

R3AL Quant — Code-Handleiding

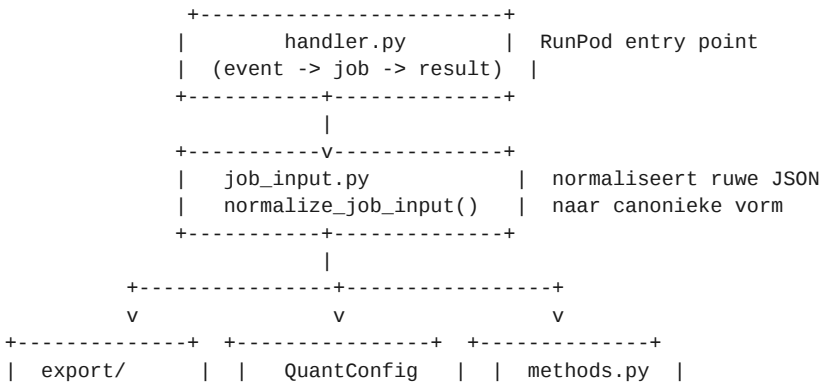
Deze handleiding legt **alle code van de r3al-quant SDK** uit, bestand voor bestand, functie voor functie. Doel: dat je zonder de code zelf te lezen precies begrijpt wat elk onderdeel doet, waarom het zo gebouwd is, en hoe de stukken samenwerken.

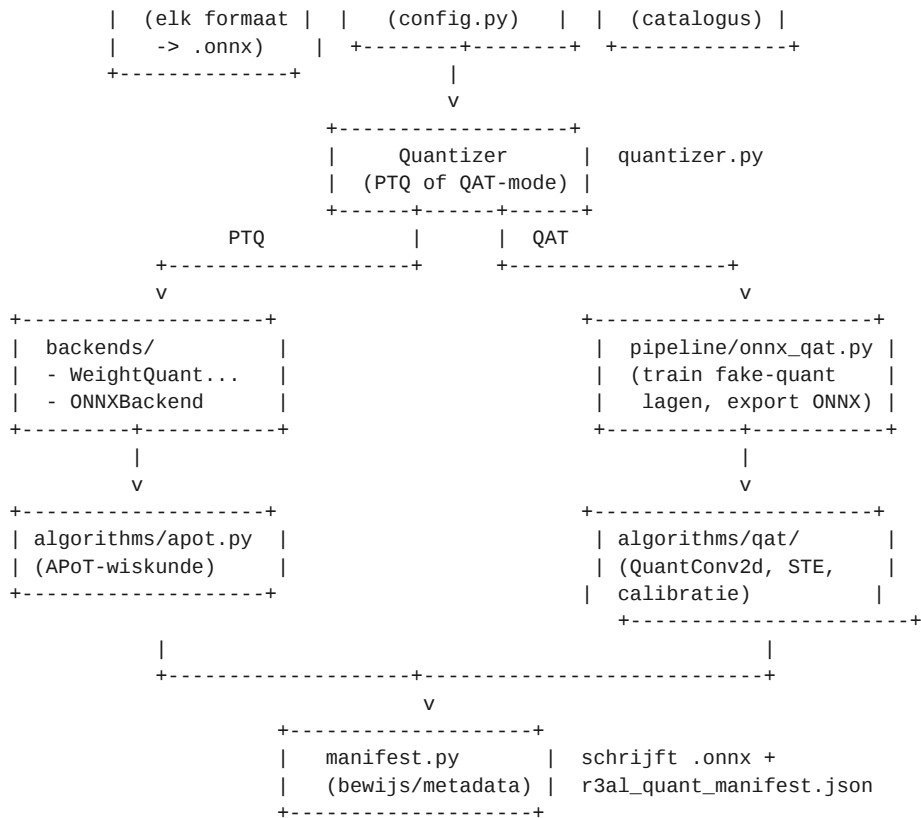
Voor een gebruikersgerichte (API/CLI) uitleg zie [docs/HANDLEIDING.md](#) en de Mintlify-site in [docs-site/](#). Dit document is de **technische/code-laag** eronder.

Inhoudsopgave

- 1. [Architectuur in vogelvlucht](#)
- 2. [Bestandsstructuur](#)
- 3. [Kernlaag: types, config, errors](#)
- 4. [Publieke API: `\_\_init\_\_.py`](#)
- 5. [quantizer.py — Quantizer en QuantizedModel](#)
- 6. [Backends: hoe quantisatie echt gebeurt](#)
- 7. [Algorithms: de wiskunde \(APoT, fake-quant lagen\)](#)
- 8. [Pipeline: QAT end-to-end](#)
- 9. [Export: elk model-formaat -> ONNX](#)
- 10. [Manifest, calibratie en benchmark](#)
- 11. [job_input.py — API-normalisatie](#)
- 12. [methods.py — method-catalogus](#)
- 13. [client.py — HTTP-client + lokale helpers](#)
- 14. [handler.py — RunPod entry point](#)
- 15. [Volledige request-flows \(stap voor stap\)](#)
- 16. [Tests](#)
- 17. [Cheat sheet](#)

1. Architectuur in vogelvlucht





Twee paradigma's, één input-formaat (ONNX):

Paradigma	Actie	Traint?	Backend/pipeline
PTQ (Post-Training Quantization)	quantize	Nee	backends/vision/*.py
QAT (Quantization-Aware Training)	qat_pipeline	Ja	pipeline/onnx_qat.py

Alles draait om **.onnx in**, **.onnx uit** — nooit een ander formaat als eindresultaat. Dat maakt de hele SDK model-agnostisch: ResNet, YOLO, EfficientNet, een custom architectuur — zolang het naar ONNX geëxporteerd is, werkt dezelfde pipeline.

2. Bestandsstructuur

```

D:\R3AL.ai\
+-- handler.py                # RunPod serverless entry point
+-- pyproject.toml            # package metadata + dependency extras
+-- Dockerfile                # RunPod GPU image
+-- runpod.yaml               # RunPod deploy config
+-- src/r3al_quant/
|   +-- __init__.py           # publieke exports (top-level API)
|   +-- types.py              # Enums + BenchmarkResult (Pydantic)
|   +-- config.py             # QuantConfig (Pydantic, validatie)
|   +-- errors.py             # Custom exception-hiërarchie
|   +-- quantizer.py          # Quantizer + QuantizedModel
|   +-- client.py             # R3ALQuantClient (HTTP) + lokale helpers
|   +-- job_input.py          # normalize_job_input()
|   +-- methods.py            # list_quantization_methods()
|   +-- manifest.py           # manifest/metadata bestand-writers

```

```

| +-- calibration.py          # calibratie-validatie/generatoren
| +-- benchmark.py          # Benchmark (latency vergelijking)
| +-- backends/
| | +-- base.py             # BaseBackend (abstracte interface)
| | +-- __init__.py         # get_backend() registry
| | +-- vision/
| | | +-- onnx.py           # ONNXBackend (int8_dynamic/static)
| | | +-- onnx_weight_quant.py # WeightQuantONNXBackend (APoT/uniform)
| | | +-- onnx_utils.py     # primary_input_name()
| | | +-- onnx_loader.py    # ONNX -> PyTorch (voor QAT)
| | | +-- onnx_export.py    # finalize_onnx_deliverable()
| +-- algorithms/
| | +-- apot.py             # APoT-wiskunde (framework-agnostisch)
| | +-- qat/
| | | +-- layers.py         # QuantConv2d, STE fake-quant
| | | +-- calibration.py    # calibrate_activations()
| | | +-- onnx_model.py     # Conv2d ↔ QuantConv2d graph-surgery
| +-- pipeline/
| | +-- onnx_qat.py         # run_onnx_qat_pipeline() – QAT end-to-end
| +-- export/
| | +-- base.py             # ExportOptions/ExportResult/manifest
| | +-- registry.py        # resolve_adapter() – kiest converter
| | +-- adapters/
| | | +-- onnx.py           # passthrough (valideert bestaand ONNX)
| | | +-- ultralytics.py    # YOLO .pt -> ONNX
| | | +-- pytorch.py        # generieke torch.nn.Module -> ONNX
| | | +-- tensorflow.py     # SavedModel/.h5/.pb -> ONNX
| | | +-- tflite.py         # .tflite -> ONNX
| | | +-- paddle.py         # Paddle inference -> ONNX
+-- tests/                  # pytest-suite (1-op-1 met bovenstaande)

```

3. Kernlaag: types, config, errors

3.1 `src/r3a1_quant/types.py`

Definieert alle **Enums** en het `BenchmarkResult` Pydantic-model. Dit zijn de "woordenschat" van de SDK — elk ander bestand importeert hiervandaan.

```

class ModelType(str, Enum):
    VISION = "vision"

class QuantizationMode(str, Enum):
    """Top-level quantization paradigm."""

    PTQ = "ptq" # Post-training: ONNX in -> ONNX out
    QAT = "qat" # Quantization-aware training -> ONNX export

class VisionMethod(str, Enum):
    """PTQ methods – ONNX in, ONNX out (no training)."""

    WEIGHT_QUANT = "weight_quant"
    INT8_DYNAMIC = "int8_dynamic"
    INT8_STATIC = "int8_static"

```

- **ModelType** — momenteel alleen `VISION` (SDK is puur vision-ONNX; geen LLM/tekst-quantisatie).
- **QuantizationMode** — `PTQ` (geen training) of `QAT` (wel training). Dit is de eerste "fork" in alle logica: `Quantizer.__init__`, `get_backend()`, `job_input.py` kijken allemaal eerst naar `mode`.
- **VisionMethod** — de drie PTQ-methodes. Elke waarde komt terug als string in JSON-payloads (`"method": "weight_quant"`).
- **QATScheme** — `uniform` of `apot`, het type quantisatie-grid.
- **QuantBackend** — `cuda` / `cpu` / `auto`, welk device gebruikt wordt.

- **BenchmarkResult** — Pydantic-model met alle velden die `Quantizer.benchmark()` teruggeeft (latency, VRAM, kwaliteit). Elk veld is `optional` behalve de kern-latency's, omdat niet elke backend VRAM of een kwaliteitsscore kan meten.

3.2 `src/r3a1_quant/config.py` — `QuantConfig`

Het **centrale configuratie-object**. Elke actie (`quantize`, `qat_pipeline`, `benchmark`) bouwt hier eerst een `QuantConfig` van, en al het gedrag stroomt daaruit voort via `@property`'s.

```
class QuantConfig(BaseModel):
    """Configuration for vision quantization (PTQ or QAT)."""

    model_type: ModelType = Field(default=ModelType.VISION, ...)
    mode: QuantizationMode = Field(default=QuantizationMode.PTQ, ...)
    bits: int = Field(default=8, ge=3, le=8, ...)
    method: str = Field(default=VisionMethod.WEIGHT_QUANT.value, ...)
    backend: QuantBackend = Field(default=QuantBackend.AUTO, ...)
    target_device: str = Field(default="auto", ...)
    apot_smoothing: float = Field(default=0.6, gt=0.0, le=1.0, ...)
    apot_per_channel: bool = Field(default=True, ...)
    qat_wbit: int = Field(default=8, ge=3, le=8, ...)
    qat_abit: int = Field(default=8, ge=3, le=8, ...)
    qat_scheme: str = Field(default="apot", ...)
    qat_first_layer_bit: int | None = Field(default=8, ...)
    qat_quant_act: bool = Field(default=True, ...)
    qat_calib_batches: int = Field(default=16, ge=1, ...)
    qat_calib_method: str = Field(default="percentile", ...)
    qat_calib_q: float = Field(default=0.9999, gt=0.0, le=1.0, ...)
    qat_epochs: int = Field(default=1, ge=1, ...)
    output_format: Literal["onnx"] = Field(default="onnx", ...)
```

Belangrijke velden en waarom ze bestaan:

Veld	Betekenis	Gebruikt door
<code>mode</code>	ptq of qat	<code>Quantizer.__init__</code> , <code>get_backend()</code>
<code>method</code>	welke PTQ-methode (<code>weight_quant/int8_dynamic/int8_static</code>)	<code>get_backend()</code> selecteert de backend-klasse
<code>bits</code>	bit-breedte voor ONNX Runtime int8-methodes	validatie: moet in {8} zitten voor <code>int8_*</code>
<code>qat_wbit</code> / <code>qat_abit</code>	weight/activation bit-breedte voor <code>weight_quant</code> én QAT	<code>weight_quantize_fn</code> , <code>act_quantize_fn</code>
<code>qat_scheme</code>	apot of uniform — het quant-grid	<code>_quantize_weight_array()</code> , <code>QuantConv2d</code>
<code>apot_smoothing</code>	exponent voor R3AL's <code>apot_r3a1</code> -variant	<code>build_apot_r3a1_levels()</code>
<code>apot_per_channel</code>	per-kanaal of per-tensor schaal	<code>apot_quantize_tensor()</code>

<code>qat_first_layer_bit</code>	eerste conv-laag apart bit-breedte houden (vaak gevoeliger)	<code>quantize_onnx_model()</code>
<code>qat_calib_batches / qat_calib_method / qat_calib_q</code>	hoe activatie-ranges gemeten worden vóór training	<code>calibrate_activations()</code>
<code>qat_epochs</code>	trainings-epochs (default 1 = smoke test)	<code>run_onnx_qat_pipeline()</code>

Validatie (`model_validator`) — dit is waar "foute" configs meteen worden afgekeurd, vóórdat er ooit een model geladen wordt:

```
@model_validator(mode="after")
def validate_config(self) -> QuantConfig:
    if self.model_type != ModelType.VISION:
        raise UnsupportedConfigError("Only vision ONNX quantization is supported.")
    self._validate_vision()
    return self

def _validate_vision(self) -> None:
    if self.qat_scheme not in {"uniform", "apot"}:
        raise UnsupportedConfigError("qat_scheme must be 'uniform' or 'apot'")

    if self.mode == QuantizationMode.QAT:
        if self.qat_calib_method not in {"percentile", "max"}:
            raise UnsupportedConfigError("qat_calib_method must be 'percentile' or 'max'")
        self.bits = self.qat_wbit
        return

    if self.method not in VISION_METHODS:
        raise UnsupportedConfigError(...)

    if self.method == VisionMethod.WEIGHT_QUANT.value:
        self.bits = self.qat_wbit
        return

    if self.bits not in VISION_BITS:
        raise UnsupportedBitsError(self.bits, self.method, sorted(VISION_BITS))
```

Let op de **side-effect**: bij `mode="qat"` of `method="weight_quant"` wordt `self.bits` **overschreven** met `qat_wbit`. Dat is bewust: `bits` is oorspronkelijk bedoeld voor de ONNX Runtime int8-methodes (die alleen 8-bit steunen), terwijl `weight_quant/QAT` via `qat_wbit` 3–8 bit ondersteunen. Door `bits` gelijk te zetten blijft elk stuk code dat naar `config.bits` kijkt (bijv. manifest, logging) correct, ook voor de flexibele methodes.

Berekende properties (geen opgeslagen state, puur afgeleid):

```
@property
def requires_calibration(self) -> bool:
    if self.mode == QuantizationMode.QAT:
        return True
    return self.method == VisionMethod.INT8_STATIC.value

@property
def is_apot(self) -> bool:
    return self.qat_scheme == "apot"

@property
def is_dynamic(self) -> bool:
    return not self.requires_calibration

@property
def requires_onnx_input(self) -> bool:
    return self.method in ONNX_INPUT_METHODS
```

`requires_calibration` is de kern-vraag: *heb ik afbeeldingen nodig?* QAT altijd ja (traint erop), `int8_static` ja (meet activatie-ranges), `weight_quant/int8_dynamic` nee (puur wiskunde op de gewichten).

3.3 `src/r3al_quant/errors.py`

Eén basis-exception (`R3ALQuantError`) met specifieke subklassen. Alles buiten deze hiërarchie (bijv. een onverwachte `KeyError`) wordt in `handler.py` apart afgevangen en als generieke fout teruggegeven — zo blijft de API altijd JSON, nooit een kale Python-traceback naar de klant (behalve in het `traceback`-veld, expliciet toegevoegd voor debugging).

```
class R3ALQuantError(Exception): ...
class UnsupportedConfigError(R3ALQuantError): ...
class UnsupportedBitsError(UnsupportedConfigError):
    def __init__(self, bits, backend, allowed): ...
class BackendNotInstalledError(R3ALQuantError):
    def __init__(self, backend, extra): ... # "pip install r3al-quant[extra]"
class CalibrationRequiredError(R3ALQuantError): ...
class ModelLoadError(R3ALQuantError): ...
class QuantizationError(R3ALQuantError): ...
class BenchmarkError(R3ALQuantError): ...
```

`BackendNotInstalledError` is bijzonder handig: hij bouwt automatisch de juiste `pip install`-hint (`r3al-quant[vision]`, `r3al-quant[export-tf]`, ...) zodat een gebruiker die bijv. `onnx2torch` niet heeft geïnstalleerd meteen weet wat te doen.

