```
          "operation": "multiply",
          "number_two": 42
        }
      }
    ]
    res = 0
    if json_data != expected:
        print("  Result is not as expected!")
        print("  This can happen on highly quantized models")
        res = 1
    tools_map = {tool.__name__:tool for tool in tools}
    for call in json_data:
      tool = tools_map.get(call["function"])
      if not tool:
          print(f"Error: unknown tool {call['function']}")
          return 1
      result = tool(**call["params"]).run()
      print(f"  Call {call['function']} returned {result}")
    # Should output something like this:
    #   Call get_current_datetime returned 2024-07-15 09:50:38
    #   Call get_current_weather returned {"location": "London", "temperature": "42", "unit": "celsius"}
    #   Call Calculator returned 1764
    return res


def main():
    parser = argparse.ArgumentParser(description=sys.modules[__name__].__doc__)
    parser.add_argument("--host", default="localhost:8080", help="llama.cpp server")
    parser.add_argument("-v", "--verbose", action="store_true", help="enables logging")
    args = parser.parse_args()
    logging.basicConfig(level=logging.INFO if args.verbose else logging.ERROR)
    ret = 0
    # Comment out below to only run the example you want.
    ret = ret or example_rce(args.host)
    ret = ret or example_calculator(args.host)
    ret = ret or example_struct(args.host)
    ret = ret or example_concurrent(args.host)
    return ret


if __name__ == "__main__":
    sys.exit(main())

==== quant_feeder_setup.py ====
# quant_feeder_setup.py
# Fully automated setup for quant_prompt_feeder

import subprocess
import os
from pathlib import Path
import sys
import time
import urllib.request
import zipfile
```

```
LLAMA_REPO = "https://github.com/ggerganov/llama.cpp.git"
MODEL_URL                                                                                                    =
"https://huggingface.co/afrideva/Tinystories-gpt-0.1-3m-GGUF/resolve/main/TinyStories-GPT-0.1-3M.Q2_K.gguf"

MODEL_DIR = Path("models")
MODEL_FILE = MODEL_DIR / "TinyStories.Q2_K.gguf"
LLAMA_DIR = Path("llama.cpp")
LLAMA_BIN = LLAMA_DIR / "build/bin/main"

def install_dependencies():
    print("[setup] CONFIG Installing dependencies...")
    subprocess.run([sys.executable, "-m", "pip", "install", "--quiet", "--upgrade", "pip"])
    subprocess.run([sys.executable, "-m", "pip", "install", "--quiet", "requests"])

def clone_llama_cpp():
    if not LLAMA_DIR.exists():
        print("[setup] INFO Cloning llama.cpp...")
        subprocess.run(["git", "clone", LLAMA_REPO])
    else:
        print("[setup] [OK] llama.cpp already exists")

def build_llama_cpp():
    print("[setup] ? Building llama.cpp...")
    os.makedirs(LLAMA_DIR / "build", exist_ok=True)
    subprocess.run(["cmake", "-B", "build"], cwd=LLAMA_DIR)
    subprocess.run(["cmake", "--build", "build", "--config", "Release"], cwd=LLAMA_DIR)

def download_model():
    if MODEL_FILE.exists():
        print(f"[setup] [OK] Model already downloaded: {MODEL_FILE.name}")
        return
    print(f"[setup] ??  Downloading model to {MODEL_FILE}...")
    MODEL_DIR.mkdir(parents=True, exist_ok=True)
    urllib.request.urlretrieve(MODEL_URL, MODEL_FILE)

def patch_feeder():
    print("[setup] ?? Patching quant_prompt_feeder.py with model and llama path")
    feeder_code = Path("quant_prompt_feeder.py").read_text(encoding="utf-8")
    patched = feeder_code.replace(
        'MODEL_PATH = Path("models/TinyLlama.Q4_0.gguf")',
        f'MODEL_PATH = Path("{MODEL_FILE.as_posix()}")'
    ).replace(
        'LLAMA_CPP_PATH = Path("llama.cpp/build/bin/main")',
        f'LLAMA_CPP_PATH = Path("{LLAMA_BIN.as_posix()}")'
    )
    Path("quant_prompt_feeder.py").write_text(patched, encoding="utf-8")

def run_feeder():
    print("[setup] LAUNCH Running quant_prompt_feeder.py...")
    subprocess.run(["python", "quant_prompt_feeder.py"])

if __name__ == "__main__":
    install_dependencies()
```

```
    clone_llama_cpp()
    build_llama_cpp()
    download_model()
    patch_feeder()
    run_feeder()

==== quant_prompt_feeder.py ====
# quant_prompt_feeder.py
# Uses a local .gguf model to generate beliefs and feed the LogicShredder

import subprocess
import time
from pathlib import Path

MODEL_PATH = Path("models/TinyLlama.Q4_0.gguf")  # CHANGE if different
LLAMA_CPP_PATH = Path("llama.cpp/build/bin/main")  # Point to llama.cpp binary
PROMPT = "List 25 fundamental beliefs about the universe, logic, and consciousness."
OUTPUT_FILE = Path("logic_input/generated_beliefs.txt")

def generate_beliefs():
    print(f"[feeder] INFO Generating beliefs using {MODEL_PATH.name}")
    with open("prompt.txt", "w", encoding="utf-8") as f:
        f.write(PROMPT)

    try:
        result = subprocess.run([
            str(LLAMA_CPP_PATH),
            "-m", str(MODEL_PATH),
            "-p", PROMPT,
            "--n-predict", "300",
            "--top-k", "40",
            "--top-p", "0.9",
            "--temp", "0.7"
        ], capture_output=True, text=True, timeout=120)

        if result.returncode == 0:
            text = result.stdout
            with open(OUTPUT_FILE, "w", encoding="utf-8") as out:
                out.write(text)
            print(f"[feeder] [OK] Beliefs saved to {OUTPUT_FILE}")
        else:
            print(f"[feeder] ERROR Model failed to respond properly.")

    except Exception as e:
        print(f"[feeder] ERROR Model execution failed: {e}")

def feed_beliefs():
    print("[feeder] ? Feeding into LogicShredder...")
    subprocess.run(["python", "total_devourer.py"])
    subprocess.run(["python", "run_logicshredder.py"])

if __name__ == "__main__":
    generate_beliefs()
    time.sleep(1)
```

```
    feed_beliefs()

==== quants.py ====
from __future__ import annotations
from abc import ABC, abstractmethod
from typing import Any, Callable, Sequence
from math import log2, ceil

from numpy.typing import DTypeLike

from .constants import GGML_QUANT_SIZES, GGMLQuantizationType, QK_K
from .lazy import LazyNumpyTensor

import numpy as np


def quant_shape_to_byte_shape(shape: Sequence[int], quant_type: GGMLQuantizationType) -> tuple[int, ...]:
    block_size, type_size = GGML_QUANT_SIZES[quant_type]
    if shape[-1] % block_size != 0:
        raise ValueError(f"Quantized tensor row size ({shape[-1]}) is not a multiple of {quant_type.name} block
size ({block_size})")
    return (*shape[:-1], shape[-1] // block_size * type_size)


def quant_shape_from_byte_shape(shape: Sequence[int], quant_type: GGMLQuantizationType) -> tuple[int, ...]:
    block_size, type_size = GGML_QUANT_SIZES[quant_type]
    if shape[-1] % type_size != 0:
        raise ValueError(f"Quantized tensor bytes per row ({shape[-1]}) is not a multiple of {quant_type.name}
type size ({type_size})")
    return (*shape[:-1], shape[-1] // type_size * block_size)


# This is faster than np.vectorize and np.apply_along_axis because it works on more than one row at a time
def _apply_over_grouped_rows(func: Callable[[np.ndarray], np.ndarray], arr: np.ndarray, otype: DTypeLike,
oshape: tuple[int, ...]) -> np.ndarray:
    rows = arr.reshape((-1, arr.shape[-1]))
    osize = 1
    for dim in oshape:
        osize *= dim
    out = np.empty(shape=osize, dtype=otype)
    # compute over groups of 16 rows (arbitrary, but seems good for performance)
    n_groups = (rows.shape[0] // 16) or 1
    np.concatenate([func(group).ravel() for group in np.array_split(rows, n_groups)], axis=0, out=out)
    return out.reshape(oshape)


# round away from zero
# ref: https://stackoverflow.com/a/59143326/22827863
def np_roundf(n: np.ndarray) -> np.ndarray:
    a = abs(n)
    floored = np.floor(a)
    b = floored + np.floor(2 * (a - floored))
    return np.sign(n) * b
```

```
class QuantError(Exception): ...


_type_traits: dict[GGMLQuantizationType, type[__Quant]] = {}


def quantize(data: np.ndarray, qtype: GGMLQuantizationType) -> np.ndarray:
    if qtype == GGMLQuantizationType.F32:
        return data.astype(np.float32, copy=False)
    elif qtype == GGMLQuantizationType.F16:
        return data.astype(np.float16, copy=False)
    elif (q := _type_traits.get(qtype)) is not None:
        return q.quantize(data)
    else:
        raise NotImplementedError(f"Quantization for {qtype.name} is not yet implemented")


def dequantize(data: np.ndarray, qtype: GGMLQuantizationType) -> np.ndarray:
    if qtype == GGMLQuantizationType.F32:
        return data.view(np.float32)
    elif qtype == GGMLQuantizationType.F16:
        return data.view(np.float16).astype(np.float32)
    elif (q := _type_traits.get(qtype)) is not None:
        return q.dequantize(data)
    else:
        raise NotImplementedError(f"Dequantization for {qtype.name} is not yet implemented")


class __Quant(ABC):
    qtype: GGMLQuantizationType
    block_size: int
    type_size: int

    grid: np.ndarray[Any, np.dtype[np.float32]] | None = None
    grid_shape: tuple[int, int] = (0, 0)
    grid_map: tuple[int | float, ...] = ()
    grid_hex: bytes | None = None

    def __init__(self):
        return TypeError("Quant conversion classes can't have instances")

    def __init_subclass__(cls, qtype: GGMLQuantizationType) -> None:
        cls.qtype = qtype
        cls.block_size, cls.type_size = GGML_QUANT_SIZES[qtype]
        cls.__quantize_lazy = LazyNumpyTensor._wrap_fn(
            cls.__quantize_array,
            meta_noop=(np.uint8, cls.__shape_to_bytes)
        )
        cls.__dequantize_lazy = LazyNumpyTensor._wrap_fn(
            cls.__dequantize_array,
            meta_noop=(np.float32, cls.__shape_from_bytes)
        )
        assert qtype not in _type_traits
```

```
        _type_traits[qtype] = cls

    @classmethod
    def init_grid(cls):
        if cls.grid is not None or cls.grid_hex is None:
            return

        bits_per_elem = ceil(log2(len(cls.grid_map)))
        assert bits_per_elem != 0, cls.qtype.name
        elems_per_byte = 8 // bits_per_elem

        grid = np.frombuffer(cls.grid_hex, dtype=np.uint8)
        # decode hexadecimal chars from grid
        grid = grid.reshape((-1, 2))
         grid = (np.where(grid > 0x40, grid + 9, grid) & 0x0F) << np.array([4, 0], dtype=np.uint8).reshape((1,
2))
        grid = grid[..., 0] | grid[..., 1]
        # unpack the grid values
