Releases
View API changes between releases using the GitHub compare tool.

Copyright