4. Publieke API: `__init__.py`

```
"""R3AL Quant – Post-Training Quantization SDK for RunPod."""

from r3al_quant.client import (...)
from r3al_quant.config import QuantConfig
from r3al_quant.job_input import normalize_job_input
from r3al_quant.methods import list_quantization_methods
from r3al_quant.pipeline import OnnxQATPipelineResult, run_onnx_qat_pipeline
from r3al_quant.errors import (...)
from r3al_quant.export import ExportResult, export_to_onnx, list_export_sources, maybe_export_to_onnx
from r3al_quant.quantizer import QuantizedModel, Quantizer
from r3al_quant.types import (...)

__version__ = "0.1.0"
__all__ = [...]
```

Dit bestand is een **her-export laag**: het verzamelt de belangrijkste klassen/functies uit alle submodules zodat gebruikers alleen hoeven te doen:

```
from r3al_quant import Quantizer, QuantConfig
```

in plaats van de interne modulepaden te moeten kennen (`r3al_quant.quantizer.Quantizer`). Alles in `__all__` is bewust **stabiele publieke API** — interne helpers (zoals `_quantize_weight_array`) staan hier niet in.

5. `quantizer.py`

Het startpunt voor bijna elke interactie met de SDK. Bevat twee klassen.

5.1 `QuantizedModel`

Een dun "handle"-object rond een output-directory:

```
class QuantizedModel:
    """Handle to a quantized model on disk."""
```

```

def __init__(self, path: Path, config: QuantConfig) -> None:
    self.path = path
    self.config = config
    self._backend = get_backend(config)

def save(self, output_dir: str | Path | None = None) -> Path:
    """Return the output path (model is already saved during quantize)."""
    if output_dir is not None:
        import shutil
        dest = Path(output_dir)
        dest.mkdir(parents=True, exist_ok=True)
        shutil.copytree(self.path, dest, dirs_exist_ok=True)
        return dest
    return self.path

def load(self, **kwargs: Any) -> Any:
    """Load the quantized model for inference."""
    return self._backend.load(self.path, **kwargs)

```

`save()` is een beetje misleidend genoemd: het model staat er al (de backend schrijft het tijdens `quantize()`). `save()` is puur een **kopieer**-optie als je het resultaat ergens anders ook wil hebben. `load()` delegeert naar dezelfde backend die het gemaakt heeft, zodat laad-logica (manifest lezen, ONNX Runtime sessie opzetten) niet gedupliceerd wordt.

5.2 quantizer

```

def __init__(self, config: QuantConfig) -> None:
    self.config = config
    if config.mode == QuantizationMode.PTQ:
        self._backend = get_backend(config)
        self._benchmark = Benchmark(config)
    else:
        self._backend = None
        self._benchmark = None

def _require_ptq(self) -> None:
    if self.config.mode != QuantizationMode.PTQ:
        raise UnsupportedConfigError(
            "quantize, benchmark, and load require PTQ mode. "
            "Use train_qat() for QAT (mode='qat')."
        )

```

Bij constructie wordt **meteen** de backend opgezocht als `mode=ptq` — dat betekent dat een verkeerde `method`-waarde al bij `Quantizer(config)` faalt, niet pas bij `.quantize()`. Bij `mode=qat` wordt géén backend gebouwd (die route heeft er geen nodig — QAT gebruikt de losse pipeline/`onnx_qat.py`).

quantize() — de PTQ-hoofdmethode:

```

def quantize(
    self,
    model: str | Path,
    calibration_data: Iterable[Any] | None = None,
    output_dir: str | Path = "./quantized_output",
    **kwargs: Any,
) -> QuantizedModel:
    self._require_ptq()
    output_path = self._backend.quantize(
        model=model,
        output_dir=output_dir,
        calibration_data=calibration_data,
        **kwargs,
    )
    return QuantizedModel(output_path, self.config)

```

Puur een dunne wrapper: valideer mode, delegeer naar de backend, wrap het resultaat in `QuantizedModel`. Alle **echte** logica zit in de backend (sectie 6).

benchmark() — vergelijkt origineel met gequantiseerd model:

```

def benchmark(
    self,

```

```

original_model: str | Path,
quantized_model: str | Path | QuantizedModel,
eval_data: Iterable[Any],
*,
quality_fn: Callable[[Any, Iterable[Any]], float] | None = None,
quality_metric: str = "perplexity",
warmup_runs: int = 3,
benchmark_runs: int = 10,
) -> BenchmarkResult:
    self._require_ptq()
    quant_path = quantized_model.path if isinstance(quantized_model, QuantizedModel) else quantized_model
    return self._benchmark.run(...)

```

Accepteert zowel een `QuantizedModel`-object als een kaal pad-string — handig zodat je direct na `quantize()` kunt benchmarken zonder het pad zelf te onthouden.

`train_qat()` — de QAT-ingang:

```

def train_qat(
    self,
    model: str | Path,
    *,
    calibration_data: Iterable[Any] | None = None,
    output_dir: str | Path = "./qat_output",
    **kwargs: Any,
) -> Any:
    if self.config.mode != QuantizationMode.QAT:
        raise UnsupportedConfigError(...)
    if self.config.model_type != ModelType.VISION:
        raise UnsupportedConfigError(...)

    model_path = Path(model)
    if model_path.suffix.lower() != ".onnx":
        raise UnsupportedConfigError(
            "QAT requires an ONNX model (.onnx). Export your checkpoint to ONNX first."
        )
    if calibration_data is None:
        raise UnsupportedConfigError("QAT requires calibration_data (training images).")

    from r3a1_quant.pipeline.onnx_qat import run_onnx_qat_pipeline
    return run_onnx_qat_pipeline(
        config=self.config, model=model, output_dir=output_dir,
        calibration_data=calibration_data, **kwargs,
    )

```

Drie guard-clauses vóórdat er iets zwaars gebeurt: verkeerde mode, niet-ONNX bestand, geen calibratiedata -> allemaal direct een duidelijke foutmelding in plaats van pas diep in de trainingslus te crashen. De import van `run_onnx_qat_pipeline` staat bewust **lokaal in de functie** (lazy import) zodat `torch/onnx2torch` niet al bij `import r3a1_quant` geladen hoeven worden als je alleen PTQ gebruikt.

6. Backends

Backends zijn de klassen die **daadwerkelijk** een ONNX-bestand lezen, transformeren en terugschrijven voor PTQ. Elke backend implementeert dezelfde abstracte interface.

6.1 `backends/base.py` — `BaseBackend`

```

class BaseBackend(ABC):
    """Interface all quantization backends must implement."""

    name: str

    def __init__(self, config: QuantConfig) -> None:
        self.config = config

```

```

@abstractmethod
def quantize(self, model, output_dir, calibration_data=None, **kwargs) -> Path: ...

@abstractmethod
def load(self, model_path, **kwargs) -> Any: ...

def _resolve_device(self) -> str:
    import torch
    if self.config.target_device != "auto":
        return self.config.target_device
    if self.config.backend == QuantBackend.CPU:
        return "cpu"
    if self.config.backend == QuantBackend.CUDA:
        return "cuda" if torch.cuda.is_available() else "cpu"
    return "cuda" if torch.cuda.is_available() else "cpu"

```

Elke backend moet dus minstens `quantize()` en `load()` implementeren. `_resolve_device()` is gedeelde logica: expliciete device-string wilt, anders val terug op backend-instelling, anders auto-detecteer CUDA.

6.2 `backends/_init_.py` — `get_backend(): de registry`

```

_VISION_BACKENDS: dict[str, type[BaseBackend]] = {
    VisionMethod.INT8_DYNAMIC.value: ONNXBackend,
    VisionMethod.INT8_STATIC.value: ONNXBackend,
    VisionMethod.WEIGHT_QUANT.value: WeightQuantONNXBackend,
}

def get_backend(config: QuantConfig) -> BaseBackend:
    """Resolve and instantiate the appropriate backend for the config."""
    if config.model_type != ModelType.VISION:
        raise UnsupportedConfigError("Only vision ONNX quantization is supported.")
    if config.mode != QuantizationMode.PTQ:
        raise UnsupportedConfigError(
            "Backend selection is PTQ-only. For QAT use action='qat_pipeline' (mode='qat')."
        )
    cls = _VISION_BACKENDS.get(config.method)
    if cls is None:
        raise UnsupportedConfigError(f"Unknown vision method: {config.method}")
    return cls(config)

```

Simpele dictionary-lookup: `method` -> backend-klasse. Merk op dat `int8_dynamic` én `int8_static` **dezelfde klasse** (`ONNXBackend`) gebruiken — het onderscheid tussen dynamic/static zit *binnen* die klasse (zie 6.3), gestuurd door `config.method`.

6.3 `backends/vision/onnx.py` — `ONNXBackend`

Verantwoordelijk voor **echte INT8** via ONNX Runtime's quantization-tools (`int8_dynamic` en `int8_static`).

`quantize()` — het hoofdpad:

```

def quantize(self, model, output_dir, calibration_data=None, **kwargs) -> Path:
    is_static = self.config.method == VisionMethod.INT8_STATIC.value
    if is_static:
        validate_calibration(self.config.method, self.config.requires_calibration, calibration_data)

    from onnxruntime.quantization import QuantFormat, QuantType, quantize_dynamic, quantize_static

    output_path = Path(output_dir)
    output_path.mkdir(parents=True, exist_ok=True)
    model_path = Path(model)
    if model_path.suffix.lower() != ".onnx":
        raise ModelLoadError(...)

    domain = kwargs.get("domain")
    deliverable = output_path / f"{model_path.stem}.quantized.onnx"
    temp_out = output_path / f"{model_path.stem}_ort_int8.onnx"

    if is_static:
        input_name = primary_input_name(model_path, kwargs.get("input_name"))

```

```

        calib_data_reader = self._build_calib_reader(calibration_data, input_name,
kwargs.get("max_calib_samples", 100))
        quantize_static(
            model_input=str(model_path), model_output=str(temp_out),
            calibration_data_reader=calib_data_reader,
            quant_format=QuantFormat.QDQ,
            activation_type=QuantType.QUInt8, weight_type=QuantType.QInt8,
        )
        quant_type = "onnxruntime_static_int8_qdq"
    else:
        quantize_dynamic(model_input=str(model_path), model_output=str(temp_out), weight_type=QuantType.QInt8)
        quant_type = "onnxruntime_dynamic_int8"

deliverable.write_bytes(temp_out.read_bytes())
temp_out.unlink(missing_ok=True)

stamp_onnx_metadata(deliverable, {...})
manifest = build_quant_manifest(...)
write_quant_manifest(output_path, manifest)
write_legacy_meta(output_path, {...})

```

Stap voor stap:

1. **Calibratie-check** — alleen `int8_static` heeft afbeeldingen nodig (dat is `requires_calibration` uit `QuantConfig`).
2. `quantize_static` gebruikt het **QDQ-formaat** (`QuantizeLinear/DequantizeLinear`-nodes ingevoegd in de graph) — `activation_type=QUInt8` (unsigned, want activaties na ReLU zijn ≥ 0 in veel netwerken) en `weight_type=QInt8` (signed, want gewichten kunnen negatief zijn). Dit is de meest gangbare/portable ORT-INT8-vorm.
3. `quantize_dynamic` heeft geen calibratiedata nodig: het kwantiseert alleen de **gewichten**; activaties worden **tijdens inference** on-the-fly berekend (vandaar "dynamic"). Sneller op te zetten, maar meestal iets minder snel op CPU dan static QDQ omdat er per-inference nog quant/dequant-berekeningen gebeuren.
4. Resultaat wordt **naar een tijdelijk bestand** geschreven en dan pas hernoemd naar de definitieve deliverable-naam — dit voorkomt dat een gedeeltelijk geschreven bestand als "klaar" gezien wordt bij een crash halverwege.
5. `stamp_onnx_metadata` zet R3AL-eigen sleutel/waarde-paren in het ONNX `metadata_props`-veld (zie sectie 10.1) — dit is hoe je *in het ONNX- bestand zelf* kunt zien dat het door R3AL is verwerkt, los van het externe manifest-JSON-bestand.

`_build_calib_reader()` — een adapter rond ORT's calibratie-interface:

```

def _build_calib_reader(self, calibration_data, input_name, max_samples) -> Any:
    samples = list(vision_calibration_generator(calibration_data, max_samples=max_samples))

    class _CalibDataReader:
        def __init__(self) -> None:
            self._index = 0

        def get_next(self) -> dict[str, np.ndarray] | None:
            if self._index >= len(samples):
                return None
            sample = samples[self._index]
            self._index += 1
            if hasattr(sample, "numpy"):
                arr = sample.numpy()
            elif isinstance(sample, np.ndarray):
                arr = sample
            else:
                arr = np.asarray(sample, dtype=np.float32)
            if arr.ndim == 3:
                arr = arr[np.newaxis, ...]
            return {input_name: arr.astype(np.float32)}

    return _CalibDataReader()

```

ONNX Runtime verwacht een object met een `get_next()`-methode die telkens een `dict[input_name, array]` teruggeeft, of `None` als de data op is (vergelijkbaar met een iterator, maar als expliciete klasse omdat dat de ORT C++ API-conventie is). De innerlijke klasse: - Accepteert torch tensors (`.numpy()`), numpy arrays, of ruwe lijsten. - Voegt automatisch een batch-dimensie toe als een los `(C,H,W)`-beeld binnenkomt (`ndim == 3 -> [np.newaxis, ...]` maakt er `(1,C,H,W)` van).

load() — laad een eerder gequantiseerd model terug voor inference:

```
def load(self, model_path, **kwargs) -> Any:
    import onnxruntime as ort
    onnx_file, meta = resolve_quantized_onnx_bundle(model_path)
    if not meta.get("quantized", True):
        raise ModelLoadError("Bundle manifest indicates model is not quantized.")
    providers = ["CUDAExecutionProvider", "CPUExecutionProvider"]
    session = ort.InferenceSession(str(onnx_file), providers=providers)
    return {"session": session, "path": str(onnx_file), "manifest": meta}
```

`resolve_quantized_onnx_bundle` (uit `manifest.py`) zoekt het manifest naast het opgegeven pad en haalt daaruit het echte ONNX-bestandspad. De `InferenceSession` probeert eerst CUDA, valt terug op CPU als er geen GPU is — dit is de **enige plek** waar ORT daadwerkelijk een sessie opent voor "laad dit gequantiseerde model".

6.4 backends/vision/onnx_weight_quant.py — WeightQuantONNXBackend

De **default** backend — geen ONNX Runtime int8-transformatie, maar directe manipulatie van de **initializer-tensors** in de ONNX-graph. Dit is *fake quantization*: gewichten worden afgerond naar een grid, maar blijven opgeslagen als `float32`.

Module-level helper — welke tensor kwantiseren we?

```
def _should_quantize_initializer(name: str, array: np.ndarray) -> bool:
    if array.size == 0 or array.dtype not in (np.float32, np.float64):
        return False
    if array.ndim < 2:
        return False
    lowered = name.lower()
    if any(token in lowered for token in ("running", "num_batches", "tracked")):
        return False
    return True
```

Filt: - Lege tensors of niet-float (bijv. int64 shape-tensors) -> skip. - **1-D tensors** (bias-vectoren, BatchNorm-gamma/beta) -> skip, alleen ≥ 2 -D (Conv-filters, Linear-gewichten) worden gequantiseerd. Bias-termen hebben relatief weinig invloed en profiteren nauwelijks van quantisatie, terwijl fouten daar makkelijker doorwerken. - Namen met `running/num_batches/tracked` -> dit zijn BatchNorm's **running statistics** (`running_mean`, `running_var`, `num_batches_tracked`), geen "echte" geleerde gewichten — die moet je nooit kwantiseren, want ze zijn essentieel voor de exacte normalisatie-wiskunde.

quantize_weight_array() — dispatcht naar het juiste schema:

```
def _quantize_weight_array(array, *, scheme, bits, apot_smoothing, per_channel) -> np.ndarray:
    tensor = torch.from_numpy(array.astype(np.float32))
    if scheme == "apot":
        wq = weight_quantize_fn(w_bit=bits, power=True)
        wq.init_alpha(tensor)
        with torch.no_grad():
            return wq(tensor).numpy()

    if scheme == "uniform":
        wq = weight_quantize_fn(w_bit=bits, power=False)
        wq.init_alpha(tensor)
        with torch.no_grad():
            return wq(tensor).numpy()

    if scheme == "apot_ptq":
        levels = build_apot_levels(bits)
        dequant, _ = apot_quantize_tensor(tensor, levels, per_channel=per_channel, channel_dim=0)
        return dequant.numpy()
```

```

if scheme == "apot_r3a1":
    levels = build_apot_r3a1_levels(bits, apot_smoothing)
    dequant, _ = apot_quantize_tensor(tensor, levels, per_channel=per_channel, channel_dim=0)
    return dequant.numpy()

raise QuantizationError(f"Unknown weight quant scheme '{scheme}'")

```

Vier schema's, twee families: - **apot/uniform** — hergebruiken de QAT-laag `weight_quantize_fn` (sectie 7.2), die zelf `init_alpha()` (schaal = max-abs van de tensor) aanroept en dan een forward-pass doet zonder gradient. Dit is dus *dezelfde* quantisatie-logica als tijdens QAT-training, maar hier eenmalig toegepast zonder training. - **apot_ptq/apot_r3a1** — gebruiken de losse, framework-onafhankelijke functies uit `algorithms/apot.py` direct (sectie 7.1). Dit zijn de "pure PTQ"-varianten zonder de STE/`nn.Module`-machinerie.