              grid = grid.reshape((-1, 1)) >> np.array([i for i in range(0, 8, 8 // elems_per_byte)],
dtype=np.uint8).reshape((1, elems_per_byte))
        grid = (grid & ((1 << bits_per_elem) - 1)).reshape((-1, 1))
        grid_map = np.array(cls.grid_map, dtype=np.float32).reshape((1, -1))
        grid = np.take_along_axis(grid_map, grid, axis=-1)
        cls.grid = grid.reshape((1, 1, *cls.grid_shape))

    @classmethod
    @abstractmethod
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        raise NotImplementedError

    @classmethod
    @abstractmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        raise NotImplementedError

    @classmethod
    def quantize_rows(cls, rows: np.ndarray) -> np.ndarray:
        rows = rows.astype(np.float32, copy=False)
        shape = rows.shape
        n_blocks = rows.size // cls.block_size
        blocks = rows.reshape((n_blocks, cls.block_size))
        blocks = cls.quantize_blocks(blocks)
        assert blocks.dtype == np.uint8
        assert blocks.shape[-1] == cls.type_size
        return blocks.reshape(cls.__shape_to_bytes(shape))

    @classmethod
    def dequantize_rows(cls, rows: np.ndarray) -> np.ndarray:
        rows = rows.view(np.uint8)
        shape = rows.shape
        n_blocks = rows.size // cls.type_size
        blocks = rows.reshape((n_blocks, cls.type_size))
        blocks = cls.dequantize_blocks(blocks)
        assert blocks.dtype == np.float32
```

```
        assert blocks.shape[-1] == cls.block_size
        return blocks.reshape(cls.__shape_from_bytes(shape))

    @classmethod
    def __shape_to_bytes(cls, shape: Sequence[int]):
        return quant_shape_to_byte_shape(shape, cls.qtype)

    @classmethod
    def __shape_from_bytes(cls, shape: Sequence[int]):
        return quant_shape_from_byte_shape(shape, cls.qtype)

    @classmethod
    def __quantize_array(cls, array: np.ndarray) -> np.ndarray:
                            return    _apply_over_grouped_rows(cls.quantize_rows,    arr=array,    otype=np.uint8,
oshape=cls.__shape_to_bytes(array.shape))

    @classmethod
    def __dequantize_array(cls, array: np.ndarray) -> np.ndarray:
        cls.init_grid()
                        return   _apply_over_grouped_rows(cls.dequantize_rows,   arr=array,   otype=np.float32,
oshape=cls.__shape_from_bytes(array.shape))

    @classmethod
    def __quantize_lazy(cls, lazy_tensor: LazyNumpyTensor, /) -> Any:
        pass

    @classmethod
    def __dequantize_lazy(cls, lazy_tensor: LazyNumpyTensor, /) -> Any:
        pass

    @classmethod
    def can_quantize(cls, tensor: np.ndarray | LazyNumpyTensor) -> bool:
        return tensor.shape[-1] % cls.block_size == 0

    @classmethod
    def quantize(cls, tensor: np.ndarray | LazyNumpyTensor) -> np.ndarray:
        if not cls.can_quantize(tensor):
            raise QuantError(f"Can't quantize tensor with shape {tensor.shape} to {cls.qtype.name}")
        if isinstance(tensor, LazyNumpyTensor):
            return cls.__quantize_lazy(tensor)
        else:
            return cls.__quantize_array(tensor)

    @classmethod
    def dequantize(cls, tensor: np.ndarray | LazyNumpyTensor) -> np.ndarray:
        if isinstance(tensor, LazyNumpyTensor):
            return cls.__dequantize_lazy(tensor)
        else:
            return cls.__dequantize_array(tensor)


class BF16(__Quant, qtype=GGMLQuantizationType.BF16):
    @classmethod
    # same as ggml_compute_fp32_to_bf16 in ggml-impl.h
```

```
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n = blocks.view(np.uint32)
        # force nan to quiet
        n = np.where((n & 0x7fffffff) > 0x7f800000, (n & np.uint32(0xffff0000)) | np.uint32(64 << 16), n)
        # round to nearest even
        n = (np.uint64(n) + (0x7fff + ((n >> 16) & 1))) >> 16
        return n.astype(np.uint16).view(np.uint8)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        return (blocks.view(np.int16).astype(np.int32) << 16).view(np.float32)


class Q4_0(__Quant, qtype=GGMLQuantizationType.Q4_0):
    @classmethod
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        imax = abs(blocks).argmax(axis=-1, keepdims=True)
        max = np.take_along_axis(blocks, imax, axis=-1)

        d = max / -8
        with np.errstate(divide="ignore"):
            id = np.where(d == 0, 0, 1 / d)
        # FIXME: Q4_0's reference rounding is cursed and depends on FMA
                            qs    =    np.trunc((np.float64(blocks)    *    np.float64(id))    +    np.float64(8.5),
dtype=np.float32).astype(np.uint8).clip(0, 15)

        qs = qs.reshape((n_blocks, 2, cls.block_size // 2))
        qs = qs[..., 0, :] | (qs[..., 1, :] << np.uint8(4))

        d = d.astype(np.float16).view(np.uint8)

        return np.concatenate([d, qs], axis=-1)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, qs = np.hsplit(blocks, [2])

        d = d.view(np.float16).astype(np.float32)

        qs = qs.reshape((n_blocks, -1, 1, cls.block_size // 2)) >> np.array([0, 4], dtype=np.uint8).reshape((1,
1, 2, 1))
        qs = (qs & np.uint8(0x0F)).reshape((n_blocks, -1)).astype(np.int8) - np.int8(8)

        return (d * qs.astype(np.float32))


class Q4_1(__Quant, qtype=GGMLQuantizationType.Q4_1):
    @classmethod
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]
```

```
        max = blocks.max(axis=-1, keepdims=True)
        min = blocks.min(axis=-1, keepdims=True)

        d = (max - min) / 15
        with np.errstate(divide="ignore"):
            id = np.where(d == 0, 0, 1 / d)
        qs = np.trunc((blocks - min) * id + np.float32(0.5), dtype=np.float32).astype(np.uint8).clip(0, 15)

        qs = qs.reshape((n_blocks, 2, cls.block_size // 2))
        qs = qs[..., 0, :] | (qs[..., 1, :] << np.uint8(4))

        d = d.astype(np.float16).view(np.uint8)
        m = min.astype(np.float16).view(np.uint8)

        return np.concatenate([d, m, qs], axis=-1)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        m, qs = np.hsplit(rest, [2])

        d = d.view(np.float16).astype(np.float32)
        m = m.view(np.float16).astype(np.float32)

        qs = qs.reshape((n_blocks, -1, 1, cls.block_size // 2)) >> np.array([0, 4], dtype=np.uint8).reshape((1,
1, 2, 1))
        qs = (qs & np.uint8(0x0F)).reshape((n_blocks, -1)).astype(np.float32)

        return (d * qs) + m


class Q5_0(__Quant, qtype=GGMLQuantizationType.Q5_0):
    @classmethod
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        imax = abs(blocks).argmax(axis=-1, keepdims=True)
        max = np.take_along_axis(blocks, imax, axis=-1)

        d = max / -16
        with np.errstate(divide="ignore"):
            id = np.where(d == 0, 0, 1 / d)
        # FIXME: Q5_0's reference rounding is cursed and depends on FMA
                            q    =    np.trunc((np.float64(blocks)    *    np.float64(id))    +    np.float64(16.5),
dtype=np.float32).astype(np.uint8).clip(0, 31)

        qs = q.reshape((n_blocks, 2, cls.block_size // 2))
        qs = (qs[..., 0, :] & np.uint8(0x0F)) | (qs[..., 1, :] << np.uint8(4))

                            qh    =    np.packbits(q.reshape((n_blocks,    1,    32))    >>    np.uint8(4),    axis=-1,
bitorder="little").reshape(n_blocks, 4)
```

```
        d = d.astype(np.float16).view(np.uint8)

        return np.concatenate([d, qh, qs], axis=-1)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        qh, qs = np.hsplit(rest, [4])

        d = d.view(np.float16).astype(np.float32)
        qh = qh.view(np.uint32)

        qh = qh.reshape((n_blocks, 1)) >> np.array([i for i in range(32)], dtype=np.uint32).reshape((1, 32))
        ql = qs.reshape((n_blocks, -1, 1, cls.block_size // 2)) >> np.array([0, 4], dtype=np.uint8).reshape((1,
1, 2, 1))
        qh = (qh & np.uint32(0x01)).astype(np.uint8)
        ql = (ql & np.uint8(0x0F)).reshape((n_blocks, -1))

        qs = (ql | (qh << np.uint8(4))).astype(np.int8) - np.int8(16)

        return (d * qs.astype(np.float32))


class Q5_1(__Quant, qtype=GGMLQuantizationType.Q5_1):
    @classmethod
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        max = blocks.max(axis=-1, keepdims=True)
        min = blocks.min(axis=-1, keepdims=True)

        d = (max - min) / 31
        with np.errstate(divide="ignore"):
            id = np.where(d == 0, 0, 1 / d)
        q = np.trunc((blocks - min) * id + np.float32(0.5), dtype=np.float32).astype(np.uint8).clip(0, 31)

        qs = q.reshape((n_blocks, 2, cls.block_size // 2))
        qs = (qs[..., 0, :] & np.uint8(0x0F)) | (qs[..., 1, :] << np.uint8(4))

                        qh   =   np.packbits(q.reshape((n_blocks,   1,   32))   >>   np.uint8(4),   axis=-1,
bitorder="little").reshape(n_blocks, 4)

        d = d.astype(np.float16).view(np.uint8)
        m = min.astype(np.float16).view(np.uint8)

        return np.concatenate([d, m, qh, qs], axis=-1)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]
```

```
        d, rest = np.hsplit(blocks, [2])
        m, rest = np.hsplit(rest, [2])
        qh, qs = np.hsplit(rest, [4])

        d = d.view(np.float16).astype(np.float32)
        m = m.view(np.float16).astype(np.float32)
        qh = qh.view(np.uint32)

        qh = qh.reshape((n_blocks, 1)) >> np.array([i for i in range(32)], dtype=np.uint32).reshape((1, 32))
        ql = qs.reshape((n_blocks, -1, 1, cls.block_size // 2)) >> np.array([0, 4], dtype=np.uint8).reshape((1,
1, 2, 1))
        qh = (qh & np.uint32(0x01)).astype(np.uint8)
        ql = (ql & np.uint8(0x0F)).reshape((n_blocks, -1))

        qs = (ql | (qh << np.uint8(4))).astype(np.float32)

        return (d * qs) + m


class Q8_0(__Quant, qtype=GGMLQuantizationType.Q8_0):
    @classmethod
    # Implementation of Q8_0 with bit-exact same results as reference implementation in ggml-quants.c
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:

        d = abs(blocks).max(axis=1, keepdims=True) / 127
        with np.errstate(divide="ignore"):
            id = np.where(d == 0, 0, 1 / d)
        qs = np_roundf(blocks * id)

        # (n_blocks, 2)
        d = d.astype(np.float16).view(np.uint8)
        # (n_blocks, block_size)
        qs = qs.astype(np.int8).view(np.uint8)

        return np.concatenate([d, qs], axis=1)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        d, x = np.split(blocks, [2], axis=1)
        d = d.view(np.float16).astype(np.float32)
        x = x.view(np.int8).astype(np.float32)

        return (x * d)


class Q2_K(__Quant, qtype=GGMLQuantizationType.Q2_K):
    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        scales, rest = np.hsplit(blocks, [QK_K // 16])
        qs, rest = np.hsplit(rest, [QK_K // 4])
        d, dmin = np.hsplit(rest, [2])
```

```
        d = d.view(np.float16).astype(np.float32)
        dmin = dmin.view(np.float16).astype(np.float32)

        # (n_blocks, 16, 1)
        dl = (d * (scales & 0xF).astype(np.float32)).reshape((n_blocks, QK_K // 16, 1))
        ml = (dmin * (scales >> 4).astype(np.float32)).reshape((n_blocks, QK_K // 16, 1))

        shift = np.array([0, 2, 4, 6], dtype=np.uint8).reshape((1, 1, 4, 1))

        qs = (qs.reshape((n_blocks, -1, 1, 32)) >> shift) & np.uint8(3)

        qs = qs.reshape((n_blocks, QK_K // 16, 16)).astype(np.float32)

        qs = dl * qs - ml

        return qs.reshape((n_blocks, -1))


class Q3_K(__Quant, qtype=GGMLQuantizationType.Q3_K):
    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        hmask, rest = np.hsplit(blocks, [QK_K // 8])
        qs, rest = np.hsplit(rest, [QK_K // 4])
        scales, d = np.hsplit(rest, [12])

        d = d.view(np.float16).astype(np.float32)

        # The scales are packed at 6-bit each in this pattern:
        #  0: IIIIAAAA
        #  1: JJJJBBBB
        #  2: KKKKCCCC
        #  3: LLLLDDDD
        #  4: MMMMEEEE
        #  5: NNNNFFFF
        #  6: OOOOGGGG
        #  7: PPPPHHHH
        #  8: MMIIEEAA
        #  9: NNJJFFBB
        # 10: OOKKGGCC
        # 11: PPLLHHDD
        lscales, hscales = np.hsplit(scales, [8])
        lscales = lscales.reshape((n_blocks, 1, 8)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 2, 1))
        lscales = lscales.reshape((n_blocks, 16))
         hscales = hscales.reshape((n_blocks, 1, 4)) >> np.array([0, 2, 4, 6], dtype=np.uint8).reshape((1, 4,
1))
        hscales = hscales.reshape((n_blocks, 16))
        scales = (lscales & np.uint8(0x0F)) | ((hscales & np.uint8(0x03)) << np.uint8(4))
        scales = (scales.astype(np.int8) - np.int8(32)).astype(np.float32)

        dl = (d * scales).reshape((n_blocks, 16, 1))

        ql = qs.reshape((n_blocks, -1, 1, 32)) >> np.array([0, 2, 4, 6], dtype=np.uint8).reshape((1, 1, 4, 1))
```

```
        qh = hmask.reshape(n_blocks, -1, 1, 32) >> np.array([i for i in range(8)], dtype=np.uint8).reshape((1,
1, 8, 1))
        ql = ql.reshape((n_blocks, 16, QK_K // 16)) & np.uint8(3)
        qh = (qh.reshape((n_blocks, 16, QK_K // 16)) & np.uint8(1))
        qh = qh ^ np.uint8(1)  # strangely, the offset is zero when the bitmask is 1
        q = (ql.astype(np.int8) - (qh << np.uint8(2)).astype(np.int8)).astype(np.float32)

        return (dl * q).reshape((n_blocks, QK_K))


class Q4_K(__Quant, qtype=GGMLQuantizationType.Q4_K):
    K_SCALE_SIZE = 12

    @staticmethod
    def get_scale_min(scales: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
        n_blocks = scales.shape[0]
        scales = scales.view(np.uint8)
        ### Unpacking the following: ###
        #  0 EEAAAAAA
        #  1 FFBBBBBB
        #  2 GGCCCCCC
        #  3 HHDDDDDD
        #  4 eeaaaaaa
        #  5 ffbbbbbb
        #  6 ggcccccc
        #  7 hhdddddd
        #  8 eeeeEEEE
        #  9 ffffFFFF
        # 10 ggggGGGG
        # 11 hhhhHHHH
        scales = scales.reshape((n_blocks, 3, 4))
        d, m, m_d = np.split(scales, 3, axis=-2)

        sc = np.concatenate([d & 0x3F, (m_d & 0x0F) | ((d >> 2) & 0x30)], axis=-1)
        min = np.concatenate([m & 0x3F, (m_d >> 4) | ((m >> 2) & 0x30)], axis=-1)

        return (sc.reshape((n_blocks, 8)), min.reshape((n_blocks, 8)))

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        dmin, rest = np.hsplit(rest, [2])
        scales, qs = np.hsplit(rest, [cls.K_SCALE_SIZE])

        d = d.view(np.float16).astype(np.float32)
        dmin = dmin.view(np.float16).astype(np.float32)

        sc, m = Q4_K.get_scale_min(scales)

        d = (d * sc.astype(np.float32)).reshape((n_blocks, -1, 1))
        dm = (dmin * m.astype(np.float32)).reshape((n_blocks, -1, 1))
```

```
        qs = qs.reshape((n_blocks, -1, 1, 32)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2, 1))
        qs = (qs & np.uint8(0x0F)).reshape((n_blocks, -1, 32)).astype(np.float32)

        return (d * qs - dm).reshape((n_blocks, QK_K))


class Q5_K(__Quant, qtype=GGMLQuantizationType.Q5_K):
    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        dmin, rest = np.hsplit(rest, [2])
        scales, rest = np.hsplit(rest, [Q4_K.K_SCALE_SIZE])
        qh, qs = np.hsplit(rest, [QK_K // 8])

        d = d.view(np.float16).astype(np.float32)
        dmin = dmin.view(np.float16).astype(np.float32)

        sc, m = Q4_K.get_scale_min(scales)

        d = (d * sc.astype(np.float32)).reshape((n_blocks, -1, 1))
        dm = (dmin * m.astype(np.float32)).reshape((n_blocks, -1, 1))

        ql = qs.reshape((n_blocks, -1, 1, 32)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2, 1))
         qh = qh.reshape((n_blocks, -1, 1, 32)) >> np.array([i for i in range(8)], dtype=np.uint8).reshape((1,
1, 8, 1))
        ql = (ql & np.uint8(0x0F)).reshape((n_blocks, -1, 32))
        qh = (qh & np.uint8(0x01)).reshape((n_blocks, -1, 32))
        q = (ql | (qh << np.uint8(4))).astype(np.float32)

        return (d * q - dm).reshape((n_blocks, QK_K))


class Q6_K(__Quant, qtype=GGMLQuantizationType.Q6_K):
    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        ql, rest = np.hsplit(blocks, [QK_K // 2])
        qh, rest = np.hsplit(rest, [QK_K // 4])
        scales, d = np.hsplit(rest, [QK_K // 16])

        scales = scales.view(np.int8).astype(np.float32)
        d = d.view(np.float16).astype(np.float32)
        d = (d * scales).reshape((n_blocks, QK_K // 16, 1))

        ql = ql.reshape((n_blocks, -1, 1, 64)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2, 1))
        ql = (ql & np.uint8(0x0F)).reshape((n_blocks, -1, 32))
        qh = qh.reshape((n_blocks, -1, 1, 32)) >> np.array([0, 2, 4, 6], dtype=np.uint8).reshape((1, 1, 4, 1))
        qh = (qh & np.uint8(0x03)).reshape((n_blocks, -1, 32))
        q = (ql | (qh << np.uint8(4))).astype(np.int8) - np.int8(32)
        q = q.reshape((n_blocks, QK_K // 16, -1)).astype(np.float32)
```

```
        return (d * q).reshape((n_blocks, QK_K))


class TQ1_0(__Quant, qtype=GGMLQuantizationType.TQ1_0):
    @classmethod
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d = abs(blocks).max(axis=-1, keepdims=True)
        with np.errstate(divide="ignore"):
            id = np.where(d == 0, 0, 1 / d)
        qs = np_roundf(blocks * id)
        qs = (qs.astype(np.int8) + np.int8(1)).astype(np.uint8)

        qs0, qs1, qh = qs[..., :(32 * 5)], qs[..., (32 * 5):(48 * 5)], qs[..., (48 * 5):]
         qs0 = qs0.reshape((n_blocks, -1, 5, 32)) * np.array([81, 27, 9, 3, 1], dtype=np.uint8).reshape((1, 1,
5, 1))
        qs0 = np.sum(qs0, axis=-2).reshape((n_blocks, -1))
         qs1 = qs1.reshape((n_blocks, -1, 5, 16)) * np.array([81, 27, 9, 3, 1], dtype=np.uint8).reshape((1, 1,
5, 1))
        qs1 = np.sum(qs1, axis=-2).reshape((n_blocks, -1))
        qh = qh.reshape((n_blocks, -1, 4, 4)) * np.array([81, 27, 9, 3], dtype=np.uint8).reshape((1, 1, 4, 1))
        qh = np.sum(qh, axis=-2).reshape((n_blocks, -1))
        qs = np.concatenate([qs0, qs1, qh], axis=-1)
        qs = (qs.astype(np.uint16) * 256 + (243 - 1)) // 243

        qs = qs.astype(np.uint8)
        d = d.astype(np.float16).view(np.uint8)

        return np.concatenate([qs, d], axis=-1)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        qs, rest = np.hsplit(blocks, [(QK_K - 4 * QK_K // 64) // 5])
        qh, d = np.hsplit(rest, [QK_K // 64])

        d = d.view(np.float16).astype(np.float32)

        qs0, qs1 = qs[..., :32], qs[..., 32:]
         qs0 = qs0.reshape((n_blocks, -1, 1, 32)) * np.array([1, 3, 9, 27, 81], dtype=np.uint8).reshape((1, 1,
5, 1))
        qs0 = qs0.reshape((n_blocks, -1))
         qs1 = qs1.reshape((n_blocks, -1, 1, 16)) * np.array([1, 3, 9, 27, 81], dtype=np.uint8).reshape((1, 1,
5, 1))
        qs1 = qs1.reshape((n_blocks, -1))
        qh = qh.reshape((n_blocks, -1, 1, 4)) * np.array([1, 3, 9, 27], dtype=np.uint8).reshape((1, 1, 4, 1))
        qh = qh.reshape((n_blocks, -1))
        qs = np.concatenate([qs0, qs1, qh], axis=-1)
        qs = ((qs.astype(np.uint16) * 3) >> 8).astype(np.int8) - np.int8(1)

        return (d * qs.astype(np.float32))
```

```
class TQ2_0(__Quant, qtype=GGMLQuantizationType.TQ2_0):
    @classmethod
    def quantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d = abs(blocks).max(axis=-1, keepdims=True)
        with np.errstate(divide="ignore"):
            id = np.where(d == 0, 0, 1 / d)
        qs = np_roundf(blocks * id)
        qs = (qs.astype(np.int8) + np.int8(1)).astype(np.uint8)

        qs = qs.reshape((n_blocks, -1, 4, 32)) << np.array([0, 2, 4, 6], dtype=np.uint8).reshape((1, 1, 4, 1))
        qs = qs[..., 0, :] | qs[..., 1, :] | qs[..., 2, :] | qs[..., 3, :]
        qs = qs.reshape((n_blocks, -1))

        d = d.astype(np.float16).view(np.uint8)

        return np.concatenate([qs, d], axis=-1)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        qs, d = np.hsplit(blocks, [QK_K // 4])

        d = d.view(np.float16).astype(np.float32)

        qs = qs.reshape((n_blocks, -1, 1, 32)) >> np.array([0, 2, 4, 6], dtype=np.uint8).reshape((1, 1, 4, 1))
        qs = (qs & 0x03).reshape((n_blocks, -1)).astype(np.int8) - np.int8(1)

        return (d * qs.astype(np.float32))


class IQ2_XXS(__Quant, qtype=GGMLQuantizationType.IQ2_XXS):
    ksigns: bytes = (
        b"\x00\x81\x82\x03\x84\x05\x06\x87\x88\x09\x0a\x8b\x0c\x8d\x8e\x0f"
        b"\x90\x11\x12\x93\x14\x95\x96\x17\x18\x99\x9a\x1b\x9c\x1d\x1e\x9f"
        b"\xa0\x21\x22\xa3\x24\xa5\xa6\x27\x28\xa9\xaa\x2b\xac\x2d\x2e\xaf"
        b"\x30\xb1\xb2\x33\xb4\x35\x36\xb7\xb8\x39\x3a\xbb\x3c\xbd\xbe\x3f"
        b"\xc0\x41\x42\xc3\x44\xc5\xc6\x47\x48\xc9\xca\x4b\xcc\x4d\x4e\xcf"
        b"\x50\xd1\xd2\x53\xd4\x55\x56\xd7\xd8\x59\x5a\xdb\x5c\xdd\xde\x5f"
        b"\x60\xe1\xe2\x63\xe4\x65\x66\xe7\xe8\x69\x6a\xeb\x6c\xed\xee\x6f"
        b"\xf0\x71\x72\xf3\x74\xf5\xf6\x77\x78\xf9\xfa\x7b\xfc\x7d\x7e\xff"
    )