Hoofd-quantize()-methode:

```

def quantize(self, model, output_dir, calibration_data=None, **kwargs) -> Path:
    import onnx
    model_path = Path(model)
    if model_path.suffix.lower() != ".onnx":
        raise ModelLoadError(...)

    output_path = Path(output_dir)
    output_path.mkdir(parents=True, exist_ok=True)
    scheme = kwargs.get("scheme") or self.config.qat_scheme
    bits = int(kwargs.get("bits") or self.config.qat_wbit)
    domain = kwargs.get("domain")
    int8_runtime = kwargs.get("int8_runtime", False)

    onnx_model = onnx.load(str(model_path))
    quantized_layers = 0

    for initializer in onnx_model.graph.initializer:
        array = onnx.numpy_helper.to_array(initializer)
        if not _should_quantize_initializer(initializer.name, array):
            continue
        q_array = _quantize_weight_array(array, scheme=scheme, bits=bits,
                                         apot_smoothing=self.config.apot_smoothing,
                                         per_channel=self.config.apot_per_channel)
        initializer.CopyFrom(onnx.numpy_helper.from_array(q_array, initializer.name))
        quantized_layers += 1

    if quantized_layers == 0:
        raise QuantizationError("No quantizable weight tensors found in ONNX graph.")

    deliverable = output_path / f"{model_path.stem}.quantized.onnx"
    onnx.save(onnx_model, str(deliverable))
    stamp_onnx_metadata(deliverable, {...})

    manifest = build_quant_manifest(..., extra={"scheme": scheme, "bits": bits,
                                                "quantized_layers": quantized_layers,
                                                "quantization_type": "onnx_weight_quant"})
    write_quant_manifest(output_path, manifest)
    write_legacy_meta(output_path, {...})

    if int8_runtime:
        self._also_export_ort_int8(model_path, output_path, calibration_data, kwargs, deliverable, domain)

    return output_path

```

Kernidee: **loop over elke initializer** in de ONNX-graph (dat zijn alle "constante" tensors — gewichten, bias, BatchNorm-parameters), filter met `_should_quantize_initializer`, kwantiseer met `_quantize_weight_array`, en **schrijf het resultaat terug in-place** via `initializer.CopyFrom(...)`. De graph-structuur (welke node met welke node verbonden is) blijft **volledig ongewijzigd** — alleen de getallen in de tensors veranderen. Dat is waarom het bestand ongeveer even groot blijft (nog steeds float32) en overal draait waar het originele ONNX ook draaide.

`int8_runtime=True` is een **bonus-optie**: naast de fake-quant deliverable ook nog een *echte* INT8-versie maken via `ONNXBackend`, in een submap `onnxruntime_int8/`. Zo kun je in één job zowel de APoT-geanalyseerde als de

productie-INT8-versie krijgen.

`_also_export_ort_int8()`:

```
def _also_export_ort_int8(self, source, output_path, calibration_data, kwargs, fake_quant_onnx, domain) ->
None:
    ort_config = QuantConfig(model_type="vision", method=VisionMethod.INT8_STATIC.value, bits=8)
    runtime_dir = output_path / "onnxruntime_int8"
    ONNXBackend(ort_config).quantize(
        fake_quant_onnx, runtime_dir, calibration_data=calibration_data,
        input_name=primary_input_name(fake_quant_onnx, kwargs.get("input_name")),
        max_calib_samples=kwargs.get("max_calib_samples", 100), domain=domain,
    )
```

Interessant detail: dit kwantiseert de **al-fake-gequantiseerde** ONNX (`fake_quant_onnx`) verder met `ONNXBackend/int8_static` — niet het originele bestand. Dat is bewust: zo zie je het gecombineerde effect van eerst APoT-afroonden en dan pas echte INT8-opslag.

`load()` — spiegelbeeld van `ONNXBackend.load()`, met dezelfde manifest-resolutie maar via `r3al_quant.manifest.resolve_quantized_onnx_bundle`.

6.5 `backends/vision/onnx_utils.py`

```
def primary_input_name(model_path: str | Path, override: str | None = None) -> str:
    """Return the primary graph input name for calibration / inference."""
    if override:
        return override
    path = Path(model_path)
    if path.suffix.lower() != ".onnx" or not path.is_file():
        return "input"
    import onnx
    model = onnx.load(str(path))
    if model.graph.input:
        return model.graph.input[0].name
    return "input"
```

Kleine maar belangrijke helper: elk ONNX-model heeft een eigen naam voor zijn input-tensor ("`images`" bij YOLO, "`input`" bij veel andere exports, etc.). In plaats van de gebruiker te vragen dit steeds handmatig op te geven, wordt het **automatisch uit de graph gelezen** — tenzij expliciet overschreven via `input_name` in de job-payload.

6.6 `backends/vision/onnx_loader.py` — ONNX -> PyTorch (voor QAT)

QAT kan niet direct op een ONNX-graph trainen (geen autograd-support in het ONNX-formaat zelf), dus wordt het model eerst terug omgezet naar een PyTorch `nn.Module`.

```
def onnx_input_spec(model_path, *, batch_size=1, imgsz=None) -> tuple[str, list[int]]:
    """Return primary ONNX input name and NCHW shape (defaults for dynamic dims)."""
    graph_in = onnx.load(str(path)).graph.input[0]
    name = graph_in.name
    dims = graph_in.type.tensor_type.shape.dim
    shape: list[int] = []
    for index, dim in enumerate(dims):
        if dim.dim_value > 0:
            shape.append(int(dim.dim_value))
        elif index == 0:
            shape.append(batch_size)
        elif index in (2, 3):
            shape.append(imgsz or 224)
        else:
            shape.append(3 if index == 1 else 1)
    return name, shape
```

Leest de **input-shape** uit de graph. Lastigheid: veel ONNX-exports hebben **dynamische dimensies** (bijv. batch-grootte `-1` zodat het model met elke batch-grootte werkt). Voor elke dimensie die niet vast is (`dim_value <= 0`) wordt een verstandige default ingevuld: dim 0 (batch) -> `batch_size` parameter, dim 2/3 (hoogte/breedte bij NCHW) -> `imgsz` of 224, dim 1 (kanalen) -> 3 (RGB) tenzij het de batch-dim zelf is.

```
def load_onnx_as_torch(model_path: str | Path) -> nn.Module:
    """Convert an ONNX file to a trainable PyTorch module."""
    path = Path(model_path)
    if path.suffix.lower() != ".onnx":
        raise ModelLoadError(...)
    from onnx2torch import convert
    model = convert(str(path))
    if not isinstance(model, nn.Module):
        raise ModelLoadError("onnx2torch did not return an nn.Module")
    return model
```

Gebruikt de externe library `onnx2torch` (optionele dependency, extra `[vision]`) om elke ONNX-operator (Conv, BatchNorm, Relu, ...) om te zetten naar het equivalente `torch.nn`-component. Het resultaat is een volledig trainbaar PyTorch-model met **exact dezelfde gewichten** als het originele ONNX-bestand.

```
def resolve_torch_device(device: str | None) -> torch.device:
    if not device or device == "auto":
        return torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    if device == "cpu":
        return torch.device("cpu")
    if device.isdigit():
        return torch.device(f"cuda:{device}")
    return torch.device(device)
```

Accepteert "auto", "cpu", een los cijfer ("0" -> cuda:0), of een volle device-string ("cuda:1").

6.7 backends/vision/onnx_export.py

```
def finalize_onnx_deliverable(onnx_path, output_dir, *, config_dict, source_model, domain=None, extra=None) -> Path:
    """Copy ONNX to output_dir and write the customer manifest."""
    output_dir.mkdir(parents=True, exist_ok=True)
    deliverable = output_dir / f"{onnx_path.stem}.quantized.onnx"
    deliverable.write_bytes(onnx_path.read_bytes())
    manifest = build_quant_manifest(config=config_dict, source_model=source_model,
                                    output_model=deliverable, output_format="onnx",
                                    domain=domain, extra=extra)
    write_quant_manifest(output_dir, manifest)
    return deliverable
```

Gedeelde "laatste stap" — kopieer het uiteindelijke ONNX-bestand naar de klant-facing naam (*.quantized.onnx) en schrijf het manifest ernaast. Wordt **alleen** gebruikt door de QAT-pipeline (`pipeline/onnx_qat.py`); de PTQ- backends doen dit inline in hun eigen `quantize()`-methode omdat ze net iets andere extra-velden in het manifest zetten.

7. Algorithms

Deze map bevat de **pure wiskunde** — geen I/O, geen bestanden, geen ONNX- specifieke code. Herbruikbaar door zowel de PTQ-backend als de QAT-pipeline.

7.1 algorithms/apot.py — APoT-quantisatie

Gebaseerd op het paper *Li, Dong & Wang (2020), "Additive Powers-of-Two Quantization"*, plus een R3AL-variant.

Waarom niet gewoon uniform quantiseren? Neurale netwerk-gewichten zijn doorgaans **klokvormig verdeeld** rond nul (veel kleine waarden, een paar grote uitschieters). Een uniform grid (evenredig verdeelde niveaus tussen -max en +max) verspilt resolutie: te weinig niveaus vlak bij nul (waar de meeste gewichten zitten) en een grof net in de staarten. APoT bouwt elk niveau als **som van een paar machten van twee**, wat van nature meer niveaus concentreert rond nul.

build_apot_levels() — bouwt de positieve helft van het grid:

```
def build_apot_levels(bits: int) -> torch.Tensor:
    if bits < 2 or bits % 2 != 0:
        raise UnsupportedConfigError(
```

```

        f"APoT quantization currently supports even bit-widths >= 2 (got {bits}). Use 4 or 8 bits."
    )
    n_terms = bits // 2
    bases: list[list[float]] = []
    for term in range(n_terms):
        base = [0.0]
        for i in range(_NONZERO_TERMS_PER_BASE): # = 3
            exponent = -(n_terms * i + term + 1)
            base.append(2.0**exponent)
        bases.append(base)

    magnitudes = {sum(combo) for combo in itertools.product(*bases)}
    levels = torch.tensor(sorted(magnitudes), dtype=torch.float64)
    levels = levels / levels.max()
    return levels.to(torch.float32)

```

Uitgelegd voor `bits=8`: `n_terms = 4`. Voor elke van de 4 "bases" wordt een lijstje van 4 mogelijke waarden gebouwd (0 plus 3 machten van twee met specifieke exponenten). Dan wordt het **cartesisch product** van die 4 lijstjes genomen (`itertools.product`) en van elke combinatie de **som** berekend. Met 4 bases $\times$ 4 opties = $4 \times 4 = 256 = 2^8$ unieke sommen — precies het aantal niveaus dat een 8-bit quantizer nodig heeft. Elke som is per constructie een exacte optelling van machten van twee, wat betekent dat vermenigvuldigen-met-een-niveau in hardware kan met **bit-shifts** in plaats van volledige vermenigvuldigers (de kern-belofte van het APoT-paper).

Resultaat wordt genormaliseerd zodat het grootste niveau exact 1.0 is — schaling naar de werkelijke gewicht-range gebeurt later apart (`apot_quantize_tensor`).

build_apot_r3a1_levels() — R3A1's aanpassing:

```

def build_apot_r3a1_levels(bits: int, smoothing: float = 0.6) -> torch.Tensor:
    if not (0.0 < smoothing <= 1.0):
        raise ValueError("apot_smoothing must be in the range (0, 1].")
    levels = build_apot_levels(bits)
    reshaped = levels.pow(smoothing)
    reshaped = reshaped / reshaped.max()
    return reshaped

```

Past een **concave machtstransformatie** $x \rightarrow x^{\text{smoothing}}$ toe op de genormaliseerde niveaus. Omdat dit een monotone (orde-behoudende) transformatie is, blijft het aantal niveaus, hun volgorde en symmetrie hetzelfde — maar de **spreiding** verandert: kleine waarden (die bij standaard-APoT al dicht bij elkaar liggen) worden verder uit elkaar getrokken, terwijl grote waarden (schaars in de staart bij standaard-APoT) dichter bij elkaar komen — dus de staart wordt beter opgevuld. Voor `smoothing < 1.0` schuiven er dus relatief meer niveaus richting de staart. Bij `smoothing = 1.0` is dit exact gelijk aan standaard-APoT. Het compensatie: dit is **geen** exacte som van machten van twee meer, dus de bit-shift-hardware-eigenschap gaat verloren — een bewuste trade-off tussen hardware-snelheid en beter passen bij de werkelijke gewichtsverdeling.

symmetric_levels():

```

def symmetric_levels(positive_levels: torch.Tensor) -> torch.Tensor:
    """Mirror the positive half (including zero) into a full symmetric, sorted level set."""
    negative = -positive_levels.flip(0)[:1] # drop the mirrored zero to avoid a duplicate
    return torch.cat([negative, positive_levels])

```

`build_apot_levels` geeft alleen de **positieve** helft (inclusief 0) terug. Voor echte symmetrische quantisatie (gewichten kunnen negatief zijn) wordt hier de negatieve spiegeling toegevoegd. `.flip(0)[:1]` keert de volgorde om en laat de laatste waarde (die 0.0 zou zijn na omkering van de eerste) weg, zodat 0 niet dubbel voorkomt na de `torch.cat`.

_nearest_level_indices() — efficiënt dichtsbijzijnde-niveau zoeken:

```

def _nearest_level_indices(normalized: torch.Tensor, sorted_levels: torch.Tensor) -> torch.Tensor:
    idx = torch.searchsorted(sorted_levels, normalized.contiguous())
    idx = idx.clamp(max=len(sorted_levels) - 1)
    idx_left = (idx - 1).clamp(min=0)
    left_vals = sorted_levels[idx_left]
    right_vals = sorted_levels[idx]
    pick_left = (normalized - left_vals).abs() <= (right_vals - normalized).abs()
    return torch.where(pick_left, idx_left, idx)

```

Gebruikt `torch.searchsorted` (binaire zoek, $O(N \log L)$) in plaats van een volledige afstandsmatrix ($O(N * L)$) — belangrijk omdat gewicht-tensors miljoenen elementen kunnen hebben terwijl het niveau-grid maar een paar honderd waarden telt. Voor elk element wordt de **invoegepositie** in het gesorteerde grid gevonden, en dan vergeleken of de linker- of rechterbuur dichterbij ligt.

`apot_quantize_tensor()` — de eigenlijke fake-quantize/dequantize:

```
def apot_quantize_tensor(tensor, levels, per_channel=True, channel_dim=0) -> tuple[torch.Tensor,
torch.Tensor]:
    full_levels = symmetric_levels(levels).to(device=tensor.device, dtype=tensor.dtype)

    if per_channel and tensor.dim() > 1:
        reduce_dims = tuple(d for d in range(tensor.dim()) if d != channel_dim)
        scale = tensor.abs().amax(dim=reduce_dims, keepdim=True)
    else:
        scale = tensor.abs().max()
    scale = scale.clamp(min=1e-8)

    normalized = (tensor / scale).clamp(-1.0, 1.0)
    idx = _nearest_level_indices(normalized, full_levels)
    quantized = full_levels[idx]

    dequantized = quantized * scale
    return dequantized, scale
```

Dit implementeert **symmetrische max-abs quantisatie** (dezelfde conventie als TensorRT gebruikt, expliciet genoemd in de whitepaper):

1. **Schaal bepalen** — `per_channel=True` geeft een aparte schaal per output-kanaal (meestal beter voor Conv/Linear-gewichten, omdat verschillende filters heel verschillende amplitudes kunnen hebben); anders één schaal voor de hele tensor.
2. **Normaliseren** naar `[-1, 1]` door te delen door de schaal.
3. **Snap naar dichtstbijzijnde niveau** in het APoT-grid.
4. **De-normaliseren** ($\times$ schaal terug) zodat het resultaat weer in de originele waarde-range zit.

Het resultaat is *fake* quantization: `dequantized` heeft dezelfde `dtype/shape/device` als de input — het is nog steeds float, alleen met afgeronde waarden. Dit is precies wat nodig is om accuracy-impact te meten zonder een aparte INT8-runtime-kernel te hoeven schrijven.

7.2 `algorithms/qat/layers.py` — Fake-quant lagen met STE

Dit bestand bevat de **trainbare** quantisatie-bouwstenen, gebruikt door zowel `weight_quant` (PTQ, eenmalig toegepast) als door de volledige QAT- trainingslus.