    # iq2xxs_grid, but with each byte of the original packed in 2 bits,
    # by mapping 0x08 to 0, 0x19 to 1, and 0x2b to 2.
    grid_shape = (256, 8)
    grid_map = (0x08, 0x19, 0x2b)
    grid_hex = (
        b"00000200050008000a0011001400200022002800 2a0041004400500058006100"
        b"6400800082008a00a2000101040110011501400184019801000202022202 8202"
        b"0104040410042104240440044204480460048104840490 04a404000502050805"
```

```
        b"2005460569058005910509061006400684406a406000805080808140828084108"
        b"4408500852088808040940090020a140a011004101010211040106010841090010"
        b"9510001108112011501150115a11801124124512001408142014251449148014181 5"
        b"6215001616160118041810184018811800190519a019511a002002200a204420"
        b"6120802082202921482100220222012404241024402456240025412564259026"
        b"082820289428442a01400440104018402140244040404840564060408140844 0"
        b"9040004120416141804185410142104248425642684200440844204480449944"
        b"1245244500460148044810484048454800495849614982494 54a904a00500850"
        b"1150195020508050885004514251a4519152905492540a550156545600581158"
        b"195864584059085a046010604060686000615561186260620064056410651265"
        b"8465426800800280 0a80418082800481188140811182018404841084158440 84"
        b"6084008546859485098640866086028804891 18a0490109024904090a1901691"
        b"8091459200942294449451958198209902a050a085a009a100a218a450a804a9"
    )

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, qs = np.hsplit(blocks, [2])

        d = d.view(np.float16).astype(np.float32)

        qs = qs.view(np.uint32).reshape(n_blocks, -1, 2)

        db = d * (np.float32(0.5) + (qs[..., 1] >> 28).astype(np.float32)) * np.float32(0.25)
        db = db.reshape((n_blocks, -1, 1, 1))

        # get the sign indices and unpack the bits
         signs = qs[..., 1].reshape((n_blocks, -1, 1)) >> np.array([0, 7, 14, 21], dtype=np.uint32).reshape((1,
1, 4))
        ksigns = np.frombuffer(cls.ksigns, dtype=np.uint8).reshape((1, 1, 1, 128))
        signs = (signs & np.uint32(0x7F)).reshape((n_blocks, -1, 4, 1))
        signs = np.take_along_axis(ksigns, signs, axis=-1)
                  signs  =  signs.reshape((n_blocks,  -1,  4,  1))  >>  np.array([i  for  i  in  range(8)],
dtype=np.uint8).reshape((1, 1, 1, 8))
        signs = signs & np.uint8(0x01)
        signs = np.where(signs == 0, np.float32(1), np.float32(-1))
        signs = signs.reshape((n_blocks, -1, 4, 8))

        assert cls.grid is not None
          grid = np.take_along_axis(cls.grid, qs[..., 0].copy().view(np.uint8).reshape((n_blocks, -1, 1, 1)),
axis=-2)
        grid = grid.reshape((n_blocks, -1, 4, 8))

        return (db * grid * signs).reshape((n_blocks, -1))


class IQ2_XS(__Quant, qtype=GGMLQuantizationType.IQ2_XS):
    # iq2xs_grid, but with each byte of the original packed in 2 bits,
    # by mapping 0x08 to 0, 0x19 to 1, and 0x2b to 2.
    grid_shape = (512, 8)
    grid_map = (0x08, 0x19, 0x2b)
    grid_hex = (
```

```
        b"0000020005000800a0011001400160019002000220025002800410044004600"
        b"4900500052005500580061006400800082008500880091009400990 0a0000101"
        b"0401060109011001120115011801 1a012101240140014201450148015101540 1"
        b"6001680181018401900100020202050208021102140220024102440250025502"
        b"80028a02010404040604090410041204150418042104240440044204450448 04"
        b"5104540456046004810484049004000502050505080511051405200541054405"
        b"5005610580050106040610062606400642068406000802080508080a081108"
        b"1408200825084108440850085808800 8a008aa0801090409100940098109890 9"
        b"000a200a280a960aa00a011004100610091010101210151018102110241040 10"
        b"4210451048105110541060106a1081108410901000110211051108111111141 1"
        b"2011411144115011801194119611011204120612101240126012001402140514"
        b"0814111414142014411444144914501464148014011504151015401500161416"
        b"4916011804181018121840185418861800190519661951 1aa91a002002200520"
        b"08200a2011201420202041204420502080 20a020012104211021402148216521"
        b"002222228022a822012404241024292440240025412552259925012 61a26a626"
        b"002808280a28202855288828a22868299029082a202a822a882a8a2a01400440"
        b"0640094010401240154018402140244040404240454048404a40514054406040"
        b"6540814084409040004102410541084111411441204141414441504180418541"
        b"a241014204421042122942404200440244054408441144144419442044414 4"
        b"4444504480449444014504451045244540459a4500460a464446504601480448"
        b"1048404845485448624800491149444950496949044a00500250055008501150"
        b"1450205028504150445050508050015104511051155140514251005244 52aa52"
        b"0154045410542154405460548154a154005508558055885521566856a1560058"
        b"1458415850589958 1a5940594259855a0160046010604060546062608660a960"
        b"006124624a629262006416641065406545 65a46501686a682569066a546a626a"
        b"0080028005800880118014802080 2a80418044805080808280a880aa800181"
        b"0481068110814081518159810082208280828282a082a8820184048410841284"
        b"1584408460848984008544 85a58518866a860088088825885a8880888288a888"
        b"0689228a808a888a968aa88a0190049010904090569084900091229164915692"
        b"8992009405944494509458942995909592954196519 8a6984999159a609a00a0"
        b"02a008a00aa020a02aa0a0a051a159a1a6a100a202a208a22aa280a2a0a240a4"
        b"95a465a698a60aa820a822a828a8a0a8a8a804a984a986a928aa2aaa91aaaaaa"
    )

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        qs, scales = np.hsplit(rest, [2 * QK_K // 8])

        d = d.view(np.float16).astype(np.float32)
        qs = qs.view(np.uint16)

        scales = scales.reshape((n_blocks, -1, 1)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2))
        scales = (scales & 0x0F).reshape((n_blocks, -1))
        db = d * (np.float32(0.5) + scales) * np.float32(0.25)
        db = db.reshape((n_blocks, -1, 1, 1))

        # get the sign indices and unpack the bits
        signs = np.frombuffer(IQ2_XXS.ksigns, dtype=np.uint8).reshape(1, 1, 128)
        signs = np.take_along_axis(signs, (qs >> 9).reshape((n_blocks, -1, 1)), axis=-1)
        signs = signs.reshape((n_blocks, -1, 1)) >> np.array([i for i in range(8)], dtype=np.uint8).reshape((1,
1, 8))
```

```
        signs = signs & np.uint8(0x01)
        signs = np.where(signs == 0, np.float32(1), np.float32(-1))
        signs = signs.reshape((n_blocks, -1, 2, 8))

        assert cls.grid is not None
        grid = np.take_along_axis(cls.grid, (qs & np.uint16(511)).reshape((n_blocks, -1, 1, 1)), axis=-2)
        grid = grid.reshape((n_blocks, -1, 2, 8))

        return (db * grid * signs).reshape((n_blocks, -1))


class IQ2_S(__Quant, qtype=GGMLQuantizationType.IQ2_S):
    # iq2s_grid, but with each byte of the original packed in 2 bits,
    # by mapping 0x08 to 0, 0x19 to 1, and 0x2b to 2.
    grid_shape = (1024, 8)
    grid_map = (0x08, 0x19, 0x2b)
    grid_hex = (
        b"00000200050008000a00110014001600190020002200250028004100440046 00"
        b"490050005200550058006100640066006900800082008500880091009400a000"
        b"a500aa000101040106010901100112011501180121012401400142014501480 1"
        b"510154015601590160016501680181018401900192019501a101a40100020202"
        b"050208021102140220022a02410244024602490250025502800285028a029402"
        b"a202010404040604090410041204150418042104240426042904400442044504"
        b"48044a045104540456045904600462046504810484048604890490049504980 4"
        b"a104a404000502050508050a05110514051605190520052505280541054405"
        b"460549055005520555055805610564058005820585058805910594 05a0050106"
        b"040606060906100615064006450648065106540660068106840690060008020 8"
        b"050808081108140816081908200825082a0841084408460849085008520855 08"
        b"580861086408800885089408aa08010904091009120915091809210940094509"
        b"480951095409600981099009000a110a140a220a280a2a0a500a990a01100410"
        b"061009101010121015101810211024102610401042104510481051105410561 0"
        b"591060106210651068108110841086109010951098 10a110a410001102110511"
        b"08110a1111111411161119112011221125112811411144114611491150115211"
        b"551158116111641180118211851188119111941101120412091210121512211 2"
        b"241240124512511254128112841290120014021405140814111414141614191 4"
        b"201425142814411444144614491450145214551458146114641480148214851 4"
        b"881491149414a01401150415061509151015121515151815211524154015421 5"
        b"451548155115541560158115841590150016051608161116141620164116441 6"
        b"50168016aa160118041806180918101815181818211840184218451848185118"
        b"541860188118841800190219051908191119141920194119441950196919a219"
        b"041a101a401a561a00200220052008201120142016201920202025202a204120"
        b"442050205220552064208020 8a209420aa200121042110211221152121214021"
        b"422145215121542160218121842190210022 0a22222228222a22442250228822"
        b"8a22a822012404240624092410241524182421242424402442244524482451 24"
        b"542460248124842490240025052508251125142520254125442550256625802 5"
        b"012604261026402659260028052811281428412844285028 8a28aa2801290429"
        b"102995290a2a222a642a882a8a2a0140044006400940104012401540180 1a40"
        b"214024402640404042404540484 04a40514054405640594060406240650814 0"
        b"844090409540984 0a140a44000410241054108411141144116411941204122 41"
        b"254141414441464149415041524155415841614164418041824185418841914 1"
        b"9441a041014204421042124215421842244240424542484251425442604281 42"
        b"844200440244054408440a44114414441644194420442244254428444144444 4"
        b"464449445044524455445844614464448044824485448844914494 44a0440145"
        b"044506450945104512451545184521452445404542454545484551455445604 5"
```

```
        b"6a458145844590450046024605460846114614462046414644650468046a546"
        b"0148044809481048124815481848214824484048424845484848514854486048"
        b"8448904800490249054908491149144920494149444950498049964901 4a044a"
        b"104a404a005002500550085011501450165019502050225025502850415044 50"
        b"4650495050505250555058506150645080508250855088509150945001510451"
        b"0651095110511251155118512151245140514251455148515151545160518151"
        b"8451905100520552085211521452205241524452505269528052015404540654"
        b"0954105412541554185421542454405442544554485451545454605481548454"
        b"9054005502550555085511551455205541554455505580550156045610562656"
        b"4056005802580558085811581458205841584458505 85a588058015904591059"
        b"4059005a195a855aa85a0160046006601060126015601860216024604060 4560"
        b"4860516054606060846090600061026105610861116114612061416144615061"
        b"8061996104621062406256 62a162006405640864116414642064416444645064"
        b"8064016504651065406 54a6568659265006694660168046810686568986800 69"
        b"2a69426aa16a008002800580088011801480198020802580418044805080528 0"
        b"5580588061808080858091809480018104810981108112811581188121812481"
        b"4081428145814881518154818181848190 81a981008205820a82118214824182"
        b"4482508201840484068409841084128415841884218440844284458448845184"
        b"5484608481848484908400850285058508851185148520854185448550858085"
        b"8a85018604861086298640860088058811881488418844885088a28801890489"
        b"40896589228a588a5a8a828aa28a019004900990109012901590189024904090"