`build_power_value()` — een **andere**, met-de-hand-uitgeschreven variant van het APoT-grid (specifiek afgestemd op de YOLOv5-QAT-referentie-code waar deze SDK op voortbouwt):

```
def build_power_value(B: int = 2, additive: bool = True) -> torch.Tensor:
    """Build the APoT magnitude grid used by QAT (Versimpel / YOLOv5 scheme)."""
    base_a = [0.0]; base_b = [0.0]; base_c = [0.0]; base_d = [0.0]
    if additive:
        if B == 2: ...
        elif B == 4: ...
        elif B == 6: ...
        elif B == 3: ...
        elif B == 5: ...
        elif B == 7: ...
        elif B == 8: ...
        else:
            raise NotImplementedError("APoT QAT grid only implemented for bitwidth B in {2,3,4,5,6,7,8}.")
    else:
        for i in range(2**B - 1):
            base_a.append(2 ** (-i - 1))
    values = [a + b + c + d for a in base_a for b in base_b for c in base_c for d in base_d]
```

```
levels = torch.tensor(list(set(values)))
return levels.mul(1.0 / levels.max())
```

Merk op dat `B` hier de bit-breedte **min 1** is (zie `weight_quantize_fn` hieronder: `self.b = w_bit - 1`), omdat één bit "gereserveerd" is voor het teken (`sign`). Voor elke bit-breedte 2 t/m 8 is er een handmatig uitgewerkte combinatie van tot 4 "bases" (`base_a..d`), elk met machten van twee op specifieke exponenten — dit is functioneel vergelijkbaar met `algorithms/apot.py::build_apot_levels`, maar met een net andere constructie die exact matcht met de originele referentie-implementatie waar dit gedeelte van gebaseerd is. `additive=False` geeft de **uniforme** variant: gewoon alle machten van twee tot `2**B - 1`, zonder de additieve combinatie-structuur.

`_uniform_quant()` / `_power_quant()` — de daadwerkelijke rond-operatie:

```
def _uniform_quant(x: torch.Tensor, bits: int) -> torch.Tensor:
    n = 2**bits - 1
    return x.mul(n).round().div(n)

def _power_quant(x: torch.Tensor, value_s: torch.Tensor) -> torch.Tensor:
    shape = x.shape
    x = x.view(-1)
    v = torch.sort(value_s.type_as(x))[0]
    idx = torch.searchsorted(v, x).clamp(1, v.numel() - 1)
    left, right = v[idx - 1], v[idx]
    xhard = torch.where((x - left).abs() <= (right - x).abs(), left, right)
    return xhard.view(shape)
```

`_uniform_quant` is klassieke uniforme afronding: schaal naar `[0, n]`, rond naar geheel getal, schaal terug. `_power_quant` doet hetzelfde soort "dichtstbijzijnde-niveau"-zoektocht als `apot_quantize_tensor` in sectie 7.1, maar met een net iets andere implementatie (werkt op platte 1-D tensor + `view` terug naar originele vorm).

`WeightQuantSTE` — de **Straight-Through Estimator**, de kern van QAT:

```
class _WeightQuantSTE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, alpha, b, grids, power):
        input = input.div(alpha)
        input_c = input.clamp(min=-1, max=1)
        sign = input_c.sign()
        input_abs = input_c.abs()
        if power:
            input_q = _power_quant(input_abs, grids).mul(sign)
        else:
            input_q = _uniform_quant(input_abs, b).mul(sign)
        ctx.save_for_backward(input, input_q)
        return input_q.mul(alpha)

    @staticmethod
    def backward(ctx, grad_output):
        grad_input = grad_output.clone()
        input, input_q = ctx.saved_tensors
        i = (input.abs() > 1.0).float()
        sign = input.sign()
        grad_alpha = (grad_output * (sign * i + (input_q - input) * (1 - i))).sum()
        return grad_input, grad_alpha, None, None, None
```

Waarom is dit nodig? Het "rond naar dichtstbijzijnde niveau"-operatie (`round()`, `searchsorted`) heeft overal **gradiënt nul** (het is een trapfunctie) — als je die direct in een normaal PyTorch-netwerk zou zetten, zou backpropagation nooit een signaal doorgeven om de gewichten te verbeteren. De **Straight-Through Estimator**-truc lost dit op door een *custom* backward-functie te definiëren die **doet alsof** de forward-pass de identiteitsfunctie was:

- **Forward:** normaliseer met `alpha` (leerbare clip-drempel), clip naar `[-1, 1]`, kwantiseer het teken-loze deel, herstel het teken, schaal terug met `alpha`.
- **Backward:** `grad_input = grad_output.clone()` — de gradiënt wordt **ongewijzigd doorgegeven** alsof er geen afronding gebeurd was (dit is de "straight-through" in de naam). Alleen voor `alpha` zelf wordt een aparte, wél betekenisvolle gradiënt berekend: buiten het clip-bereik (`i = input.abs() > 1.0`) duwt de gradiënt `alpha` in de richting van het teken van de clip-overtreding (dus `alpha` groeit als er veel clipping is); binnen bereik wordt het verschil tussen gekwantiseerde en originele waarde gebruikt.

Dit stelt het netwerk in staat om **tijdens training te leren** hoe goed het met afgeronde gewichten kan werken, en zelfs de clip-drempel `alpha` optimaal te zetten — dat is precies wat QAT onderscheidt van simpele PTQ.

`weight_quantize_fn` — de `nn.Module`-wrapper om `_WeightQuantSTE`:

```
class weight_quantize_fn(nn.Module):
    def __init__(self, w_bit: int, power: bool = True) -> None:
        super().__init__()
        if w_bit != 32 and not (0 < w_bit <= 8):
            raise ValueError("w_bit must be 32 or in 1..8")
        self.w_bit = w_bit
        self.power = power and (3 <= w_bit <= 8)
        self.b = w_bit - 1
        self.grids = build_power_value(self.b, additive=True) if self.power else None
        self.register_parameter("wgt_alpha", Parameter(torch.tensor(3.0)))

    @torch.no_grad()
    def init_alpha(self, weight: torch.Tensor) -> None:
        alpha = weight.detach().abs().max()
        if alpha > 0:
            self.wgt_alpha.data.fill_(float(alpha))

    def forward(self, weight: torch.Tensor) -> torch.Tensor:
        if self.w_bit == 32:
            return weight
        return _WeightQuantSTE.apply(weight, self.wgt_alpha, self.b, self.grids, self.power)
```

`wgt_alpha` is een **leerbare parameter** (`nn.Parameter`) — dit betekent dat de clip-schaal tijdens training kan meebewegen (via de optimizer, zie `quant_alpha_parameters()` in `algorithms/qat/calibration.py`). `power` wordt alleen echt geactiveerd voor `3 <= w_bit <= 8` (bij lagere bit-breedtes heeft de additieve constructie geen zin). `init_alpha()` initialiseert de schaal op de max-abs-waarde van de echte gewichten — een goede startwaarde vóórdat training/eenmalige PTQ-toepassing begint.

`_ActQuantSTE` / `act_quantize_fn` — hetzelfde principe, maar voor **activaties** (tussenresultaten na elke laag) in plaats van gewichten:

```
class _ActQuantSTE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, alpha, b, grid, signed, power):
        input = input.div(alpha)
        if signed:
            ... # zelfde als weight-STE
        else:
            input_c = input.clamp(min=0, max=1) # bv. na ReLU: alleen positief
            ...
        ...

class act_quantize_fn(nn.Module):
    def __init__(self, a_bit, signed=True, power=True) -> None:
        ...
        self.register_buffer("calib_val", torch.tensor(0.0))

    def forward(self, x):
        if self.a_bit == 32:
            return x
        if self.calibrating:
            self._observe(x)
            return x
        return _ActQuantSTE.apply(x, self.act_alpha, self.b, self.grid, self.signed, self.power)

    def start_calibration(self, method="percentile", q=0.9999) -> None:
        self.calib_method = method
        self.calib_q = q
        self.calib_val = torch.zeros_like(self.calib_val)
        self.calibrating = True

    @torch.no_grad()
    def _observe(self, x) -> None:
        x = x.detach()
```

```

if self.calib_method == "max":
    v = x.abs().max()
else:
    flat = x.abs().flatten().float()
    if flat.numel() > (1 << 20):
        flat = flat[torch.randint(0, flat.numel(), (1 << 20,), device=flat.device)]
    v = torch.quantile(flat, self.calib_q)
self.calib_val = torch.maximum(self.calib_val.to(v.device), v)

@torch.no_grad()
def finish_calibration(self) -> None:
    self.calibrating = False
    if float(self.calib_val) > 0:
        self.act_alpha.data.fill_(float(self.calib_val))

```

Belangrijke verschillen met gewichten: - **signed**-parameter — sommige activaties (na ReLU) zijn altijd ≥ 0 , dus die kunnen unsigned gekwantiseerd worden (b bits i.p.v. b-1 + teken), wat effectief meer resolutie geeft. - **Calibratie-modus** (calibrating=True) — vóóordat er getraind wordt, doet de laag een aantal "observatie"-forward-passes waarbij hij **geen** quantisatie toepast maar wel de bereikstatistiek (calib_val) bijhoudt: - method="max" — puur de max-absolute waarde die ooit gezien is. - method="percentile" (default) — het **99.99e percentiel** (calib_q) van de absolute activatie-waarden, robuuster tegen incidentele uitschieters dan een harde max. Bij zeer grote tensors wordt eerst (willekeurig) gesubsampled tot maximaal ~1M elementen ($1 << 20 = 1048576$) om het percentiel-berekenen betaalbaar te houden. - torch.maximum(...) zorgt dat over meerdere calibratie-batches heen het **maximum** van de geobserveerde waarden bewaard blijft (conservatieve schatting van het echte bereik). - Na calibratie (finish_calibration()) wordt act_alpha vastgezet op de geobserveerde waarde, en schakelt de laag over naar normale quantisatie-forward via _ActQuantSTE.

QuantConv2d — een drop-in vervanging voor nn.Conv2d:

```

class QuantConv2d(nn.Conv2d):
    """Drop-in quantized Conv2d for YOLOv5 QAT (weights + optional activations)."""

    def __init__(self, in_channels, out_channels, kernel_size, ..., w_bit=8, a_bit=8, scheme="uniform",
quant_act=True) -> None:
        super().__init__(...)
        power = scheme == "apot"
        self.weight_quant = weight_quantize_fn(w_bit=w_bit, power=power)
        self.act_quant = act_quantize_fn(a_bit=a_bit, signed=True, power=power) if quant_act else None

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        if self.act_quant is not None:
            x = self.act_quant(x)
        weight_q = self.weight_quant(self.weight)
        return F.conv2d(x, weight_q, self.bias, self.stride, self.padding, self.dilation, self.groups)

    @classmethod
    def from_conv2d(cls, conv, w_bit=8, a_bit=8, scheme="uniform", quant_act=True) -> QuantConv2d:
        qc = cls(conv.in_channels, conv.out_channels, conv.kernel_size, conv.stride, conv.padding,
            conv.dilation, conv.groups, bias=conv.bias is not None,
            w_bit=w_bit, a_bit=a_bit, scheme=scheme, quant_act=quant_act)
        qc.weight.data.copy_(conv.weight.data)
        if conv.bias is not None:
            qc.bias.data.copy_(conv.bias.data)
        qc.weight_quant.init_alpha(qc.weight)
        return qc

```

Erft van nn.Conv2d (dus alle standaard Conv2d-attributen/gedrag blijven bestaan), maar overschrijft forward() om **eerst** de input te kwantiseren (als act_quant actief is), **dan** de gewichten te kwantiseren, en **dan pas** de gewone F.conv2d-berekening uit te voeren met de gekwantiseerde versies. from_conv2d() is de "converteer een bestaande laag"-fabrieksmethode — kopieert de originele gewichten over en initialiseert meteen wgt_alpha.

7.3 algorithms/qat/calibration.py

```

@torch.no_grad()
def calibrate_activations(model, data, num_batches=16, device="cpu", method="percentile", q=0.9999,
verbose=True) -> nn.Module:

```

```

"""Set each act_alpha from observed activation ranges before QAT/PTQ eval."""
acts = [m for m in model.modules() if isinstance(m, act_quantize_fn)]
if not acts:
    return model

was_training = model.training
model.eval().to(device)
for act in acts:
    act.start_calibration(method=method, q=q)

seen = 0
for batch in data:
    images = batch[0] if isinstance(batch, (list, tuple)) else batch
    was_uint8 = images.dtype == torch.uint8
    images = images.to(device).float()
    if was_uint8:
        images /= 255.0
    model(images)
    seen += 1
    if seen >= num_batches:
        break

for act in acts:
    act.finish_calibration()
if was_training:
    model.train()
return model

```

Orkestreert de calibratie-fase over het **hele model** in één keer: 1. Zoek alle `act_quantize_fn`-instanties in de module-boom (`model.modules()` doorloopt recursief alle sublagen). 2. Zet ze allemaal in calibratie-modus. 3. Voer een aantal batches door het model — dit triggert `act._observe()` in elke laag (via de normale `forward()`-aanroepen, zie sectie 7.2). 4. Zet ze weer uit calibratie-modus (`finish_calibration()` bevriest `act_alpha`). 5. Herstel de oorspronkelijke `training/eval`-modus van het model.

Detail: `uint8`-afbeeldingen worden automatisch naar `[0, 1]` float genormaliseerd (`/255.0`) — een veelvoorkomende afbeeldings-conventie die hier transparant wordt afgehandeld.

```

def quant_alpha_parameters(model: nn.Module) -> list[torch.nn.Parameter]:
    """Learnable clip thresholds for a dedicated optimizer param group."""
    alphas: list[torch.nn.Parameter] = []
    for module in model.modules():
        if isinstance(module, QuantConv2d):
            alphas.append(module.weight_quant.wgt_alpha)
            if module.act_quant is not None:
                alphas.append(module.act_quant.act_alpha)
    return alphas

```

Verzamelt **alleen** de `wgt_alpha/act_alpha`-parameters (niet de echte Conv-gewichten!) — dit is precies de parameterlijst die `pipeline/onnx_qat.py` aan de optimizer geeft. QAT in deze SDK traint dus **niet** de originele netwerkgewichten opnieuw, maar leert alleen de **clip-drempels** te optimaliseren rond de al-bestaande (bevroren, via teacher-vergelijking) gewichten — een lichtgewicht, snelle vorm van QAT.

7.4 `algorithms/qat/onnx_model.py` — graph-surgery

```

def quantize_onnx_model(model, w_bit=8, a_bit=8, scheme="uniform", first_layer_bit=8, quant_act=True,
verbose=True) -> nn.Module:
    """Swap Conv2d layers with QuantConv2d in any PyTorch module tree."""
    conv_layers: list[tuple[nn.Module, str, nn.Conv2d]] = []

    def _collect(parent, prefix=""):
        for name, child in parent.named_children():
            full_name = f"{prefix}.{name}" if prefix else name
            if isinstance(child, nn.Conv2d) and not isinstance(child, QuantConv2d):
                conv_layers.append((parent, name, child))
            else:
                _collect(child, full_name)

    _collect(model)

```

```

if not conv_layers:
    return model

for index, (parent, name, conv) in enumerate(conv_layers):
    if index == 0 and first_layer_bit is not None:
        bit_w, bit_a = first_layer_bit, first_layer_bit
    else:
        bit_w, bit_a = w_bit, a_bit
    setattr(parent, name, QuantConv2d.from_conv2d(conv, w_bit=bit_w, a_bit=bit_a, scheme=scheme,
quant_act=quant_act))
    return model

```

Loopt **recursief** door de hele module-boom (`_collect`, een geneste helper-functie die zichzelf aanroept voor elk kind-component) en verzamelt elke `nn.Conv2d`-laag samen met zijn **ouder-module** en **attribuutnaam** — dat is nodig omdat je in PyTorch een submodule vervangt via `setattr(parent_module, attr_name, nieuwe_module)`, en daarvoor moet je zowel de ouder als de naam weten (niet alleen de laag zelf).

Eerste-laag-uitzondering: de allereerste Conv-laag in het model (meestal de "stem" die ruwe pixels verwerkt) kan een **apart, vaak hoger** bit-budget krijgen via `first_layer_bit` — een bekende praktijk omdat de eerste laag typisch gevoeliger is voor agressieve quantisatie (het verwerkt nog-ongefilterde ruwe input, in tegenstelling tot diepere lagen die al robuustere features zien). `first_layer_bit=None` betekent: laat de eerste laag helemaal in float32 (niet quantiseren).

```

def fold_quant_conv2d(module: QuantConv2d) -> nn.Conv2d:
    """Bake fake-quant weights into a standard Conv2d for ONNX export."""
    with torch.no_grad():
        weight_q = module.weight_quant(module.weight).detach().clone()
    conv = nn.Conv2d(module.in_channels, module.out_channels, module.kernel_size, module.stride,
        module.padding, module.dilation, module.groups, bias=module.bias is not None)
    conv.weight.data.copy_(weight_q)
    if module.bias is not None:
        conv.bias.data.copy_(module.bias.detach())
    return conv

def fold_qat_model_for_export(model: nn.Module) -> nn.Module:
    """Replace QuantConv2d layers with folded float Conv2d for ONNX export."""
    model.eval()
    for module in model.modules():
        for child_name, child in list(module.named_children()):
            if isinstance(child, QuantConv2d):
                module._modules[child_name] = fold_quant_conv2d(child)
    return model

```

Waarom "folden" vóór export? `QuantConv2d` bevat interne logica (`weight_quant`, `act_quant`, STE-autograd-functies) die **niet** naar ONNX te exporteren is — ONNX kent geen "Straight-Through Estimator"-operator. `fold_quant_conv2d()` past daarom **één keer** de geleerde quantisatie toe op de gewichten (`weight_quant(weight)`, zonder gradient) en bakt het resultaat in een **kale, standaard `nn.Conv2d`** — die exporteert wél probleemloos naar ONNX via `torch.onnx.export`. Het model dat je krijgt na `fold...` heeft dus **dezelfde output-waarden** als tijdens de laatste trainings-forward-pass, maar zonder de quantisatie-specifieke laag-machinerie.

8. Pipeline: QAT end-to-end

`pipeline/onnx_qat.py` — `run_onnx_qat_pipeline()`

Dit is de **enige plek** waar alle QAT-onderdelen (loader, algorithms, calibratie) samenkomen tot één werkende trainings-run. Laten we de flow volgen.

Resultaat-datastructuur:

```

@dataclass
class OnnxQATPipelineResult:
    """Artifacts produced by universal ONNX QAT."""

```

```

output_dir: Path
output_model: Path
manifest_path: Path | None = None
steps: list[dict[str, Any]] = field(default_factory=list)
elapsed_seconds: float = 0.0

def to_dict(self) -> dict[str, Any]:
    return {"status": "success", "quantized": True, "format": "onnx",
            "output_dir": str(self.output_dir), "output_model": str(self.output_model),
            "manifest": str(self.manifest_path) if self.manifest_path else None,
            "steps": self.steps, "elapsed_seconds": round(self.elapsed_seconds, 2)}

```

`steps` is een **audit-trail** — elke fase van de pipeline hangt hier een dict aan (bijv. `{"step": "qat_train", "epoch": 1, "loss": 0.0023}`), zodat de klant/log precies kan zien wat er gebeurd is en hoe de loss zich ontwikkelde.

Data-voorbereiding — afbeeldingen inladen en normaliseren:

```

def _load_image(path, *, height, width) -> torch.Tensor:
    from PIL import Image
    img = Image.open(path).convert("RGB").resize((width, height))
    tensor = torch.from_numpy(__import__("numpy").array(img, dtype="float32"))
    return tensor.permute(2, 0, 1) / 255.0

def _coerce_sample(sample, *, height, width) -> torch.Tensor:
    if isinstance(sample, (str, Path)):
        return _load_image(sample, height=height, width=width)
    if isinstance(sample, torch.Tensor):
        tensor = sample.float()
    else:
        import numpy as np
        tensor = torch.from_numpy(np.asarray(sample, dtype=np.float32))
    if tensor.ndim == 3:
        if tensor.shape[0] not in (1, 3) and tensor.shape[-1] in (1, 3):
            tensor = tensor.permute(2, 0, 1)
        tensor = F.interpolate(tensor.unsqueeze(0), size=(height, width), mode="bilinear",
                                align_corners=False).squeeze(0)
    elif tensor.ndim == 4:
        tensor = tensor[0]
    return tensor

def _iter_training_batches(calibration_data, *, batch_size, height, width, max_batches) ->
    Iterator[torch.Tensor]:
    batch = []
    seen = 0
    for sample in calibration_data:
        batch.append(_coerce_sample(sample, height=height, width=width))
        if len(batch) >= batch_size:
            yield torch.stack(batch, dim=0)
            batch = []
            seen += 1
            if max_batches is not None and seen >= max_batches:
                return
    if batch:
        yield torch.stack(batch, dim=0)

```

`_coerce_sample()` is een **flexibele input-normalisatie**: accepteert bestandspaden (laadt met PIL), al-geladen tensors, of numpy-arrays. Detecteert automatisch **HWC vs CHW**-layout (`shape[-1] in (1, 3)` -> nog HWC, dus permute naar CHW) en herschaalt altijd naar de verwachte (`height, width`) met bilineaire interpolatie. `_iter_training_batches()` is een generator die losse samples groepeer in batches van `batch_size`, tot een `max_batches`- limiet — dit voorkomt dat een oneindige of zeer grote data-iterator de hele trainingsloop laat vastlopen.

Loss-functie — kennis-distillatie via MSE:

```

def _output_mse(student, teacher) -> torch.Tensor:
    if isinstance(student, (tuple, list)):
        student = student[0]
    if isinstance(teacher, (tuple, list)):
        teacher = teacher[0]
    if not isinstance(student, torch.Tensor):
        raise QuantizationError("QAT model output is not a tensor; unsupported architecture.")
    ...

```

```

if student.shape != teacher.shape:
    student = student.reshape(teacher.shape)
return F.mse_loss(student, teacher)

```

Belangrijk architectuur-inzicht: deze QAT-implementatie gebruikt **geen ground-truth labels** (geen classificatie- of detectie-loss) — in plaats daarvan wordt het **gequantiseerde model** (student) getraind om zo dicht mogelijk bij de output van het **ongewijzigde originele model** (teacher) te komen, via simpele MSE (`F.mse_loss`). Dit heet **zelf-distillatie**: het originele model "leert" het gequantiseerde model zijn eigen gedrag na te bootsen. Voordeel: werkt voor **elk** vision-model (classificatie, detectie, segmentatie, pose) zonder dat de SDK per taak een aparte loss-functie hoeft te kennen — je hebt alleen ongelabelde representatieve afbeeldingen nodig.

Hoofd-functie:

```

def run_onnx_qat_pipeline(*, config, model, output_dir, calibration_data, epochs=None,
                        batch_size=8, imgsz=None, device="auto", learning_rate=1e-4,
                        max_calib_samples=None, opset=17, domain=None, verbose=True) ->
OnnxQATPipelineResult:
    validate_calibration("qat", True, calibration_data)
    model_path = Path(model)
    if model_path.suffix.lower() != ".onnx":
        raise QuantizationError(...)

    out = Path(output_dir); out.mkdir(parents=True, exist_ok=True)
    dev = resolve_torch_device(device)
    resolved_epochs = epochs if epochs is not None else config.qat_epochs
    input_name, input_shape = onnx_input_spec(model_path, batch_size=batch_size, imgsz=imgsz)
    _, channels, height, width = (input_shape + [3, 224, 224])[:4]

    steps = []
    t0 = time.time()

    float_model = load_onnx_as_torch(model_path).to(dev).eval()
    teacher = copy.deepcopy(float_model)
    for param in teacher.parameters():
        param.requires_grad = False

    qat_model = quantize_onnx_model(float_model, w_bit=config.qat_wbit, a_bit=config.qat_abits,
                                   scheme=config.qat_scheme, first_layer_bit=config.qat_first_layer_bit,
                                   quant_act=config.qat_quant_act, verbose=verbose).to(dev)

    max_batches = max(resolved_epochs, 1) * max(config.qat_calib_batches, 1)
    if max_calib_samples is not None:
        max_batches = min(max_batches, max(1, max_calib_samples // max(batch_size, 1)))

    calib_batches = list(_iter_training_batches(calibration_data, batch_size=batch_size,
                                                height=height, width=width,
max_batches=config.qat_calib_batches))
    if not calib_batches:
        raise QuantizationError("calibration_data produced no usable image batches.")

    calibrate_activations(qat_model, calib_batches, num_batches=config.qat_calib_batches,
                        device=dev, method=config.qat_calib_method, q=config.qat_calib_q, verbose=verbose)
    steps.append({"step": "act_calibration", "batches": len(calib_batches)})

    trainable = quant_alpha_parameters(qat_model)
    if not trainable:
        raise QuantizationError("No QAT layers found in ONNX model (no Conv2d ops to quantize).")

    optimizer = torch.optim.Adam(trainable, lr=learning_rate)
    qat_model.train(); teacher.eval()

    train_batches = list(_iter_training_batches(calibration_data, batch_size=batch_size,
                                                height=height, width=width, max_batches=max_batches))

    for epoch in range(resolved_epochs):
        epoch_loss = 0.0
        for batch in train_batches:
            batch = batch.to(dev)
            with torch.no_grad():
                target = teacher(batch)
            output = qat_model(batch)

```

```

        loss = _output_mse(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += float(loss.detach())
    steps.append({"step": "qat_train", "epoch": epoch + 1, "loss": round(epoch_loss /
max(len(train_batches), 1), 6)})
    ...

```

De volledige flow, in woorden:

1. **Valideer input** — moet ONNX zijn, calibratiedata mag niet leeg zijn.
2. **Lees input-shape** uit de graph (`onnx_input_spec`).
3. **Laad twee kopieën** van het model: - `float_model` -> wordt straks omgezet naar het QAT-model. - `teacher = copy.deepcopy(float_model)` -> blijft **exact het origineel**, met `requires_grad = False` op alle parameters (nooit getraind, puur referentie voor de MSE-loss).
4. **Injecteer QAT-lagen**: `quantize_onnx_model()` vervangt elke `nn.Conv2d` door `QuantConv2d` (sectie 7.4).
5. **Bereken hoeveel batches** nodig zijn: `qat_calib_batches` × aantal epochs, eventueel begrensd door `max_calib_samples` als de gebruiker een harde limiet op het aantal samples zet.
6. **Bouw calibratie-batches** en roep `calibrate_activations()` aan — dit zet de initiële `act_alpha`-waarden (sectie 7.3) vóórdat er getraind wordt.
7. **Verzamel trainbare parameters** — alleen `wgt_alpha/act_alpha` (sectie 7.3, `quant_alpha_parameters`). Als die lijst leeg is (model had geen `Conv2d`-lagen), stopt de pipeline met een duidelijke fout.
8. **Trainingslus**: voor elke batch, laat de **teacher** (zonder gradient) de "juiste" output berekenen, laat het **student-QAT-model** dezelfde batch verwerken, bereken de MSE tussen beide, en update alleen de `alpha`-parameters met Adam. Per epoch wordt de gemiddelde loss bijgehouden.

Export-fase:

```

fold_qat_model_for_export(qat_model)
sample = torch.zeros(1, channels, height, width, device=dev)
export_path = out / f"{model_path.stem}_qat.onnx"
_export_onnx_model(qat_model.cpu(), sample=sample.cpu(), output_path=export_path, input_name=input_name,
opset=opset)

output_model = finalize_onnx_deliverable(export_path, out, config_dict=config.model_dump(mode="json"),
                                         source_model=model_path, domain=domain,
                                         extra={"scheme": config.qat_scheme, "quantization_type": "qat_onnx",
"epochs": resolved_epochs})
manifest_path = out / "r3al_quant_manifest.json"
steps.append({"step": "onnx_export", "output_model": str(output_model)})

meta = {"pipeline": "onnx_qat", "quantized": True, "format": "onnx", "config": config.model_dump(mode="json"),
        "source_model": str(model_path), "output_model": str(output_model), "manifest": str(manifest_path),
        "steps": steps}
(out / "r3al_quant_meta.json").write_text(json.dumps(meta, indent=2), encoding="utf-8")

return OnnxQATPipelineResult(output_dir=out, output_model=output_model,
                             manifest_path=manifest_path if manifest_path.is_file() else None,
                             steps=steps, elapsed_seconds=time.time() - t0)

```

Na de training: **fold** de `QuantConv2d`-lagen terug naar kale `Conv2d` (sectie 7.4, nodig omdat `torch.onnx.export` geen custom `autograd-Function` kan exporteren), exporteer met `torch.onnx.export` (dynamische batch-as zodat het resultaat met elke batch-grootte werkt), valideer met `onnx.checker`, en schrijf tot slot het manifest + een uitgebreid `r3al_quant_meta.json` met de volledige `steps`-audit-trail.

`_export_onnx_model()` (interne helper):

```

def _export_onnx_model(model, *, sample, output_path, input_name, opset) -> None:
    import onnx
    model.eval()
    torch.onnx.export(model, sample, str(output_path), opset_version=opset,
                      input_names=[input_name], output_names=["output"],

```

```
dynamic_axes={input_name: {0: "batch"}, "output": {0: "batch"}}}
onnx.checker.check_model(onnx.load(str(output_path)))
```

`dynamic_axes` markeert dimensie 0 (de batch-as) van zowel input als output als dynamisch — zodat het geëxporteerde model met batch-grootte 1, 4, 32, etc. werkt zonder opnieuw te moeten exporteren. `onnx.checker.check_model` valideert de graph-structuur (voorkomt dat een corrupt/incomplete ONNX- bestand als "succesvol" wordt gerapporteerd).

9. Export

Deze laag lost een ander probleem op dan quantisatie: **"ik heb een model in formaat X, ik wil ONNX."** Nodig omdat quantisatie altijd ONNX-input verwacht, maar klanten hun modellen in allerlei formaten hebben (YOLO .pt, TensorFlow SavedModel, TFLite, Paddle, ...).

9.1 `export/base.py` — gedeelde types

```
@dataclass
class ExportOptions:
    """Options passed to export adapters."""
    input_shape: tuple[int, ...] | None = None
    opset: int = 17
    imgsz: int | tuple[int, int] = 640
    simplify: bool = True
    dynamic_batch: bool = False
    input_names: list[str] | None = None
    output_names: list[str] | None = None
    extra: dict[str, Any] = field(default_factory=dict)

@dataclass
class ExportResult:
    """Deliverable from an export adapter."""
    onnx_path: Path
    output_dir: Path
    source_model: Path
    source_format: str
    export_tool: str
    manifest_path: Path | None = None

    def to_dict(self) -> dict[str, Any]:
        return {"status": "success", "output_model": str(self.onnx_path), "output_dir": str(self.output_dir),
                "source_model": str(self.source_model), "source_format": self.source_format,
                "export_tool": self.export_tool, "format": "onnx",
                "manifest": str(self.manifest_path) if self.manifest_path else None}
```

`ExportOptions` bundelt alle mogelijke export-parameters in één object — elke adapter gebruikt alleen de velden die het nodig heeft (bijv. `imgsz` alleen relevant voor Ultralytics/vision-modellen met vaste invoer-resolutie). `extra` is een vrije "vangnet"-dict voor adapter-specifieke opties (zoals Paddle's `model_filename/params_filename`).

```
def validate_onnx_model(onnx_path: Path) -> None:
    """Run ONNX structural validation on an exported model."""
    import onnx
    onnx.checker.check_model(str(onnx_path))
```

Elke adapter roept dit **na** de conversie aan — een gemeenschappelijke kwaliteitscontrole die garandeert dat wat je terugkrijgt daadwerkelijk een geldig ONNX-bestand is, ongeacht welke externe conversie-tool gebruikt werd.

9.2 `export/registry.py` — `resolve_adapter()`

```
_ADAPTERS: list[ExportAdapter] = [
    OnnxPassthroughAdapter(),
    UltralyticsAdapter(),
    PyTorchAdapter(),
    TensorFlowAdapter(),
```

```

PaddleAdapter(),
TFLiteAdapter(),
]

_SOURCE_ALIASES = {
    "yolo": "ultralytics", "torch": "pytorch", "tf": "tensorflow",
    "keras": "tensorflow", "paddlepaddle": "paddle",
}

```

Een **geordende lijst** van adapter-instanties (elk een simpele klasse met `can_handle()` + `to_onnx()`) plus een alias-tabel zodat gebruikers "vriendelijke" namen ("yolo", "torch") kunnen gebruiken in plaats van de exacte interne adapter-naam.

```

def resolve_adapter(path: Path, *, source: str | None = None) -> ExportAdapter:
    """Pick the first adapter that can handle the given model path."""
    normalized_source = _normalize_source(source)
    if normalized_source is not None:
        for adapter in _ADAPTERS:
            if adapter.name == normalized_source:
                if adapter.can_handle(path, source=normalized_source):
                    return adapter
                raise UnsupportedConfigError(
                    f"source='{normalized_source}' cannot handle '{path.name}'. "
                    f"Supported extensions: {_extensions_for(adapter)}."
                )
        raise UnsupportedConfigError(f"Unknown export source '{source}'. Choose from: {[a.name for a in _ADAPTERS]}")

    for adapter in _ADAPTERS:
        if adapter.can_handle(path):
            return adapter

    if path.suffix.lower() in {".pt", ".pth"}:
        raise UnsupportedConfigError(
            f"Could not auto-detect export source for '{path.name}'. "
            "Set source='ultralytics' for YOLO weights or source='pytorch' for a saved "
            "torch.nn.Module (with input_shape)."
        )
    raise UnsupportedConfigError(f"No export adapter for '{path.name}'. ...")

```

Twee paden: 1. **Expliciete source** — zoek de met-naam-overeenkomende adapter op en verifieer dat die het bestand ook echt aankan (anders duidelijke fout met de verwachte extensies). 2. **Auto-detectie** — probeer elke adapter in volgorde met `can_handle()` (kijkt meestal alleen naar bestandsextensie/mapstructuur) totdat er één "ja" zegt.

De speciale `.pt/.pth`-fout bestaat omdat **zowel** Ultralytics-checkpoints **als** generieke PyTorch-`nn.Module`-saves dezelfde extensie gebruiken — de SDK kan niet automatisch raden welke van de twee het is, dus wordt de gebruiker gevraagd expliciet `source=` op te geven.