        b"4290459048905190549060908190849090900091059111911491419144915091"
        b"5a9101920492109240 92a692009402940594089411941494209441944494509 4"
        b"8094969401950495109540959895a19500964696649601980498109826984098"
        b"a998009949995299909a00a005a00aa014a022a02aa041a044a050a0a2a0aaa0"
        b"40a165a102a20aa222a228a22aa282a288a28aa2a8a201a404a410a440a489a4"
        b"a4a400a519a551a60aa828a8a2a854a986a908aa0aaa20aa22aa28aa88aaaaaa"
    )

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        qs, rest = np.hsplit(rest, [QK_K // 8])
        signs, rest = np.hsplit(rest, [QK_K // 8])
        qh, scales = np.hsplit(rest, [QK_K // 32])

        d = d.view(np.float16).astype(np.float32)

        scales = scales.reshape((n_blocks, -1, 1)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2))
        scales = (scales & 0x0F).reshape((n_blocks, -1))
        db = d * (np.float32(0.5) + scales) * np.float32(0.25)
        db = db.reshape((n_blocks, -1, 1, 1))

        # unpack the sign bits
        signs = signs.reshape((n_blocks, -1, 1)) >> np.array([i for i in range(8)], dtype=np.uint8).reshape((1,
1, 8))
        signs = signs & np.uint8(0x01)
        signs = np.where(signs == 0, np.float32(1), np.float32(-1))
        signs = signs.reshape((n_blocks, -1, 2, 8))

        qh = qh.reshape((n_blocks, -1, 1)) >> np.array([0, 2, 4, 6], dtype=np.uint8).reshape((1, 1, 4))
        qs = qs.astype(np.uint16) | ((qh & 0x03).astype(np.uint16) << 8).reshape((n_blocks, -1))
```

```
        assert cls.grid is not None
        grid = np.take_along_axis(cls.grid, qs.reshape((n_blocks, -1, 1, 1)), axis=-2)
        grid = grid.reshape((n_blocks, -1, 2, 8))

        return (db * grid * signs).reshape((n_blocks, -1))


class IQ3_XXS(__Quant, qtype=GGMLQuantizationType.IQ3_XXS):
    grid_shape = (256, 4)
    grid_map = (0x04, 0x0c, 0x14, 0x1c, 0x24, 0x2c, 0x34, 0x3e)
    grid_hex = (
        b"0000020004001100130017002000220031004200730075000101030110011201"
        b"2101250130013201410154017001000202020402110220022202310233023702"
        b"5102570275020103070310031203250370031304370444045704730475040105"
        b"0705320552053506640610071407160743076107011003101010121021102310"
        b"3010321034104710501000110211111120112211011203121012121221123012"
        b"7212001302132013311346136613011405145014201524154615711505162217"
        b"4017002002201120132020202220262031204220012103210521102112212121"
        b"3021632167217021002202221122172220222222372240225522012310231423"
        b"7023742335245324032527254125742501270327162745270130103012302130"
        b"2330503065307230003102312031313144314631013203321032253252327232"
        b"1133333330344734723400350635223555351436363663363337603704401740"
        b"3540374053405740744120423742404260426642074345430444514464442545"
        b"4345704505471047124730471250415070500051065126515551145232527252"
        b"0253535310542354275472540255315550562457425724604460466064602161"
        b"6161176264623063366344640565526533660367216703700570077010703270"
        b"5270267140711272457252720073157333736073217441740075027524753076"
    )

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        qs, scales = np.hsplit(rest, [QK_K // 4])

        d = d.view(np.float16).astype(np.float32)
        scales = scales.view(np.uint32)

        db = d * (np.float32(0.5) + (scales >> 28).astype(np.float32)) * np.float32(0.5)
        db = db.reshape((n_blocks, -1, 1, 1))

        # get the sign indices and unpack the bits
         signs = scales.reshape((n_blocks, -1, 1)) >> np.array([0, 7, 14, 21], dtype=np.uint32).reshape((1, 1,
4))
        ksigns = np.frombuffer(IQ2_XXS.ksigns, dtype=np.uint8).reshape((1, 1, 1, 128))
        signs = (signs & np.uint32(0x7F)).reshape((n_blocks, -1, 4, 1))
        signs = np.take_along_axis(ksigns, signs, axis=-1)
                  signs  =  signs.reshape((n_blocks,  -1,  4,  1))  >>  np.array([i  for  i  in  range(8)],
dtype=np.uint8).reshape((1, 1, 1, 8))
        signs = signs & np.uint8(0x01)
        signs = np.where(signs == 0, np.float32(1), np.float32(-1))
        signs = signs.reshape((n_blocks, -1, 4, 8))
```

```
        assert cls.grid is not None
        grid = np.take_along_axis(cls.grid, qs.reshape((n_blocks, -1, 1, 1)), axis=-2)
        grid = grid.reshape((n_blocks, -1, 4, 8))

        return (db * grid * signs).reshape((n_blocks, -1))


class IQ3_S(__Quant, qtype=GGMLQuantizationType.IQ3_S):
    grid_shape = (512, 4)
    grid_map = (0x01, 0x03, 0x05, 0x07, 0x09, 0x0b, 0x0d, 0x0f)
    grid_hex = (
        b"0000010002000500070010001100120014001600200021002500330040004200"
        b"4500470051005300600062007100740077000001010102010401100111011501"
        b"2001230127013101350144016101650172010002010205020702100213021602"
        b"2102250230023402420245024702510253027002730203031103150320032203"
        b"3103330336034403500352036703710375030004130417042104240432044004"
        b"4304510470040205040520052205260533054105450547056605730506061106"
        b"1306310652067106000702070407200722072607330750075407001001100210"
        b"0410101011101310151017102010221031103410361054105610611072100011"
        b"0111031106111011141121113011331141115011521170117611001212121512"
        b"1712201224123212401243125512601272120113041307131013131321132713"
        b"3013341341136213701303140514121414143114331442144614501454140115"
        b"1015131521153015321551152016241627164416461601170317101712172117"
        b"3517411762177017002001200320052007201020122014201620212023202720"
        b"3020322041204320452050205220672070207320752000210221102113211721"
        b"2221252131213421422151210122042207222122322302237224122532257 22"
        b"7122742200230223052311232223242331233323422350236623012407242024"
        b"2324322435244124722475240425112522253725402553257025002602260726"
        b"2126552661260527112726273027432750270230113013301530173022303130"
        b"3330353042304430473051306330713001310331053114312131233140316031"
        b"7231763100321232203232323432503201331033143321332333273330334133"
        b"4333473355337333033411341634223431345234603464340135103512352535"
        b"3235443556357335163641360137033720372237353700400440124020402440"
        b"2740324041405040704002410741114113412241304135414341514155410142"
        b"0342104215422142334240425742624270420443114313432043224331433543"
        b"0044024424443744404471440545074521456245134634466046104715473047"
        b"4347514702501050145022504050445047505250665074500151035105511251"
        b"2151325172510052115223523052365253520253075310532753445351536553"
        b"7353015404542054325446541255265551555355425602570457225711601360"
        b"1560316033606060006120612761646112623462426255626262706200631463"
        b"2163406325644364626400650365346560650566406611671367007004700770"
        b"2070227036704070547062700271117124714371457101720472107216722172"
        b"3072517202733273357353730174057413742074507422754275027631760077"
    )

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        qs, rest = np.hsplit(rest, [QK_K // 4])
        qh, rest = np.hsplit(rest, [QK_K // 32])
        signs, scales = np.hsplit(rest, [QK_K // 8])
```

```
        d = d.view(np.float16).astype(np.float32)

        scales = scales.reshape((n_blocks, -1, 1)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2))
        scales = (scales & 0x0F).reshape((n_blocks, -1))
        db = d * (1 + 2 * scales)
        db = db.reshape((n_blocks, -1, 1, 1))

        # unpack the sign bits
        signs = signs.reshape((n_blocks, -1, 1)) >> np.array([i for i in range(8)], dtype=np.uint8).reshape((1,
1, 8))
        signs = signs & np.uint8(0x01)
        signs = np.where(signs == 0, np.float32(1), np.float32(-1))
        signs = signs.reshape((n_blocks, -1, 4, 8))

        qh = qh.reshape((n_blocks, -1, 1)) >> np.array([i for i in range(8)], dtype=np.uint8)
        qh = (qh & 0x01).astype(np.uint16).reshape((n_blocks, -1))
        qs = qs.astype(np.uint16) | (qh << 8)

        assert cls.grid is not None
        grid = np.take_along_axis(cls.grid, qs.reshape((n_blocks, -1, 1, 1)), axis=-2)
        grid = grid.reshape((n_blocks, -1, 4, 8))

        return (db * grid * signs).reshape((n_blocks, -1))


class IQ1_S(__Quant, qtype=GGMLQuantizationType.IQ1_S):
    # iq1s_grid, with each byte packed into 2 bits
    # -1, 0, 1 <=> 0, 1, 2
    grid_shape = (2048, 8)
    grid_map = (-1, 0, 1)
    grid_hex = (
        b"00000200050008000a00110015002000220028002a0045005100540056006500"
        b"8000820088008a009500a000a200a800aa00040105011101140116011901 1a01"
        b"2501410146014901520155015a0161016401660168018501910194019601a501"
        b"0002020208020a0215022002220228022a02450251025902640269028002 8202"
        b"88028a02910295029902a002a202a802aa02110414041604250441044904 5504"
        b"5a0464046504910499 04a504010504050505060515051805 1a05290540054505"
        b"4a055005510554055505560559056005620565056805 6a058105910595059805"
        b"9a05a105a405a505a605a90514061906410644065006520655065806600661 06"
        b"6606690685069106940699060008020808080a08150820082208280 82a084508"
        b"5108560865088008820888088a089508a008a208a808aa0805091109140919 09"
        b"2409250941095009510955096109640969099109940996099909a509000a020a"
        b"080a0a0a150a200a220a280a2a0a450a510a590a610a650a800a820a850a880a"
        b"8a0a950aa00aa20aa80aaa0a10101110141019102410251041104410 41050105510"
        b"5810611064106510691091109410961 0a110a5100111041106110911101112 11"
        b"151118112111241129114511 4a11501151115211541155115611591160116511"
        b"841192119511a111a411111214121612251240124612491252125512581 25a12"
        b"6412661285129112941296 12a5120114061409141414151418141914211426 14"
        b"4114451446144814 4a14511454145514561459146214651468148414891490 14"
        b"9414951498149914 9a14a114a414a514a914021505150a151115141515151615"
        b"1915201522152515281 52a154115441545154615511552155415551556155915"
        b"5a156115641565156615691580158215841585158815 8a159015911594159515"
        b"961599159a15a015a215a515011604160516061615161618161a1621162616"
```

b"401642164416451648164a16511655165616581659166116641665166816691 6"
b"6a1686168a1692169516a416a9161118161825184118441846184918501855 18"
b"58185a18601861186418661869188518911894 18a5181019121915191a192119"
b"25194219441945194819511954195519561959195a19601965196a1989199119"
b"921995199819a119a619a919091a161a241a261a441a461a491a501a521a551a"
b"581a611a661a691a851a911a961a9a1a0020022008200a201520202022202520"
b"28202a2045205120592061206520802082208820 8a209520a020a220a520a820"
b"aa200521112114211921252142214421492155215821 5a216121642165216621"
b"8521902196219921a521012208220a2211221522202222222822 2a2245225122"
b"562259226522812288228a2291229522a022a222a822aa220524142416241924"
b"252444244524462449245224552458245a24662485249124942499 24a124a524"
b"092515252125292540254525482551255425552559256225652568258925902 5"
b"9425952598259a25a125a425a625a92505261026122619262526412649265526"
b"602661266926842686269026 9a260028022808280a2815282028222828282a28"
b"45285128542865288028822888288a28a028a228a828aa280929112914291929"
b"252946294929522955296129642966296929852990299629992 9a429a529002a"
b"022a082a0a2a202a222a282a2a2a452a512a562a592a652a802a822a882a8a2a"
b"952aa02aa22aa82aaa2a0540114016402540494052405540584 05a4061406440"
b"664094409940a140a64000410141044106410941124115411641184 11a412141"