9.3 De adapters

Elke adapter implementeert hetzelfde patroon: `can_handle(path)` -> bepaal geschiktheid puur op extensie/structuur; `to_onnx(path, output_dir, options)` -> voer de daadwerkelijke conversie uit, valideer, schrijf manifest.

adapters/onnx.py — **OnnxPassthroughAdapter**: voor als het bestand al ONNX is. Kopieert het bestand (`shutil.copy2`) en valideert — nuttig zodat elke actie (`export`, `quantize` met `auto_export`) uniform door dezelfde export-laag kan gaan, zelfs als er eigenlijk niets te converteren valt.

adapters/ultralytics.py — **UltralyticsAdapter**: gebruikt de `ultralytics`-package (`YOLO(path).export(format="onnx", imgsz=..., simplify=...)`). Bijzonderheid: `os.chdir(output_dir)` vóór het aanroepen van `.export()` — de Ultralytics-library schrijft soms bestanden relatief aan de working directory, dus wordt die tijdelijk verplaatst en na afloop (in een `finally`) teruggezet.

```

def to_onnx(self, path, output_dir, options) -> ExportResult:
    if not path.is_file():
        raise ModelLoadError(f"Ultralytics weights not found: {path}")
    from ultralytics import YOLO
    output_dir.mkdir(parents=True, exist_ok=True)

```

```

cwd = Path.cwd()
try:
    os.chdir(output_dir)
    export_result = YOLO(str(path)).export(format="onnx", imgsz=options.imgsz,
                                           simplify=options.simplify, opset=options.opset)
finally:
    os.chdir(cwd)
...

```

adapters/pytorch.py — PyTorchAdapter: voor generieke `torch.nn.Module`-checkpoints. Vereist expliciet `input_shape` (kan niet automatisch afgeleid worden zoals bij YOLO, waar de architectuur bekende conventies volgt):

```

def to_onnx(self, path, output_dir, options) -> ExportResult:
    if options.input_shape is None:
        raise UnsupportedConfigError("Generic PyTorch export requires input_shape, e.g. [1, 3, 224, 224].")
    loaded = torch.load(path, map_location="cpu", weights_only=False)
    if not isinstance(loaded, torch.nn.Module):
        raise ModelLoadError(
            f"Checkpoint '{path}' is not a full torch.nn.Module. "
            "Save the model object (torch.save(model, path)) or use source='ultralytics' for YOLO weights."
        )
    ...

```

Belangrijk: `torch.load(..., weights_only=False)` laadt het **volledige object** (niet alleen een `state_dict`) — dit vereist dat het `.pt`-bestand gemaakt is met `torch.save(model, path)` (het hele model-object), niet `torch.save(model.state_dict(), path)` (alleen de gewichten zonder architectuur). De foutmelding legt dat expliciet uit als het misgaat.

adapters/tensorflow.py — TensorFlowAdapter: drie sub-paden via `tf2onnx`: `SavedModel`-directory, Keras (`.h5/.keras`), of bevroren graph (`.pb`, vereist expliciete `input_names/output_names` omdat een frozen graph geen ingebouwde signature-metadata heeft zoals `SavedModel/Keras` dat wel hebben).

adapters/tflite.py — TFLiteAdapter: dunne wrapper om `tflite2onnx.convert()`.

adapters/paddle.py — PaddleAdapter: het meest complexe qua bestands-resolutie, omdat Paddle **twee gekoppelde bestanden** gebruikt (`model.pdmodel` + `model.pdiparams`, of ouder `model.json` + dezelfde `.pdiparams`):

```

def _find_pair_in_dir(self, directory: Path) -> tuple[Path, Path] | None:
    for suffix in (".pdmodel", ".json"):
        for model_file in sorted(directory.glob(f"*{suffix}")):
            params_file = model_file.with_suffix(".pdiparams")
            if params_file.is_file():
                return model_file, params_file
    return None

def _paired_model_file(self, params_file: Path) -> Path | None:
    for suffix in (".pdmodel", ".json"):
        candidate = params_file.with_suffix(suffix)
        if candidate.is_file():
            return candidate
    return None

```

De adapter accepteert **elk van de drie ingangen** (`map`, `.pdiparams`, of het model-bestand zelf) en zoekt automatisch het bijbehorende paar — de gebruiker hoeft dus niet zelf uit te zoeken welk bestand bij welk hoort.

9.4 export/__init__.py — de publieke functies

```

def export_to_onnx(model, output_dir="/exported", *, source=None, input_shape=None, opset=17,
                  imgsz=640, simplify=True, dynamic_batch=False, input_names=None,
                  output_names=None, export_opts=None) -> ExportResult:
    """Export a vision model to ONNX using the best matching adapter."""
    path = Path(model)
    options = _build_options(...)
    adapter = resolve_adapter(path, source=source)
    return adapter.to_onnx(path, Path(output_dir), options)

```

De hoofdingang voor expliciete export-jobs (`action: "export"` in `handler.py`).

```
def maybe_export_to_onnx(model, *, auto_export=False, output_dir=None, source=None, export_opts=None,
**kwargs) -> Path:
    """Return an ONNX path, exporting first when ``auto_export`` is enabled."""
    path = Path(model)
    if path.suffix.lower() == ".onnx":
        return path
    if not auto_export:
        return path
    ...
    result = export_to_onnx(path, resolved_output, source=resolved_source, export_opts=opts)
    return result.onnx_path
```

Dit is de "slimme" variant, gebruikt door **elke** `quantize/qat_pipeline`-actie in `handler.py` via `_resolve_model_path()`: als het model **al** ONNX is, wordt er niets gedaan (retourneer gewoon het pad). Als het **geen** ONNX is maar `auto_export=True` staat, wordt eerst automatisch geconverteerd. Als het geen ONNX is en `auto_export` niet gezet is, wordt het pad **ongewijzigd** teruggegeven — de latere PTQ/QAT-validatie (`QuantConfig/job_input.py`) vangt dan de fout op met een duidelijke melding dat er een `.onnx`-bestand verwacht werd.

10. Manifest, calibratie en benchmark

10.1 `manifest.py` — het "bewijs"-systeem

Elke quantisatie-actie produceert **twee soorten bewijs**: een extern JSON- bestand (`r3al_quant_manifest.json`) en interne metadata **in** het ONNX-bestand zelf.

```
def build_quant_manifest(*, config, source_model, output_model, output_format="onnx",
                        domain=None, extra=None) -> dict[str, Any]:
    """Build the customer-facing proof that a model was quantized by R3AL Quant."""
    manifest = {
        "quantized": True, "producer": "R3AL.ai Quant SDK", "format": output_format,
        "source_model": str(source_model), "output_model": str(output_model),
        "method": config.get("method"), "config": config,
        "created_at": datetime.now(timezone.utc).isoformat(),
    }
    if domain:
        manifest["domain"] = domain
    if extra:
        manifest.update(extra)
    return manifest
```

`config` bevat de **volledige** `QuantConfig` als dict (via `config.model_dump(mode="json")` in de callers) — dus het manifest is zelfstandig genoeg om exact te reproduceren welke instellingen gebruikt zijn (schema, bits, `apot_smoothing`, etc.), zonder dat je de originele job-payload nog hoeft te bewaren. `extra` wordt per-backend ingevuld (bijv. `quantized_layers`, `quantization_type`) en overschrijft/vult aan op de basis-velden.

```
def stamp_onnx_metadata(onnx_path: Path, metadata: dict[str, str]) -> None:
    """Embed R3AL quantization markers in an ONNX model file."""
    import onnx
    model = onnx.load(str(onnx_path))
    while len(model.metadata_props):
        model.metadata_props.pop()
    for key, value in metadata.items():
        prop = model.metadata_props.add()
        prop.key = key
        prop.value = value
    onnx.save(model, str(onnx_path))
```

ONNX ondersteunt een ingebouwd `metadata_props`-veld (lijst van key-value-string-paren) dat **in het bestand zelf** reist, ongeacht waar het naartoe gekopieerd wordt. Deze functie **wist eerst alle bestaande** metadata (`while len(...): pop()`) en zet dan de R3AL-eigen sleutels (`r3al_quantized`, `r3al_method`, `r3al_bits`, optioneel `r3al_domain`). Dit betekent dat je met alleen het `.onnx`-bestand (zonder het aparte manifest-JSON) al kunt zien of/hoe het gequantiseerd is — handig als bestanden los van elkaar raken.

```
def resolve_quantized_onnx_bundle(model_path: str | Path) -> tuple[Path, dict[str, Any]]:
    """Load manifest (preferred) or legacy meta for a quantized ONNX bundle directory."""
    path = Path(model_path)
    manifest_path = path / MANIFEST_FILENAME if path.is_dir() else path.parent / MANIFEST_FILENAME
    legacy_path = path / LEGACY_META_FILENAME if path.is_dir() else path.parent / LEGACY_META_FILENAME

    if manifest_path.is_file():
        meta = json.loads(manifest_path.read_text(encoding="utf-8"))
        onnx_file = Path(meta["output_model"])
        if not onnx_file.is_file():
            onnx_file = manifest_path.parent / onnx_file.name
        return onnx_file, meta

    if legacy_path.is_file():
        ...
    raise FileNotFoundError(f"No {MANIFEST_FILENAME} or {LEGACY_META_FILENAME} found near {path}")
```

De "omgekeerde" operatie: gegeven een pad (map óf los bestand), vind het manifest en haal daaruit het echte ONNX-bestandspad. De fallback `if not onnx_file.is_file(): onnx_file = manifest_path.parent / onnx_file.name` maakt dit **verplaatsbaar** — zelfs als de map is hernoemd/verplaatst sinds het manifest geschreven werd (het absolute pad in het manifest zou dan niet meer kloppen), wordt het bestand toch gevonden zolang het naast het manifest staat. Dit wordt gebruikt door **elke** backend's `load()`-methode.

10.2 calibration.py — gedeelde calibratie-helpers

```
def validate_calibration(method, requires_calibration, calibration_data) -> None:
    if requires_calibration and calibration_data is None:
        raise CalibrationRequiredError(method)

def iter_batches(data, max_samples=None) -> Iterator[Any]:
    """Yield calibration samples up to max_samples."""
    count = 0
    for sample in data:
        yield sample
        count += 1
        if max_samples is not None and count >= max_samples:
            break

def vision_calibration_generator(data, transform=None, max_samples=100) -> Iterator[Any]:
    """Yield image tensors for vision model calibration."""
    for sample in iter_batches(data, max_samples):
        yield transform(sample) if transform is not None else sample
```

Kleine, generieke bouwstenen: `validate_calibration` is de centrale "heb-ik-data-nodig"-check gebruikt door zowel `ONNXBackend` als `pipeline/onnx_qat.py`. `iter_batches/vision_calibration_generator` zijn simpele begrensde generatoren — voorkomen dat een oneindige of zeer grote data-iterator (bijv. een dataset-loader) volledig doorlopen wordt als er maar 100 samples nodig zijn.

10.3 benchmark.py — Benchmark

```
class Benchmark:
    """Compare latency and optional quality between original and quantized ONNX models."""

    def __init__(self, config: QuantConfig) -> None:
        self.config = config

    def run(self, original_model, quantized_model, eval_data, *, warmup_runs=3, benchmark_runs=10,
            quality_fn=None, quality_metric="score") -> Any:
        backend = get_backend(self.config)
        quantized_bundle = backend.load(quantized_model)

        samples = list(eval_data)
        if not samples:
            raise BenchmarkError("eval_data must contain at least one image sample")

        sample = samples[0]
        tensor = sample if torch.is_tensor(sample) else torch.as_tensor(sample, dtype=torch.float32)
```

```

    if tensor.ndim == 3:
        tensor = tensor.unsqueeze(0)

    if "session" not in quantized_bundle:
        raise BenchmarkError("Quantized model must be an ONNX bundle with inference session.")

    return self._benchmark_onnx(original_model, quantized_bundle, tensor, quality_metric, warmup_runs,
benchmark_runs)

```

Gebruikt **dezelfde backend/load()-logica** als normale inference — laadt het gequantiseerde model via het manifest-systeem, pakt het **eerste** sample uit `eval_data` als representatieve benchmark-input (voegt een batch-dimensie toe als dat nog niet gebeurd is).

```

def _benchmark_onnx(self, original_model, quantized_bundle, tensor, quality_metric, warmup_runs,
benchmark_runs) -> Any:
    session = quantized_bundle["session"]
    input_name = session.get_inputs()[0].name
    np_input = tensor.numpy().astype(np.float32)

    def _run_onnx() -> None:
        session.run(None, {input_name: np_input})

    lat_quant = self._measure_latency(_run_onnx, warmup_runs, benchmark_runs)

    lat_orig = 0.0
    try:
        import onnxruntime as ort
        orig_session = ort.InferenceSession(str(original_model), providers=["CUDAExecutionProvider",
"CPUExecutionProvider"])
        orig_input = orig_session.get_inputs()[0].name

        def _run_orig() -> None:
            orig_session.run(None, {orig_input: np_input})

        lat_orig = self._measure_latency(_run_orig, warmup_runs, benchmark_runs)
    except Exception:
        pass

    return BenchmarkResult(latency_ms_original=lat_orig, latency_ms_quantized=lat_quant,
        latency_speedup=lat_orig / lat_quant if lat_orig > 0 and lat_quant > 0 else 0.0,
        quality_metric=quality_metric, metadata={"original_model": str(original_model)})

```

Meet de latency van **beide** modellen met dezelfde input-tensor. De `try/except` rond het origineel-benchmarken is bewust **stil** (`except Exception: pass`) — als het originele model om wat voor reden dan ook niet te laden is (bijv. verwijderd, verkeerd pad), wordt de speedup gewoon `0.0` in plaats van de hele benchmark te laten falen; je krijgt dan nog steeds de **gequantiseerde** latency terug, wat vaak het belangrijkste cijfer is.

```

@staticmethod
def _measure_latency(fn, warmup, runs) -> float:
    for _ in range(warmup):
        fn()
    if torch.cuda.is_available():
        torch.cuda.synchronize()
    start = time.perf_counter()
    for _ in range(runs):
        fn()
    if torch.cuda.is_available():
        torch.cuda.synchronize()
    elapsed_ms = (time.perf_counter() - start) * 1000
    return elapsed_ms / runs

```

Klassiek benchmark-patroon: **warmup-runs** eerst (om JIT/cache/CUDA- initialisatie-kosten niet in de meting te laten sluipen), dan `torch.cuda.synchronize()` vóór en na de gemeten sectie (nodig omdat CUDA- operaties **asynchroon** zijn — zonder `synchronize` zou je alleen de tijd meten om de GPU-calls te *queuen*, niet om ze daadwerkelijk uit te voeren). Het resultaat is de **gemiddelde tijd per run** in milliseconden.

11. job_input.py

Deze module is de **vertaallaa**g tussen "wat een gebruiker/klant typt in JSON" en "wat de rest van de SDK verwacht" (een genormaliseerde job met een losstaand `config`-object).

Config-aliassen — kortere namen voor veelgebruikte velden:

```
_CONFIG_KEYS = frozenset({"model_type", "mode", "bits", "method", "backend", "target_device",
                          "apot_smoothing", "apot_per_channel", "qat_wbit", "qat_abit", "qat_scheme",
                          "qat_first_layer_bit", "qat_quant_act", "qat_calib_batches",
                          "qat_calib_method", "qat_calib_q", "qat_epochs", "output_format"})

_CONFIG_ALIASES = {"scheme": "qat_scheme", "wbit": "qat_wbit", "abit": "qat_abit"}
```

Zo kan een gebruiker in de top-level payload gewoon `"wbit": 4` schrijven in plaats van het moeten nesten onder `"config": {"qat_wbit": 4}` — beide vormen werken.

Default output-directories, RunPod-bewust:

```
def _default_output_dir(action: str) -> str:
    if Path("/runpod-volume").is_dir():
        if action == "export": return "/runpod-volume/exported"
        if action == "quantize": return "/runpod-volume/quantized"
        return "/runpod-volume/qat_out"
    if action == "export": return "/tmp/r3al_export_output"
    if action == "quantize": return "/tmp/r3al_quant_output"
    return "/tmp/r3al_qat_pipeline"
```

Detecteert of de code op een **RunPod-worker** draait (die mount altijd `/runpod-volume` als persistente opslag) — zo niet, dan valt het terug op `/tmp`, geschikt voor lokaal testen zonder dat je zelf een `output_dir` hoeft op te geven.