b"264129414541484 14a41514154415541564159415a41654168416a4181418441"
b"8641904192419541a041a141a241054211421442164225424142524255425a42"
b"6442694289429442a542014415441944294445444844 4a445144544455445644"
b"6144624465446844 6a448144864489449044924495 44a044a144a94401450245"
b"05450a45114514451545164519452045254 52a4541454445454546454945504 5"
b"5145544555455645584559456145644565456645694582458445854588459145"
b"9445954596459945 9a45a545a845aa450146054609461446154618461a462146"
b"2446294640464246454648465046514652465546564659466246654668468146"
b"85468a4694469546a146a446a6460548114815481a4825484248494850485548"
b"5848614864486648694885489148944896489948a5480149054906490a491049"
b"1449154918492149244926494049454 94a49514952495449554956495949604 9"
b"6249654966496a4986498949924995496499849a149a449a649a949164a444a"
b"464a494a554a584a5a4a644a694a944aa54a01500450055006500950125015 50"
b"1a502150245029504050455048505150545055505650595065506850865089 50"
b"95509850a050a150a650a9500551085109510a5111511451155116511851195 1"
b"20512551265128512a5141514451455146514951505151515251545155515651"
b"585159515a516151645165516651695182518551915194519551965195 19951a051"
b"a551aa5101520652125215521a522152245242524524 5a525152545255525652"
b"595262526552855290529252955299529a52a4520454055411541454155416 54"
b"1854195421542554285 42a5441544454455446544 9544a545054515454545554"
b"5654585459545a5461546254645465546654695480548 8548a54915494549554"
b"96549954a154a454a554aa5401550255045505550655095510551155125514 55"
b"1555165519551a5521552455255526552955405541554255445545554655485 5"
b"4955505551555255545555555655585559555a5560556155645565556655685 5"
b"69556a5581558455855589558a559055915594559555965598559955a155a455"
b"a555a655a955005601560256045606560856095611561456155618561962056"
b"2156225624562556265628562956415645564656485649564a56505651565256"
b"5456555656565856595 65a566156645665566956825685568656885689568a56"
b"915695569a56a256a556a656a856a956045805580658095810581558185821 58"
b"2a58455848584a585158545855585658585859586058625864586558825885 8"
b"9058925895589858a158a9580159025905590a591159145915591659195925 59"
b"4159445945594659495950595159525954595559659585959595a5961596459"
b"6559665969598159855989599159945995599659985995959a559045a085a155a"
b"1a5a205a255a265a295a455a485a495a515a555a565a585a595a625a655a685a"
b"6a5a815a8a5a925a955a965a985a9a5aa15a056014601660196025604460506 0"

```
    b"5560566058605a606160646066606960816096 60a560016104610661096112 61"
    b"156121612261266129614561496151615561566159616561666166 16a6184618a61"
    b"92619561a161a661a961116216621962406241624662556256625862606285 62"
    b"91629662a562116412641564166416641a642164266429644064426445644864 4a64"
    b"516454645564566459645a646064626465648464856489649064926494649564"
    b"966498649a64a164a464a964056508650a651165156516651965446545654665"
    b"496550655165546555655665596561656465656566656965866589658a659165"
    b"9565966599659a65a265a565a665a865026609661566206626662866296640 66"
    b"456648664a665166546655665666658665a666066656668668066826685668a66"
    b"9466966698669966a066a466a666aa661668196825684168526855685a686168"
    b"6968856891689868a668016904691069156921692469266929694069416945 69"
    b"466948695169546955695669596960696 5696a69826984698a699569a169a469"
    b"a569a969116a166a186a416a446a496a506a556a586a5a6a646a656a696a866a"
    b"946a986a9a6aa66a0080028008800a802080228028802a804580508051805480"
    b"568059806580808082808880 8a809580a080a280a880aa800581118114811681"
    b"198125814181448149815081528155815681588159816481668169818518 5818981"
    b"948196819981a5810082028208820a821582208222822882 2a82518254825982"
    b"658280828288828a829582a082a282a882aa821484198441844484518455 84"
    b"5a846184648469849484998401850985128515851a852685298540854185458 5"
    b"488551855485558568559855a856585668568856a858185848586858958590 85"
    b"928595859885a685118616861986258641864486496 4a865086558659865a86"
    b"618666866a86858691869a86a486008802880888 0a881588208822882882a88"
    b"418845885188548859886588698880888288888 8a889588a088a288a888aa88"
    b"058906891189148916892589418944894689498950895289558 95a8961896489"
    b"858996899989a589008a028a088a0a8a158a208a228a288a2a8a458a518a548a"
    b"568a808a828a888a8a8a958aa08aa28aa88aaa8a059011901690189019902590"
    b"419046904990559058905a9069906a908590919094909690999 0a59001910491"
    b"069109911091159118911a912191249126912991409145915091519154915591"
    b"569159916291659184918691929195919891a191a491a691a991059211921492"
    b"199225924492469249925092529255925892669269928592949296 92a9920194"
    b"049406941094159418942694404 4a945194549455945694589459946094 6194"
    b"629465948494869492949495949894a194a9940095059508950a9510951195"
    b"149515951695199521952595299 52a954195449545954695499550955195529 5"
    b"549555955695589559955a956195649565956695695819585958895915959 5"
    b"949595959695995 9a95a095a295a595a895aa950196049610961596199620 96"
    b"269629964596489649961965296559656965996659668968296849689968a96"
    b"929694969596a496a696a996059816981998259841984698509852985598569 8"
    b"5a986498659885989198969899 8a59804990699099109912991599189911a99"
    b"209921992499269940994299459948994a995199549955995699599962996599"
    b"66996a998199849990999299959 99a99a199a699059a159a259a449a469a499a"
    b"509a559a589a619a859a919a949a959a969a00a002a008a00aa015a020a022a0"
    b"28a02aa045a051a054a056a059a080a082a088a08aa095a0a0a0a2a0a8a0aaa0"
    b"05a109a111a114a116a119a11aa146a149a151a155a158a15aa161a164a185a1"
    b"90a192a196a199a102a208a20aa210a219a222a228a22aa245a251a256a259a2"
    b"65a280a282a288a28aa295a2a0a2a2a2a8a2aaa219a425a441a444a450a454a4"
    b"55a458a45aa461a465a466a468a469a485a406a509a510a512a515a518a526a5"
    b"29a542a545a551a554a555a556a559a565a56aa581a584a585a586a589a592a5"
    b"95a598a505a611a616a61aa621a625a644a646a64aa652a655a656a658a660a6"
    b"62a686a690a695a696a699a6a1a6a4a6a6a600a802a808a80aa820a822a828a8"
    b"2aa851a854a856a859a880a882a888a88aa895a8a0a8a2a8a8a8aaa805a914a9"
    b"19a921a925a941a950a955a95aa961a966a969a990a996a900aa02aa08aa0aaa"
    b"20aa22aa28aa2aaa51aa54aa56aa80aa82aa88aa8aaa95aaa0aaa2aaa8aaaaaa"
)
```

```
    delta = np.float32(0.125)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        qs, qh = np.hsplit(rest, [QK_K // 8])

        d = d.view(np.float16).astype(np.float32)
        qh = qh.view(np.uint16)

        dl = d * (2 * ((qh >> 12) & 7) + 1)
        dl = dl.reshape((n_blocks, -1, 1, 1))
        delta = np.where((qh & np.uint16(0x8000)) == 0, cls.delta, -cls.delta)
        delta = delta.reshape((n_blocks, -1, 1, 1))

        qh = qh.reshape((n_blocks, -1, 1)) >> np.array([0, 3, 6, 9], dtype=np.uint16).reshape((1, 1, 4))
        qs = qs.astype(np.uint16) | ((qh & 7) << 8).reshape((n_blocks, -1))

        assert cls.grid is not None
        grid = np.take_along_axis(cls.grid, qs.reshape((n_blocks, -1, 1, 1)), axis=-2)
        grid = grid.reshape((n_blocks, -1, 4, 8))

        return (dl * (grid + delta)).reshape((n_blocks, -1))


class IQ1_M(__Quant, qtype=GGMLQuantizationType.IQ1_M):
    grid_shape = IQ1_S.grid_shape
    grid_map = IQ1_S.grid_map
    grid_hex = IQ1_S.grid_hex

    delta = IQ1_S.delta

    # Okay *this* type is weird. It's the only one which stores the f16 scales in multiple parts.
    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        qs, rest = np.hsplit(blocks, [QK_K // 8])
        qh, scales = np.hsplit(rest, [QK_K // 16])

        # The f16 scale is packed across multiple bytes
        scales = scales.view(np.uint16)
                 d  =  (scales.reshape((n_blocks,  4))  &  np.uint16(0xF000))  >>  np.array([12,  8,  4,  0],
dtype=np.uint16).reshape((1, 4))
        d = d[..., 0] | d[..., 1] | d[..., 2] | d[..., 3]
        d = d.view(np.float16).astype(np.float32).reshape((n_blocks, 1))

        scales = scales.reshape(n_blocks, -1, 1) >> np.array([0, 3, 6, 9], dtype=np.uint16).reshape((1, 1, 4))
        scales = (scales & 0x07).reshape((n_blocks, -1))
        dl = d * (2 * scales + 1)
        dl = dl.reshape((n_blocks, -1, 2, 1, 1))
```

```
        qh = qh.reshape((n_blocks, -1, 1)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2))
        qs = qs.astype(np.uint16) | ((qh & 0x07).astype(np.uint16) << 8).reshape((n_blocks, -1))

        delta = np.where(qh & 0x08 == 0, cls.delta, -cls.delta)
        delta = delta.reshape((n_blocks, -1, 2, 2, 1))

        assert cls.grid is not None
        grid = np.take_along_axis(cls.grid, qs.reshape((n_blocks, -1, 1, 1)), axis=-2)
        grid = grid.reshape((n_blocks, -1, 2, 2, 8))

        return (dl * (grid + delta)).reshape((n_blocks, -1))


class IQ4_NL(__Quant, qtype=GGMLQuantizationType.IQ4_NL):
    kvalues = (-127, -104, -83, -65, -49, -35, -22, -10, 1, 13, 25, 38, 53, 69, 89, 113)

    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, qs = np.hsplit(blocks, [2])

        d = d.view(np.float16).astype(np.float32)

        qs = qs.reshape((n_blocks, -1, 1, cls.block_size // 2)) >> np.array([0, 4], dtype=np.uint8).reshape((1,
1, 2, 1))

        qs = (qs & np.uint8(0x0F)).reshape((n_blocks, -1, 1))

        kvalues = np.array(cls.kvalues, dtype=np.int8).reshape(1, 1, 16)
        qs = np.take_along_axis(kvalues, qs, axis=-1).astype(np.float32).reshape((n_blocks, -1))

        return (d * qs)


class IQ4_XS(__Quant, qtype=GGMLQuantizationType.IQ4_XS):
    @classmethod
    def dequantize_blocks(cls, blocks: np.ndarray) -> np.ndarray:
        n_blocks = blocks.shape[0]

        d, rest = np.hsplit(blocks, [2])
        scales_h, rest = np.hsplit(rest, [2])
        scales_l, qs = np.hsplit(rest, [QK_K // 64])

        d = d.view(np.float16).astype(np.float32)
        scales_h = scales_h.view(np.uint16)

        scales_l = scales_l.reshape((n_blocks, -1, 1)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2))
           scales_h = scales_h.reshape((n_blocks, 1, -1)) >> np.array([2 * i for i in range(QK_K // 32)],
dtype=np.uint16).reshape((1, -1, 1))
        scales_l = scales_l.reshape((n_blocks, -1)) & np.uint8(0x0F)
        scales_h = scales_h.reshape((n_blocks, -1)).astype(np.uint8) & np.uint8(0x03)

        scales = (scales_l | (scales_h << np.uint8(4))).astype(np.int8) - np.int8(32)
```

```
        dl = (d * scales.astype(np.float32)).reshape((n_blocks, -1, 1))

        qs = qs.reshape((n_blocks, -1, 1, 16)) >> np.array([0, 4], dtype=np.uint8).reshape((1, 1, 2, 1))
        qs = qs.reshape((n_blocks, -1, 32, 1)) & np.uint8(0x0F)

        kvalues = np.array(IQ4_NL.kvalues, dtype=np.int8).reshape((1, 1, 1, -1))
        qs = np.take_along_axis(kvalues, qs, axis=-1).astype(np.float32).reshape((n_blocks, -1, 32))

        return (dl * qs).reshape((n_blocks, -1))

==== qwen2_vl_surgery.py ====
import argparse
from typing import Dict

import torch
import numpy as np
from gguf import *
from transformers import (
    Qwen2VLForConditionalGeneration,
    Qwen2VLProcessor,
    AutoProcessor,
    Qwen2VLConfig
)


VISION = "clip.vision"


def k(raw_key: str, arch: str) -> str:
    return raw_key.format(arch=arch)


def to_gguf_name(name: str) -> str:
    og = name
    name = name.replace("text_model", "t").replace("vision_model", "v")
    name = name.replace("blocks", "blk").replace("embeddings.", "")
    name = name.replace("attn.", "attn_")
    name = name.replace("mlp.fc1", "ffn_down").replace("mlp.fc2", "ffn_up").replace("proj.", "out.")