PTQ-methode automatisch afleiden:

```
def _infer_ptq_method(model: str) -> str:
    if Path(model).suffix.lower() == ".onnx":
        return VisionMethod.WEIGHT_QUANT.value
    raise UnsupportedConfigError(
        "PTQ (action='quantize') requires an ONNX model (.onnx). "
        "For quantization-aware training use action='qat_pipeline'."
    )
```

Als een gebruiker geen `method` opgeeft, wordt automatisch `weight_quant` gekozen zodra het bestand `.onnx` is — de meeste gebruiksvriendelijke keuze omdat `weight_quant` geen calibratiedata nodig heeft en op elk ONNX-model werkt.

normalize_job_input() — de hoofd-orkestratiefunctie:

```
def normalize_job_input(raw: dict[str, Any]) -> dict[str, Any]:
    """Expand a simple client payload into the canonical handler job shape."""
    job = dict(raw)

    for alias, target in _CONFIG_ALIASES.items():
        if alias in job and target not in job:
            job[target] = job.pop(alias)

    action = job.get("action", "quantize")
    config = dict(job.pop("config", None) or {})
    for key in list(job.keys()):
        if key in _CONFIG_KEYS:
            config[key] = job.pop(key)

    if config.get("model_type") not in (None, "vision"):
        raise UnsupportedConfigError("Only vision ONNX quantization is supported.")

    if action == "qat_pipeline":
        config.setdefault("mode", QuantizationMode.QAT.value)
    else:
        config.setdefault("mode", QuantizationMode.PTQ.value)
        model = job.get("model") or job.get("original_model")
        if (action != "export" and config.get("mode") == QuantizationMode.PTQ.value
            and "method" not in config and model):
```

```

    model_path = Path(str(model))
    if model_path.suffix.lower() == ".onnx":
        config["method"] = _infer_ptq_method(model)
    elif job.get("auto_export"):
        config["method"] = VisionMethod.WEIGHT_QUANT.value
    else:
        config["method"] = _infer_ptq_method(model)

    if "method" in config and config["method"] not in VISION_METHODS:
        raise UnsupportedConfigError(f"Unknown method '{config['method']}'". Choose from:
{sorted(VISION_METHODS)}")

    _apply_config_defaults(config, action)
    if config:
        job["config"] = config

    if action in {"export", "quantize", "qat_pipeline"} and "output_dir" not in job:
        job["output_dir"] = _default_output_dir(action)

    if action == "export" and "model" not in job:
        raise UnsupportedConfigError("action='export' requires a model path.")

    _validate_onnx_input(job)
    return job

```

Stap voor stap:

1. **Aliassen oplossen** (`wbit` -> `qat_wbit`, etc.).
2. **Top-level config-velden verzamelen** — alles wat in `_CONFIG_KEYS` zit wordt uit de top-level payload gehaald en in een apart `config`-dict gestopt (samengevoegd met een eventueel al bestaand `"config": {...}`-veld).
3. **Mode bepalen** — `qat_pipeline`-actie -> altijd `mode=qat`. Anders `mode=ptq`, en als er geen `method` is opgegeven maar wel een `model`, probeer die automatisch af te leiden (met een speciaal geval voor `auto_export=True`, waar de methode alvast op `weight_quant` gezet wordt ook al is het bronbestand nog geen ONNX — de conversie gebeurt later in `handler.py::_resolve_model_path`).
4. `_apply_config_defaults()` vult verdere per-actie defaults in (zie hieronder).
5. **Default output-dir** invullen als niet opgegeven.
6. `_validate_onnx_input()` — laatste checkpoint vóórdat de job daadwerkelijk uitgevoerd wordt.

```

def _apply_config_defaults(config: dict[str, Any], action: str) -> None:
    config.setdefault("model_type", "vision")
    config.setdefault("output_format", "onnx")

    if action == "qat_pipeline":
        config["mode"] = QuantizationMode.QAT.value
        config.pop("method", None)
        config.setdefault("qat_scheme", "apot")
        config.setdefault("qat_epochs", 1)
        return

    config.setdefault("mode", QuantizationMode.PTQ.value)
    method = config.get("method")
    if method == "yolo_train":
        raise UnsupportedConfigError(
            "method='yolo_train' is removed. Use action='qat_pipeline' with mode='qat' (no method field
needed).")
    )
    if method == VisionMethod.WEIGHT_QUANT.value:
        config.setdefault("qat_scheme", "apot")
        config.setdefault("qat_wbit", 8)
    if method in ONNX_INPUT_METHODS:
        config.setdefault("bits", 8)

```

Interessant detail: `method == "yolo_train"` wordt expliciet **geweigerd** met een migratie-hint — dit is een teken van een **verwijderde, oudere API** (waarschijnlijk een eerdere versie waar YOLO-training een aparte methode was, inmiddels vervangen door de universele `qat_pipeline`-actie). Zulke expliciete "dit bestaat niet meer, gebruik X"-fouten

zijn waardevol voor backwards- compatibiliteit met bestaande integraties.

```
def _validate_onnx_input(job: dict[str, Any]) -> None:
    action = job.get("action", "quantize")
    config = job.get("config") or {}
    model = job.get("model")
    if not model:
        return

    model_path = Path(str(model))
    if action == "export":
        return # export accepteert elk formaat – dat is precies zijn doel

    auto_export = bool(job.get("auto_export"))
    if model_path.suffix.lower() != ".onnx" and auto_export:
        return # zal later geconverteerd worden

    if action == "quantize":
        if config.get("mode", QuantizationMode.PTQ.value) != QuantizationMode.PTQ.value:
            return
        method = config.get("method", VisionMethod.WEIGHT_QUANT.value)
        if method not in ONNX_INPUT_METHODS:
            return
        if model_path.suffix.lower() != ".onnx":
            raise UnsupportedConfigError(f"method='{method}' requires an ONNX input file (.onnx), got '{model}'")
        return

    if action != "qat_pipeline":
        return
    if model_path.suffix.lower() != ".onnx":
        raise UnsupportedConfigError("QAT (action='qat_pipeline') requires an ONNX model (.onnx).")
    if not job.get("calibration_data"):
        raise UnsupportedConfigError("QAT (action='qat_pipeline') requires calibration_data (training images).")
```

De laatste veiligheidscontrole vóórdat een job daadwerkelijk wordt uitgevoerd: geeft **vroegtijdige, duidelijke** foutmeldingen in plaats van pas diep in een backend te crashen met een cryptische ONNX-parse-fout.

12. `methods.py`

Een **statische catalogus** — puur data, geen logica die iets uitvoert. Wordt gebruikt door het `list_methods`-endpoint en door documentatie/tools om te weten welke methodes/parameters bestaan zonder de rest van de SDK te hoeven importeren of een model te laden.

```
_UNIVERSAL = "any_onnx_vision_model"

_PTQ_METHOD_SPECS: list[dict[str, Any]] = [
    {"mode": "ptq", "model_type": "vision", "method": "weight_quant", "scope": _UNIVERSAL,
     "bits": [3, 4, 5, 6, 7, 8], "requires_calibration": False, "input_format": "onnx",
     "output_format": "onnx", "description": "..."},
    {"mode": "ptq", "model_type": "vision", "method": "int8_dynamic", "scope": _UNIVERSAL,
     "bits": [8], "requires_calibration": False, ...},
    {"mode": "ptq", "model_type": "vision", "method": "int8_static", "scope": _UNIVERSAL,
     "bits": [8], "requires_calibration": True, ...},
]

_QAT_SPEC: dict[str, Any] = {
    "mode": "qat", "model_type": "vision", "scope": _UNIVERSAL, "bits": [3, 4, 5, 6, 7, 8],
    "requires_calibration": True, "requires_training": True, "default_epochs": 1,
    "input_format": "onnx", "output_format": "onnx", "action": "qat_pipeline", "description": "...",
}

_METHOD_SPECS = [*_PTQ_METHOD_SPECS, _QAT_SPEC]

def list_quantization_methods(model_type=None, *, scope=None, mode=None) -> list[dict[str, Any]]:
    """Return supported paradigm/method metadata."""
```

```

if model_type is not None and model_type.lower() != ModelType.VISION.value:
    return []
specs = [dict(spec) for spec in _METHOD_SPECS]
if mode is not None:
    specs = [spec for spec in specs if spec.get("mode") == mode]
if scope is None:
    return specs
return [spec for spec in specs if spec.get("scope") == scope]

```

Filter-functie op de statische lijst. `model_type` accepteert vrijwel alleen "vision" (of None) — elke andere waarde geeft een **lege** lijst terug in plaats van een fout, wat een bewuste keuze is voor forward-compatibiliteit (als een klant per ongeluk "llm" opgeeft, krijgt die gewoon "geen methodes" in plaats van een crash).

13. `client.py`

Twee dingen in één bestand: een volwaardige **HTTP-client** voor een gedeployde RunPod-endpoint, en **lokale** helper-functies die dezelfde response-vorm teruggeven zonder een netwerk-call te doen (handig voor snelle validatie vóór je iets daadwerkelijk naar RunPod stuurt, of voor tests).

R3ALQuantClient — de HTTP-laag:

```

class R3ALQuantClient:
    def __init__(self, api_key, endpoint_id, *, base_url=DEFAULT_BASE_URL, timeout_seconds=600) -> None:
        if not api_key.strip(): raise ValueError("api_key is required")
        if not endpoint_id.strip(): raise ValueError("endpoint_id is required")
        self.api_key = api_key.strip()
        self.endpoint_id = endpoint_id.strip()
        self.base_url = base_url.rstrip("/")
        self.timeout_seconds = timeout_seconds

    def _request(self, method, path, body=None) -> dict[str, Any]:
        url = f"{self.base_url}{path}"
        headers = {"Authorization": f"Bearer {self.api_key}", "Content-Type": "application/json"}
        data = json.dumps(body).encode("utf-8") if body is not None else None
        req = urllib.request.Request(url, data=data, headers=headers, method=method)
        try:
            with urllib.request.urlopen(req, timeout=self.timeout_seconds) as resp:
                raw = resp.read().decode("utf-8")
                return json.loads(raw) if raw else {}
        except urllib.error.HTTPError as exc:
            detail = exc.read().decode("utf-8", errors="replace")
            raise R3ALQuantAPIError(f"RunPod API {method} {path} failed ({exc.code}): {detail}",
status_code=exc.code) from exc
        except urllib.error.URLError as exc:
            raise R3ALQuantAPIError(f"RunPod API request failed: {exc.reason}") from exc

```

Gebruikt bewust **alleen de standaardbibliotheek** (`urllib`, geen `requests`-dependency) — dit houdt de SDK licht, zeker relevant omdat `r3al-quant` als kern-package geen netwerk-dependency nodig heeft (alleen de optionele client-functionaliteit).

Response-normalisatie — RunPod's API heeft verschillende response-vormen afhankelijk van sync/async en succes/fout:

```

def _unwrap_job_output(self, response: dict[str, Any]) -> dict[str, Any]:
    """Normalize RunPod sync/async responses to the handler JSON payload."""
    if response.get("status") == "FAILED":
        raise R3ALQuantAPIError("RunPod job failed", response=response)

    output = response.get("output")
    if isinstance(output, dict):
        if output.get("status") == "error":
            raise R3ALQuantAPIError(output.get("error", "Job returned an error"), response=output)
        return output

    if response.get("status") == "error":
        raise R3ALQuantAPIError(response.get("error", "Job returned an error"), response=response)

```

```
return response
```

RunPod's eigen job-status (FAILED op het buitenste niveau, van het platform zelf) wordt onderscheiden van **onze eigen** handler.py-foutstatus ("status": "error", genest in het output-veld voor async jobs, of direct voor sync). Deze functie **vertaalt beide gevallen** naar dezelfde `R3ALQuantAPIError`-exception, zodat de aanroepende code niet zelf met deze twee-lagen-structuur hoeft om te gaan.

invoke() / invoke_async() / wait_for_job():

```
def invoke(self, job_input, *, sync=True) -> dict[str, Any]:
    """Send a raw handler payload (the inner ``input`` object)."""
    job_input = normalize_job_input(job_input)
    path = f"/{self.endpoint_id}/runsync" if sync else f"/{self.endpoint_id}/run"
    response = self._request("POST", path, body={"input": job_input})
    if sync: return self._unwrap_job_output(response)
    return response

def invoke_async(self, job_input) -> dict[str, Any]:
    return self.invoke(job_input, sync=False)

def wait_for_job(self, job_id, *, poll_interval=2.0) -> dict[str, Any]:
    """Poll an async job until completion."""
    deadline = time.time() + self.timeout_seconds
    while time.time() < deadline:
        response = self._request("GET", f"/{self.endpoint_id}/status/{job_id}")
        status = response.get("status")
        if status in {"COMPLETED", "FAILED"}:
            return self._unwrap_job_output(response)
        time.sleep(poll_interval)
    raise R3ALQuantAPIError(f"Timed out waiting for job {job_id} after {self.timeout_seconds}s",
response={"job_id": job_id})
```

Belangrijk: **invoke()** roept zelf **normalize_job_input()** aan — dit betekent dat de client dezelfde normalisatie/validatie lokaal uitvoert **vóórdat** er iets over het netwerk gaat. Dat vangt veel fouten (verkeerde methode-naam, ontbrekend veld) al af zonder een dure round-trip naar RunPod. **wait_for_job()** implementeert simpele polling met een deadline — als de job niet binnen `timeout_seconds` klaar is, wordt een duidelijke timeout-fout gegeven in plaats van voor altijd te blijven wachten.

Actie-specifieke methodes (`quantize`, `benchmark`, `qat_pipeline`, `validate_config`, `list_methods`) — allemaal dunne wrappers rond `_build_job_input() + invoke()/invoke_async()+wait_for_job()`. Ze bestaan puur voor een prettigere Python-API (named parameters in plaats van losse dict-sleutels moeten kennen) — kijk bijvoorbeeld naar `quantize()`:

```
def quantize(self, model, config=None, *, method=None, bits=None, output_dir=None,
calibration_data=None, sync=True, **kwargs) -> dict[str, Any]:
    """Request post-training quantization for an LLM or vision model."""
    fields = {"model": model, **kwargs}
    if method is not None: fields["method"] = method
    if bits is not None: fields["bits"] = bits
    if output_dir is not None: fields["output_dir"] = output_dir
    if calibration_data is not None: fields["calibration_data"] = calibration_data

    job_input = self._build_job_input("quantize", config=config, **fields)
    if sync:
        return self.invoke(job_input, sync=True)
    submitted = self.invoke_async(job_input)
    job_id = submitted.get("id")
    if not job_id:
        raise R3ALQuantAPIError("Async submit did not return a job id", response=submitted)
    return self.wait_for_job(job_id)
```

Lokale helpers (geen netwerk):

```
def validate_config_local(config=None, **kwargs) -> dict[str, Any]:
    """Validate config locally without calling RunPod (same shape as handler response)."""
    payload = {"action": "validate_config", **kwargs}
    if config is not None:
        payload.update(config.model_dump(mode="json") if isinstance(config, QuantConfig) else config)
    try:
        raw_config = normalize_job_input(payload).get("config") or {}
    except UnsupportedConfigError as exc:
```

```

        return {"status": "error", "error": str(exc), "type": type(exc).__name__}
    try:
        cfg = QuantConfig(**raw_config)
    except ValidationError as exc:
        return {"status": "error", "error": "Invalid config", "details": exc.errors()}
    except R3ALQuantError as exc:
        return {"status": "error", "error": str(exc), "type": type(exc).__name__}

    return {"status": "success", "valid": True, "config": cfg.model_dump(mode="json"),
           "requires_calibration": cfg.requires_calibration, "is_dynamic": cfg.is_dynamic, "is_apot":
           cfg.is_apot}

def list_methods_local(model_type=None, *, scope=None) -> dict[str, Any]:
    """Return method catalog locally (same shape as handler response)."""
    return {"status": "success", "methods": list_quantization_methods(model_type, scope=scope)}

```

`validate_config_local` en `list_methods_local` geven **exact dezelfde JSON-vorm** terug als `handler.py`'s `validate_config/list_methods`-acties (zie sectie 14) — dat is bewust zodat je client-side code (en tests) kunt schrijven die identiek werkt of je nu lokaal test of tegen een echte RunPod-endpoint praat.

14. handler.py

De **RunPod serverless entry point** — het enige bestand dat weet hoe een inkomend HTTP/RunPod-event naar de juiste SDK-functie gerouteerd wordt.

Passthrough-kwargs — bewuste "allowlists":

```

_PASSTHROUGH_QUANTIZE_KWARGS = {"max_calib_samples", "num_classes", "input_name", "verbose", "domain",
                                "int8_runtime", "opset"}
_PASSTHROUGH_QAT_PIPELINE_KWARGS = {"calibration_data", "max_calib_samples", "learning_rate", "batch_size",
                                      "device", "seed", "epochs", "verbose", "domain", "opset"}
_PASSTHROUGH_EXPORT_KWARGS = {"source", "input_shape", "opset", "imgsz", "simplify", "dynamic_batch",
                               "input_names", "output_names", "export_opts"}

```

In plaats van **alles** uit de job-payload klakkeloos door te geven aan `Quantizer.quantize(**kwargs)` (risico op onverwachte parameter-conflicten of per ongeluk lekken van interne velden), wordt hier een **expliciete whitelist** per actie bijgehouden. Elke set correspondeert met de `**kwargs`-parameters die de betreffende functie daadwerkelijk accepteert (vergelijk met `Quantizer.quantize()`, `run_onnx_qat_pipeline()`, `export_to_onnx()` — sectie 5, 8, 9).

`_resolve_model_path()` — de auto-export-brug:

```

def _resolve_model_path(job_input: dict[str, Any]) -> str:
    export_opts = job_input.get("export_opts") or {}
    return str(maybe_export_to_onnx(
        job_input["model"], auto_export=bool(job_input.get("auto_export")),
        output_dir=job_input.get("export_output_dir") or job_input.get("output_dir"),
        source=job_input.get("source"), export_opts=export_opts,
        input_shape=job_input.get("input_shape"), opset=job_input.get("opset"),
        imgsz=job_input.get("imgsz"), simplify=job_input.get("simplify"),
        dynamic_batch=job_input.get("dynamic_batch"), input_names=job_input.get("input_names"),
        output_names=job_input.get("output_names"),
    ))

```

Wordt aangeroepen door zowel `_run_quantize` als `_run_qat_pipeline` **vóórdat** de eigenlijke quantisatie/training begint — dit is waar `auto_export` daadwerkelijk werkt: als model geen `.onnx` is, wordt het hier eerst geconverteerd (zie sectie 9.4).