    # name = name.replace("layrnorm", "ln").replace("layer_norm", "ln").replace("layernorm", "ln")
    name = name.replace("norm1", "ln1").replace("norm2", "ln2")
    name = name.replace("merger.mlp", 'mm')
    print(f"[to_gguf_name] {og} --> {name}")
    return name


def find_vision_tensors(qwen2vl, dtype) -> Dict[str, np.ndarray]:
    vision_model = qwen2vl.visual
    tensor_map = {}
    for name, ten in vision_model.state_dict().items():
        ten = ten.numpy()
        if 'qkv' in name:
            if ten.ndim == 2: # weight
                c3, _ = ten.shape
            else:             # bias
```

```
                c3 = ten.shape[0]
            assert c3 % 3 == 0
            c = c3 // 3
            wq = ten[:c]
            wk = ten[c: c * 2]
            wv = ten[c * 2:]
            tensor_map[to_gguf_name(f"vision_model.{name}").replace("qkv", "q")] = wq
            tensor_map[to_gguf_name(f"vision_model.{name}").replace("qkv", "k")] = wk
            tensor_map[to_gguf_name(f"vision_model.{name}").replace("qkv", "v")] = wv
        elif 'merger' in name:
            if name.endswith("ln_q.weight"):
                tensor_map['v.post_ln.weight'] = ten
            elif name.endswith("ln_q.bias"):
                tensor_map['v.post_ln.bias'] = ten
            else:
                # "merger.mlp.%d.weight/bias" --> "mm.%d.weight/bias"
                tensor_map[to_gguf_name(name)] = ten
        elif 'patch_embed.proj.weight' in name:
            # NOTE: split Conv3D into Conv2Ds
            c1, c2, kt, kh, kw = ten.shape
            assert kt == 2, "Current implmentation only support temporal_patch_size of 2"
            tensor_map["v.patch_embd.weight"] = ten[:, :, 0, ...]
            tensor_map["v.patch_embd.weight.1"] = ten[:, :, 1, ...]
        else:
            tensor_map[to_gguf_name(f"vision_model.{name}")] = ten

    for new_name, ten in tensor_map.items():
        if ten.ndim <= 1 or new_name.endswith("_norm.weight"):
            tensor_map[new_name] = ten.astype(np.float32)
        else:
            tensor_map[new_name] = ten.astype(dtype)
    tensor_map["v.position_embd.weight"] = np.zeros([10, 10], dtype=np.float32)  # dummy tensor, just here as a
placeholder
    return tensor_map


def main(args):
    if args.data_type == 'fp32':
        dtype = torch.float32
        np_dtype = np.float32
        ftype = 0
    elif args.data_type == 'fp16':
        dtype = torch.float32
        np_dtype = np.float16
        ftype = 1
    else:
        raise ValueError()

    local_model = False
    model_path = ""
    model_name = args.model_name
    print("model_name: ", model_name)
    qwen2vl = Qwen2VLForConditionalGeneration.from_pretrained(
        model_name, torch_dtype=dtype, device_map="cpu"
```

```
    )
    cfg: Qwen2VLConfig = qwen2vl.config  # type: ignore[reportAssignmentType]
    vcfg = cfg.vision_config

    if os.path.isdir(model_name):
        local_model = True
        if model_name.endswith(os.sep):
            model_name = model_name[:-1]
        model_path = model_name
        model_name = os.path.basename(model_name)
    fname_out = f"{model_name.replace('/', '-').lower()}-vision.gguf"

    fout = GGUFWriter(path=fname_out, arch="clip")
    fout.add_description("image encoder for Qwen2VL")

    fout.add_file_type(ftype)
    fout.add_bool("clip.has_text_encoder", False)
    fout.add_bool("clip.has_vision_encoder", True)
    fout.add_bool("clip.has_qwen2vl_merger", True)
    fout.add_string("clip.projector_type", "qwen2vl_merger")

    print(cfg.vision_config)
    if 'silu' in cfg.vision_config.hidden_act.lower():
        fout.add_bool("clip.use_silu", True)
        fout.add_bool("clip.use_gelu", False)
    elif 'gelu' in cfg.vision_config.hidden_act.lower():
        fout.add_bool("clip.use_silu", False)
        fout.add_bool("clip.use_gelu", 'quick' not in cfg.vision_config.hidden_act.lower())
    else:
        raise ValueError()

    tensor_map = find_vision_tensors(qwen2vl, np_dtype)
    for name, data in tensor_map.items():
        fout.add_tensor(name, data)

    fout.add_uint32("clip.vision.patch_size", vcfg.patch_size)
    fout.add_uint32("clip.vision.image_size", 14 * 40)  # some reasonable size that is divable by (14*2)
    fout.add_uint32(k(KEY_EMBEDDING_LENGTH, VISION), vcfg.embed_dim)
    fout.add_uint32("clip.vision.projection_dim", vcfg.hidden_size)
    fout.add_uint32(k(KEY_ATTENTION_HEAD_COUNT, VISION), vcfg.num_heads)
    fout.add_float32(k(KEY_ATTENTION_LAYERNORM_EPS, VISION), 1e-6)
    fout.add_uint32(k(KEY_BLOCK_COUNT, VISION), vcfg.depth)
      fout.add_uint32(k(KEY_FEED_FORWARD_LENGTH, VISION), 0)   # not sure what this does, put 0 here as a
placeholder
    fout.add_name(model_name)
    """
    HACK: Since vision rope related parameter aren't stored in the `Qwen2VLConfig,
            it will be hardcoded in the `clip_image_build_graph` from `clip.cpp`.
    """

    if local_model:
        processor: Qwen2VLProcessor = AutoProcessor.from_pretrained(model_path)
    else:
        processor: Qwen2VLProcessor = AutoProcessor.from_pretrained(model_name)
```

```
                fout.add_array("clip.vision.image_mean",    processor.image_processor.image_mean)    #    type:
ignore[reportAttributeAccessIssue]
                 fout.add_array("clip.vision.image_std",    processor.image_processor.image_std)    #    type:
ignore[reportAttributeAccessIssue]

    fout.write_header_to_file()
    fout.write_kv_data_to_file()
    fout.write_tensors_to_file()
    fout.close()
    print("save model as: ", fname_out)



if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("model_name", nargs='?', default="Qwen/Qwen2-VL-2B-Instruct")
    parser.add_argument("--data_type", nargs='?', choices=['fp32', 'fp16'], default="fp32")
    args = parser.parse_args()
    main(args)

==== ragment_decay_engine.py ====
# fragment_decay_engine.py
# ? Symbolic fragment rot system
# Rewrites fragment metadata to simulate aging, decay, and drift

import os
import yaml
import random
import asyncio
from datetime import datetime, timedelta
from pathlib import Path
from concurrent.futures import ThreadPoolExecutor

# === CORE TRUTH ===
CORE_AXIOM = {
    "claim": "Something must stay still so everything else can move.",
    "role": "axiom",
    "immutable": True
}

# Configurable decay rules
DECAY_RULES = {
    "certainty": lambda x: max(0.0, round(x - random.uniform(0.05, 0.2), 3)),
    "urgency": lambda x: max(0.0, round(x - random.uniform(0.01, 0.05), 3)),
    "doubt": lambda x: min(1.0, round(x + random.uniform(0.05, 0.2), 3)),
    "confidence": lambda x: max(0.0, round(x - random.uniform(0.1, 0.3), 3))
}

# Optional field shuffler
def shuffle_fields(fragment):
    if 'claim' in fragment and random.random() < 0.4:
        fragment['claim'] = f"[fragmented] {fragment['claim']}"
    if 'tags' in fragment and isinstance(fragment['tags'], list):
        random.shuffle(fragment['tags'])
    return fragment
```

```
# Age threshold (e.g. 10 days = eligible for decay)
DECAY_AGE_DAYS = 10

# Base path for fragments
FRAGMENTS_PATH = Path("C:/Users/PC/Desktop/Operation Future/Allinonepy/fragments")

# Rot target output
DECAYED_PATH = Path("C:/Users/PC/Desktop/Operation Future/Allinonepy/fragments/decayed")
DECAYED_PATH.mkdir(parents=True, exist_ok=True)

# Simulated timing feedback
SIMULATED_TIMERS = {
    "fan_rpm": lambda: random.uniform(0.85, 1.15),
    "spin_drive_latency": lambda: random.uniform(0.75, 1.25)
}

def get_decay_multiplier():
    # Average the simulated timing influence
    modifiers = [fn() for fn in SIMULATED_TIMERS.values()]
    return sum(modifiers) / len(modifiers)


def should_decay(file_path):
    modified = datetime.fromtimestamp(file_path.stat().st_mtime)
    return datetime.now() - modified > timedelta(days=DECAY_AGE_DAYS)


def decay_fragment(frag):
    if not isinstance(frag, dict):
        return frag

    # Skip axiom
    if frag.get("claim") == CORE_AXIOM["claim"]:
        frag["immutable"] = True
        return frag

    multiplier = get_decay_multiplier()

    # Apply decay rules
    for field, fn in DECAY_RULES.items():
        if field in frag:
            frag[field] = fn(frag[field] * multiplier)

    # Simulate drift
    frag = shuffle_fields(frag)
    frag['decayed'] = True
    frag['decay_timestamp'] = datetime.now().isoformat()
    frag['decay_modifier'] = round(multiplier, 3)
    return frag


def process_single_fragment(path):
    if should_decay(path):
```

```
        try:
            with open(path, 'r', encoding='utf-8') as f:
                data = yaml.safe_load(f)
            decayed = decay_fragment(data)
            out_path = DECAYED_PATH / path.name
            with open(out_path, 'w', encoding='utf-8') as f:
                yaml.safe_dump(decayed, f, sort_keys=False)
            print(f"? Decayed: {path.name} -> {out_path.name}")
        except Exception as e:
            print(f"WARNING  Failed to process {path.name}: {e}")


async def process_fragments():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as executor:
        tasks = []
        for path in FRAGMENTS_PATH.rglob("*.yaml"):
            tasks.append(loop.run_in_executor(executor, process_single_fragment, path))
        await asyncio.gather(*tasks)


if __name__ == "__main__":
    print("INFO Starting fragment decay scan (async)...")
    asyncio.run(process_fragments())
    print("[OK] Decay cycle complete.")

==== reader.py ====
#!/usr/bin/env python3
import logging
import sys
from pathlib import Path

logger = logging.getLogger("reader")

# Necessary to load the local gguf package
sys.path.insert(0, str(Path(__file__).parent.parent))

from gguf.gguf_reader import GGUFReader


def read_gguf_file(gguf_file_path):
    """
    Reads and prints key-value pairs and tensor information from a GGUF file in an improved format.

    Parameters:
    - gguf_file_path: Path to the GGUF file.
    """

    reader = GGUFReader(gguf_file_path)

    # List all key-value pairs in a columnized format
    print("Key-Value Pairs:") # noqa: NP100
    max_key_length = max(len(key) for key in reader.fields.keys())
    for key, field in reader.fields.items():
```

```
        value = field.parts[field.data[0]]
        print(f"{key:{max_key_length}} : {value}") # noqa: NP100
    print("----") # noqa: NP100

    # List all tensors
    print("Tensors:") # noqa: NP100
    tensor_info_format = "{:<30} | Shape: {:<15} | Size: {:<12} | Quantization: {}"
    print(tensor_info_format.format("Tensor Name", "Shape", "Size", "Quantization")) # noqa: NP100
    print("-" * 80) # noqa: NP100
    for tensor in reader.tensors:
        shape_str = "x".join(map(str, tensor.shape))
        size_str = str(tensor.n_elements)
        quantization_str = tensor.tensor_type.name
        print(tensor_info_format.format(tensor.name, shape_str, size_str, quantization_str)) # noqa: NP100


if __name__ == '__main__':
    if len(sys.argv) < 2:
        logger.info("Usage: reader.py <path_to_gguf_file>")
        sys.exit(1)
    gguf_file_path = sys.argv[1]
    read_gguf_file(gguf_file_path)

==== rebuild_neurostore_full.py ====
import os
import zipfile
import tarfile
from pathlib import Path
from datetime import datetime

# ========== Safe File Writer ==========
def write_file(path_parts, content):
    path = Path(*path_parts)
    path.parent.mkdir(parents=True, exist_ok=True)
    path.write_text(content.strip() + "\n", encoding="utf-8")
    print(f"[OK] Wrote {path}")

# ========== Backup NeuroStore ==========
def backup_neurostore():
    EXPORT_DIR = os.path.expanduser("~/neurostore/backups")
    SOURCE_DIRS = ["agents", "fragments", "logs", "meta", "runtime", "data"]
    os.makedirs(EXPORT_DIR, exist_ok=True)
    backup_name = f"neurostore_brain_{datetime.now().strftime('%Y%m%d_%H%M%S')}.tar.gz"
    backup_path = os.path.join(EXPORT_DIR, backup_name)
    with tarfile.open(backup_path, "w:gz") as tar:
        for folder in SOURCE_DIRS:
            if os.path.exists(folder):
                print(f"[OK] Archiving {folder}/")
                tar.add(folder, arcname=folder)
            else:
                print(f"[SKIP] Missing folder: {folder}")
    print(f"[OK] Brain backup complete -> {backup_path}")

# ========== Deep File Crawler ==========
```

```
def deep_file_crawl():
    import hashlib
    def hash_file(path, chunk_size=8192):
        try:
            hasher = hashlib.md5()
            with open(path, 'rb') as f:
                for chunk in iter(lambda: f.read(chunk_size), b""):
                    hasher.update(chunk)
            return hasher.hexdigest()
        except Exception as e:
            return f"ERROR: {e}"

    def crawl_directory(root_path, out_path):
        count = 0
        with open(out_path, 'w', encoding='utf-8') as out_file:
            for dirpath, _, filenames in os.walk(root_path):
                for file in filenames:
                    full_path = os.path.join(dirpath, file)
                    try:
                        stat = os.stat(full_path)
                        hashed = hash_file(full_path)
                        line = f"{full_path} | {stat.st_size} bytes | hash: {hashed}"
                    except Exception as e:
                        line = f"{full_path} | ERROR: {str(e)}"
                    out_file.write(line + "\n")
                    count += 1
                    if count % 100 == 0:
                        print(f"[OK] {count} files crawled...")
        print(f"[OK] Crawl complete. Total files: {count}")
        print(f"[OK] Full output saved to: {out_path}")

    BASE = "."