De zeven acties:

```

def _run_export(job_input) -> dict[str, Any]:
    output_dir = job_input.get("output_dir", "/tmp/r3al_export_output")
    extra_kwargs = {k: v for k, v in job_input.items() if k in _PASSTHROUGH_EXPORT_KWARGS}
    result = export_to_onnx(job_input["model"], output_dir=output_dir, **extra_kwargs)
    return result.to_dict()

def _run_list_export_sources(job_input) -> dict[str, Any]:

```

```

    return {"status": "success", "sources": list_export_sources()}

def _run_quantize(job_input) -> dict[str, Any]:
    config = _build_config(job_input.get("config"))
    quantizer = Quantizer(config)
    output_dir = job_input.get("output_dir", "/tmp/r3al_quant_output")
    extra_kwargs = {k: v for k, v in job_input.items() if k in _PASSTHROUGH_QUANTIZE_KWARGS}
    result = quantizer.quantize(model=_resolve_model_path(job_input),
    calibration_data=job_input.get("calibration_data"),
                                output_dir=output_dir, **extra_kwargs)
    return {"status": "success", "quantized": True, "output_dir": str(result.path),
            "config": config.model_dump(mode="json"), **_read_deliverable_metadata(result.path)}

def _run_benchmark(job_input) -> dict[str, Any]:
    config = _build_config(job_input.get("config"))
    quantizer = Quantizer(config)
    bench = quantizer.benchmark(original_model=job_input["original_model"],
    quantized_model=job_input["quantized_model"],
                                eval_data=job_input["eval_data"],
    quality_metric=job_input.get("quality_metric", "perplexity"),
                                warmup_runs=job_input.get("warmup_runs", 3),
    benchmark_runs=job_input.get("benchmark_runs", 10))
    return {"status": "success", "benchmark": bench.model_dump(mode="json")}

def _run_validate_config(job_input) -> dict[str, Any]:
    return validate_config_local(job_input.get("config") or {})

def _run_list_methods(job_input) -> dict[str, Any]:
    return list_methods_local(job_input.get("model_type"), scope=job_input.get("scope"))

def _run_qat_pipeline(job_input) -> dict[str, Any]:
    config = _build_config(job_input.get("config"))
    quantizer = Quantizer(config)
    output_dir = job_input.get("output_dir", "/tmp/r3al_qat_pipeline")
    extra = {k: v for k, v in job_input.items() if k in _PASSTHROUGH_QAT_PIPELINE_KWARGS}
    result = quantizer.train_qat(model=_resolve_model_path(job_input),
    calibration_data=job_input.get("calibration_data"),
                                output_dir=output_dir, **extra)
    payload = result.to_dict()
    payload["config"] = config.model_dump(mode="json")
    return payload

```

Elke `_run_*`-functie volgt hetzelfde patroon: **(1)** bouw een `QuantConfig`, **(2)** filter de relevante `**kwargs` via de whitelist, **(3)** roep de SDK-laag aan (`Quantizer`, `export_to_onnx`, ...), **(4)** formatteer het resultaat naar JSON.

`_read_deliverable_metadata()` — leest het manifest terug om het aan de response toe te voegen:

```

def _read_deliverable_metadata(output_dir) -> dict[str, Any]:
    path = Path(output_dir)
    payload = {"format": "unknown"}
    manifest_path = path / "r3al_quant_manifest.json"
    meta_path = path / "r3al_quant_meta.json"
    if manifest_path.is_file():
        manifest = json.loads(manifest_path.read_text(encoding="utf-8"))
        payload["manifest"] = str(manifest_path)
        payload["output_model"] = manifest.get("output_model")
        payload["format"] = manifest.get("format", "onnx")
        payload["quantized"] = manifest.get("quantized", True)
    elif meta_path.is_file():
        meta = json.loads(meta_path.read_text(encoding="utf-8"))
        payload["format"] = meta.get("format", "pt")
        payload["output_model"] = meta.get("output_model")
        payload["quantized"] = meta.get("quantized", True)
    return payload

```

Voorkeur voor het nieuwe `r3al_quant_manifest.json`, met een **fallback** op `r3al_quant_meta.json` (het "legacy"-bestand — zie `manifest.py:: write_legacy_meta`) voor backwards-compatibiliteit met eerdere SDK-versies die mogelijk alleen het oude bestand schreven.

Action-dispatch-tabel en de hoofdfunctie:

```

_ACTIONS = {"export": _run_export, "quantize": _run_quantize, "benchmark": _run_benchmark,
            "validate_config": _run_validate_config, "list_methods": _run_list_methods,
            "list_export_sources": _run_list_export_sources, "qat_pipeline": _run_qat_pipeline}

def handler(event: dict[str, Any]) -> dict[str, Any]:
    """RunPod entrypoint: dispatch a quantize/benchmark job and return a JSON result."""
    job_input = normalize_job_input(event.get("input", {}))
    action = job_input.get("action", "quantize")

    handler_fn = _ACTIONS.get(action)
    if handler_fn is None:
        return {"status": "error", "error": f"Unknown action '{action}'. Expected one of: {sorted(_ACTIONS)}"}

    try:
        return handler_fn(job_input)
    except ValidationError as exc:
        return {"status": "error", "error": "Invalid config", "details": exc.errors()}
    except R3ALQuantError as exc:
        return {"status": "error", "error": str(exc), "type": type(exc).__name__}
    except KeyError as exc:
        return {"status": "error", "error": f"Missing required input field: {exc}"}
    except Exception as exc:
        return {"status": "error", "error": str(exc), "type": type(exc).__name__, "traceback":
        traceback.format_exc()}
    finally:
        try:
            import torch
            if torch.cuda.is_available():
                torch.cuda.empty_cache()
        except ImportError:
            pass
        gc.collect()

```

Dit is de **enige plek** waarin Python-exceptions **nooit** naar boven mogen lekken als kale traceback — alles wordt netjes naar een JSON-{"status": "error", ...}-object vertaald, met **oplopende specificiteit**:

1. `ValidationError` (Pydantic) -> gestructureerde `details` met alle veld-specifieke fouten.
2. `R3ALQuantError` (onze eigen hiërarchie) -> nette `error/type`.
3. `KeyError` -> meestal een vergeten verplicht veld (`job_input["model"]` bijvoorbeeld) -> duidelijke "missing field"-melding.
4. **Alles anders** -> generieke catch-all, met de **volledige traceback** toegevoegd zodat debugging nog mogelijk is zonder dat de klant een onleesbare Python-stacktrace als enige foutmelding krijgt.

De `finally`-block is een **operationeel** detail specifiek voor RunPod Serverless: workers blijven **warm** tussen jobs (voor snelheid), dus als job A een GPU-model laadt en job B (op dezelfde warme worker) een ander model-grootte gebruikt, kan de GPU-memory van job A een out-of-memory in job B veroorzaken als die niet netjes wordt opgeruimd. `torch.cuda.empty_cache() + gc.collect()` na **elke** job (ook bij succes!) voorkomt dit soort cross-job geheugenlekken.

Bootstrap voor lokaal draaien op RunPod:

```

if __name__ == "__main__":
    if runpod is None:
        raise ImportError("The RunPod serverless handler requires the 'runpod' package. Install it with: pip install r3al-quant[serverless]")
    runpod.serverless.start({"handler": handler})

```

`runpod` wordt **optioneel** geïmporteerd bovenaan het bestand (`try: import runpod except ImportError: runpod = None`) — zo kun je `from handler import handler` gebruiken (bijv. in tests, of `scripts/handler_tests/_run.py`) zonder het `runpod`-package geïnstalleerd te hebben. Alleen als je het bestand **direct** uitvoert (`python handler.py`, zoals RunPod's productie-container doet) is `runpod` echt vereist, en start het de serverless event-loop die `handler()` per inkomende job aanroept.

15. Volledige request-flows

15.1 quantize met weight_quant (het meest voorkomende pad)

```
1. RunPod stuurt: {"input": {"action": "quantize", "model": "model.onnx"}}
2. handler.handler(event)
   -> normalize_job_input() [job_input.py]
       - method niet opgegeven + .onnx bestand -> method = "weight_quant"
       - mode = "ptq", qat_scheme = "apot", qat_wbit = 8 (defaults)
       - output_dir = "/tmp/r3al_quant_output" (of /runpod-volume/quantized)
   -> _run_quantize(job_input) [handler.py]
       -> QuantConfig(**config) [config.py] – valideert, zet bits = qat_wbit
       -> Quantizer(config) [quantizer.py]
           -> get_backend(config) [backends/__init__.py]
               -> WeightQuantONNXBackend(config) [onnx_weight_quant.py]
       -> _resolve_model_path(job_input)
           -> maybe_export_to_onnx(...) [export/__init__.py]
               - al .onnx -> geen conversie, retourneer pad ongewijzigd
       -> quantizer.quantize(model, output_dir=...)
           -> backend.quantize(...) [onnx_weight_quant.py]
               - onnx.load() -> loop over initializers
               - _should_quantize_initializer() filtert BatchNorm-stats/1D
               - _quantize_weight_array() -> weight_quantize_fn (APoT, algorithms/qat/layers.py)
                   of apot_quantize_tensor (algorithms/apot.py) afhankelijk van scheme
               - onnx.save() -> *.quantized.onnx
               - stamp_onnx_metadata() [manifest.py] -> r3al_* velden in ONNX zelf
               - build_quant_manifest() + write_quant_manifest() [manifest.py]
                   -> r3al_quant_manifest.json ernaast
       -> _read_deliverable_metadata() [handler.py] – leest manifest terug
3. Return: {"status": "success", "quantized": true, "output_dir": ..., "output_model": ..., "manifest": ...}
```

15.2 qat_pipeline

```
1. RunPod stuurt: {"input": {"action": "qat_pipeline", "model": "model.onnx",
                             "calibration_data": ["img1.jpg", "img2.jpg"]}}
2. handler.handler(event)
   -> normalize_job_input()
       - mode = "qat" (geen "method"-veld – QAT gebruikt geen PTQ-methode)
       - qat_scheme = "apot", qat_epochs = 1 (defaults)
   -> _validate_onnx_input() – checkt: .onnx + calibration_data aanwezig
   -> _run_qat_pipeline(job_input)
       -> QuantConfig(mode="qat", ...)
       -> Quantizer(config) – géén backend gebouwd (mode != PTQ)
       -> quantizer.train_qat(model, calibration_data, output_dir)
           -> run_onnx_qat_pipeline() [pipeline/onnx_qat.py]
               1. onnx_input_spec() -> input-naam + shape uit graph
               2. load_onnx_as_torch() -> onnx2torch.convert() [onnx_loader.py]
               3. teacher = deepcopy(float_model), requires_grad=False
               4. quantize_onnx_model() -> Conv2d -> QuantConv2d [algorithms/qat/onnx_model.py]
               5. _iter_training_batches() -> afbeeldingen laden/batchen
               6. calibrate_activations() -> act_alpha initialiseren [algorithms/qat/calibration.py]
               7. quant_alpha_parameters() -> alleen wgt_alpha/act_alpha trainbaar
               8. training loop: student vs teacher MSE-loss, Adam-optimizer
               9. fold_qat_model_for_export() -> QuantConv2d -> kale Conv2d [onnx_model.py]
               10. torch.onnx.export() -> *_qat.onnx
               11. finalize_onnx_deliverable() -> *.quantized.onnx + manifest [onnx_export.py]
3. Return: {"status": "success", "quantized": true, "output_model": ..., "steps": [...], "elapsed_seconds": ...}
```

15.3 quantize met auto_export vanaf een .pt-bestand

```
1. Input: {"action": "quantize", "model": "yolov8n.pt", "auto_export": true, "source": "ultralytics"}
2. normalize_job_input()
   - model eindigt niet op .onnx, maar auto_export=true -> method = "weight_quant" alvast gezet
3. _run_quantize()
   -> _resolve_model_path(job_input)
       -> maybe_export_to_onnx(model="yolov8n.pt", auto_export=True, source="ultralytics")
```

```

-> export_to_onnx() [export/__init__.py]
  -> resolve_adapter(path, source="ultralytics") [export/registry.py]
    -> UltralyticsAdapter (matched op naam + .pt extensie)
  -> adapter.to_onnx() [export/adapters/ultralytics.py]
    -> YOLO(path).export(format="onnx", ...)
    -> validate_onnx_model() [export/base.py]
    -> build_export_manifest() + write_export_manifest()
      -> r3a1_export_manifest.json (LET OP: andere naam dan quant-manifest!)
  -> retourneert het pad naar het verse .onnx-bestand
-> quantizer.quantize(dat .onnx-pad, ...) [verder identiek aan 15.1]

```

Let op: het export-manifest (`r3a1_export_manifest.json`) en het quantisatie-manifest (`r3a1_quant_manifest.json`) zijn **twee verschillende bestanden** met een andere structuur (`build_export_manifest` in `export/base.py` vs. `build_quant_manifest` in `manifest.py`) — de export- stap documenteert de *conversie*, de quantize-stap documenteert de *quantisatie*.

16. Tests

De `tests/-map` spiegelt de `src/r3a1_quant/-`structuur, en test elk onderdeel in isolatie zonder een echte RunPod-omgeving nodig te hebben.

Testbestand	Test-doel
<code>test_api.py</code>	<code>handler()</code> -dispatch, <code>normalize_job_input()</code> , methode-catalogus
<code>test_config.py</code>	QuantConfig-validatie en berekende properties
<code>test_apot.py</code>	<code>algorithms/apot.py</code> — grid-constructie, quantize/dequantize
<code>test_weight_quant.py</code>	<code>WeightQuantONNXBackend</code> end-to-end op een klein ONNX-model
<code>test_onnx_utils.py</code>	<code>primary_input_name()</code>
<code>test_onnx_manifest.py</code>	<code>manifest.py</code> — schrijven/teruglezen
<code>test_onnx_qat.py</code>	<code>algorithms/qat/*</code> — QuantConv2d, calibratie
<code>test_qat.py</code> / <code>test_qat_pipeline.py</code>	volledige QAT-trainingsloop op een mini-model
<code>test_export.py</code>	export-adapters (<code>resolve_adapter</code> , elke adapter met een klein testbestand)

`scripts/handler_tests/` bevat daarnaast **handmatige smoke-tests** die echte modellen (YOLOv8, custom face-pose-checkpoints) downloaden en door de volledige pipeline halen — nuttig voor "werkt dit ook echt met een groot, realistisch model" naast de snelle, geïsoleerde pytest-suite.

Draaien:

```
pip install -e "[vision,yolo,dev]"
pytest
pytest tests/test_face_pose.py -m integration -v # langzame, netwerk-afhankelijke tests
```

17. Cheat sheet

Waar moet ik zijn als ik...

...wil weten hoe	Kijk in
een nieuw PTQ-methode toevoegen	<code>backends/vision/</code> , registreer in <code>backends/__init__.py::_VISION_BACKENDS</code> , voeg toe aan <code>types.py::VisionMethod</code> en <code>methods.py</code>
APoT-wiskunde werkt	<code>algorithms/apot.py</code>
QAT daadwerkelijk traint	<code>pipeline/onnx_qat.py</code> (orkestratie) + <code>algorithms/qat/layers.py</code> (STE-lagen)
een config-veld gevalideerd wordt	<code>config.py::QuantConfig._validate_vision()</code>
een nieuw export-formaat toevoegen	maak een klasse met <code>can_handle()/to_onnx()</code> in <code>export/adapters/</code> , registreer in <code>export/registry.py::_ADAPTERS</code>
het manifest/bewijs-bestand gebouwd wordt	<code>manifest.py::build_quant_manifest()</code>
een RunPod-actie werkt	<code>handler.py::_ACTIONS</code> + de bijbehorende <code>_run_*()</code> -functie
JSON-payloads genormaliseerd worden	<code>job_input.py::normalize_job_input()</code>
benchmarks gemeten worden	<code>benchmark.py::Benchmark._measure_latency()</code>

Belangrijkste invarianten om te onthouden:

1. **Alles is ONNX in, ONNX uit.** Nooit een ander eindformaat.
2. **QuantConfig is de single source of truth** voor alle instellingen — elke actie bouwt er eerst één van en valideert daarmee vroeg.
3. **Elke deliverable krijgt een manifest** (`r3al_quant_manifest.json`) én interne ONNX-metadata (`stamp_onnx_metadata`) — twee onafhankelijke bewijslagen.
4. **PTQ = geen training** (backends werken direct op ONNX-initializers of via ONNX Runtime's quantization-tools). **QAT = wel training**, maar traint alleen lichte `alpha`-clipdrempels, niet de volledige gewichten (zelf-distillatie tegen een bevroren teacher-kopie).
5. **handler.py laat nooit een kale exception ontsnappen** — alles wordt `{"status": "error", ...}` JSON, met oplopende specificiteit per exception-type.