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_txt = f"neurostore_crawl_output_{timestamp}.txt"
    print(f"[OK] Starting deep crawl on: {BASE}")
    crawl_directory(BASE, output_txt)

# ========== Zip Project ==========
def zip_project():
    BASE = Path(".").resolve()
    zip_path = BASE.parent / f"{BASE.name}_project.zip"
    print(f"[OK] Zipping project to: {zip_path}")
    with zipfile.ZipFile(zip_path, "w", zipfile.ZIP_DEFLATED) as zipf:
        for file in BASE.rglob("*"):
            if file.is_file():
                zipf.write(file, arcname=file.relative_to(BASE))
    print("[OK] ZIP complete.")

# ========== Entrypoint ==========
if __name__ == "__main__":
    backup_neurostore()
    deep_file_crawl()
    zip_project()
# ========== symbol_seed_generator.py ==========
```

```
write_file(["symbol_seed_generator.py"], '''
import os
import yaml
import hashlib
from datetime import datetime

USE_LLM = False
MODEL_PATH = "models/mistral-7b-q4.gguf"
SEED_OUTPUT_DIR = "fragments/core/"
SEED_COUNT = 100

BASE_SEEDS = [
    "truth is important",
    "conflict creates learning",
    "change is constant",
    "observation precedes action",
    "emotion influences memory",
    "self seeks meaning",
    "logic guides belief",
    "doubt triggers inquiry",
    "energy becomes form",
    "ideas replicate",
    "something must stay_still so everything else can move"
]

def generate_id(content):
    return hashlib.sha256(content.encode()).hexdigest()[:12]

def to_fragment(statement):
    parts = statement.split()
    if len(parts) < 3:
        return None
    subj = parts[0]
    pred = parts[1]
    obj = "_".join(parts[2:])
    return {
        "id": generate_id(statement),
        "predicate": pred,
        "arguments": [subj, obj],
        "confidence": 1.0,
        "emotion": {
            "curiosity": 0.8,
            "certainty": 1.0
        },
        "tags": ["seed", "immutable", "core"],
        "immutable": True,
        "claim": statement,
        "timestamp": datetime.utcnow().isoformat()
    }

def save_fragment(fragment, output_dir):
    fname = f"frag_{fragment['id']}.yaml"
    path = os.path.join(output_dir, fname)
    with open(path, 'w') as f:
```

```
        yaml.dump(fragment, f)

def generate_symbolic_seeds():
    if not os.path.exists(SEED_OUTPUT_DIR):
        os.makedirs(SEED_OUTPUT_DIR)
    seed_statements = BASE_SEEDS[:SEED_COUNT]
    count = 0
    for stmt in seed_statements:
        frag = to_fragment(stmt)
        if frag:
            save_fragment(frag, SEED_OUTPUT_DIR)
            count += 1
    print(f"Generated {count} symbolic seed fragments in {SEED_OUTPUT_DIR}")

if __name__ == "__main__":
    generate_symbolic_seeds()
''')

# ========== token_agent.py ==========
write_file(["token_agent.py"], '''
import time
import random
import yaml
from pathlib import Path
from core.cortex_bus import send_message

FRAG_DIR = Path("fragments/core")

class TokenAgent:
    def __init__(self, agent_id="token_agent_01"):
        self.agent_id = agent_id
        self.frag_path = FRAG_DIR
        self.fragment_cache = []

    def load_fragments(self):
        files = list(self.frag_path.glob("*.yaml"))
        random.shuffle(files)
        for f in files:
            with open(f, 'r', encoding='utf-8') as file:
                try:
                    frag = yaml.safe_load(file)
                    if frag:
                        self.fragment_cache.append((f, frag))
                except yaml.YAMLError as e:
                    print(f"[{self.agent_id}] YAML error in {f.name}: {e}")

    def walk_fragment(self, path, frag):
        if 'claim' not in frag:
            return
        walk_log = {
            'fragment': path.name,
            'claim': frag['claim'],
            'tags': frag.get('tags', []),
            'confidence': frag.get('confidence', 0.5),
```

```
            'walk_time': time.time()
        }
        if random.random() < 0.2:
            walk_log['flag_mutation'] = True
        send_message({
            'from': self.agent_id,
            'type': 'walk_log',
            'payload': walk_log,
            'timestamp': int(time.time())
        })

    def run(self):
        self.load_fragments()
        for path, frag in self.fragment_cache:
            self.walk_fragment(path, frag)
            time.sleep(0.1)

if __name__ == "__main__":
    agent = TokenAgent()
    agent.run()
''')
# ========== backup_and_export.py ==========
write_file(["backup_and_export.py"], '''
import os
import tarfile
from datetime import datetime

EXPORT_DIR = os.path.expanduser("~/neurostore/backups")
SOURCE_DIRS = ["agents", "fragments", "logs", "meta", "runtime", "data"]

os.makedirs(EXPORT_DIR, exist_ok=True)
backup_name = f"neurostore_brain_{datetime.now().strftime('%Y%m%d_%H%M%S')}.tar.gz"
backup_path = os.path.join(EXPORT_DIR, backup_name)

with tarfile.open(backup_path, "w:gz") as tar:
    for folder in SOURCE_DIRS:
        if os.path.exists(folder):
            print(f"[+] Archiving {folder}/")
            tar.add(folder, arcname=folder)
        else:
            print(f"[-] Skipped missing folder: {folder}")

print(f"[OK] Brain backup complete -> {backup_path}")
''')

# ========== deep_file_crawler.py ==========
write_file(["deep_file_crawler.py"], '''
import os
import hashlib
from datetime import datetime

def hash_file(path, chunk_size=8192):
    try:
        hasher = hashlib.md5()
```

```
        with open(path, 'rb') as f:
            for chunk in iter(lambda: f.read(chunk_size), b""):
                hasher.update(chunk)
        return hasher.hexdigest()
    except Exception as e:
        return f"ERROR: {e}"

def crawl_directory(root_path, out_path):
    count = 0
    with open(out_path, 'w') as out_file:
        for dirpath, dirnames, filenames in os.walk(root_path):
            for file in filenames:
                full_path = os.path.join(dirpath, file)
                try:
                    stat = os.stat(full_path)
                    hashed = hash_file(full_path)
                    line = f"{full_path} | {stat.st_size} bytes | hash: {hashed}"
                except Exception as e:
                    line = f"{full_path} | ERROR: {str(e)}"
                out_file.write(line + "\\n")
                count += 1
                if count % 100 == 0:
                    print(f"[+] {count} files crawled...")

    print(f"[OK] Crawl complete. Total files: {count}")
    print(f"[OK] Full output saved to: {out_path}")

if __name__ == "__main__":
    BASE = "."
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_txt = f"neurostore_crawl_output_{timestamp}.txt"
    print(f"[*] Starting deep crawl on: {BASE}")
    crawl_directory(BASE, output_txt)
''')

# ========== boot_wrapper.py ==========
write_file(["boot_wrapper.py"], '''
import subprocess
import os
import platform
import time
import psutil
from pathlib import Path

SCRIPTS = [
    "deep_system_scan.py",
    "auto_configurator.py",
    "path_optimizer.py",
    "fragment_teleporter.py",
    "run_logicshredder.py"
]

LOG_PATH = Path("logs/boot_times.log")
LOG_PATH.parent.mkdir(exist_ok=True)
```

```
def run_script(name, timings):
    if not Path(name).exists():
        print(f"[boot] ? Missing script: {name}")
        timings.append((name, "MISSING", "-", "-"))
        return False

    print(f"[boot] ? Running: {name}")
    start = time.time()
    proc = psutil.Popen(["python", name])

    peak_mem = 0
    cpu_percent = []

    try:
        while proc.is_running():
            mem = proc.memory_info().rss / (1024**2)
            peak_mem = max(peak_mem, mem)
            cpu = proc.cpu_percent(interval=0.1)
            cpu_percent.append(cpu)
    except Exception:
        pass

    end = time.time()
    duration = round(end - start, 2)
    avg_cpu = round(sum(cpu_percent) / len(cpu_percent), 1) if cpu_percent else 0

    print(f"[boot] ? {name} finished in {duration}s | CPU: {avg_cpu}% | MEM: {int(peak_mem)}MB")
    timings.append((name, duration, avg_cpu, int(peak_mem)))
    return proc.returncode == 0

def log_timings(timings, total):
    with open(LOG_PATH, "a", encoding="utf-8") as log:
        log.write(f"\\n=== BOOT TELEMETRY [{time.strftime('%Y-%m-%d %H:%M:%S')}] ===\\n")
        for name, dur, cpu, mem in timings:
            log.write(f" - {name}: {dur}s | CPU: {cpu}% | MEM: {mem}MB\\n")
        log.write(f"TOTAL BOOT TIME: {round(total, 2)} seconds\\n")

def main():
    print("? LOGICSHREDDER SYSTEM BOOT STARTED")
    print(f"? Platform: {platform.system()} | Python: {platform.python_version()}")
    print("==============================================\\n")

    start_total = time.time()
    timings = []

    for script in SCRIPTS:
        success = run_script(script, timings)
        if not success:
            print(f"[boot] ? Boot aborted due to failure in {script}")
            break

    total_time = time.time() - start_total
    print(f"[OK] BOOT COMPLETE in {round(total_time, 2)} seconds.")
```

```
    log_timings(timings, total_time)

if __name__ == "__main__":
    main()
''')
# ========== nvme_memory_shim.py ==========
write_file(["nvme_memory_shim.py"], '''
import os
import time
import yaml
import psutil
from pathlib import Path
from shutil import disk_usage

BASE = Path(__file__).parent
CONFIG_PATH = BASE / "system_config.yaml"
LOGIC_CACHE = BASE / "hotcache"

def detect_nvmes():
    nvmes = []
    fallback_mounts = ['C', 'D', 'E', 'F']
    for part in psutil.disk_partitions():
        label = part.device.lower()
        try:
            usage = disk_usage(part.mountpoint)
            is_nvme = any(x in label for x in ['nvme', 'ssd'])
            is_fallback = part.mountpoint.strip(':').upper() in fallback_mounts
            if is_nvme or is_fallback:
                nvmes.append({
                    'mount': part.mountpoint,
                    'fstype': part.fstype,
                    'free_gb': round(usage.free / 1e9, 2),
                    'total_gb': round(usage.total / 1e9, 2)
                })
        except Exception:
            continue
    print(f"[shim] Detected {len(nvmes)} logic-capable drive(s): {[n['mount'] for n in nvmes]}")
    return sorted(nvmes, key=lambda d: d['free_gb'], reverse=True)

def assign_as_logic_ram(nvmes):
    logic_zones = {}
    for i, nvme in enumerate(nvmes[:4]):
        zone = f"ram_shard_{i+1}"
        path = Path(nvme['mount']) / "logicshred_cache"
        path.mkdir(exist_ok=True)
        logic_zones[zone] = str(path)
    return logic_zones

def update_config(zones):
    if CONFIG_PATH.exists():
        with open(CONFIG_PATH, 'r') as f:
            config = yaml.safe_load(f)
    else:
        config = {}
```

```
    config['logic_ram'] = zones
    config['hotcache_path'] = str(LOGIC_CACHE)
    with open(CONFIG_PATH, 'w') as f:
        yaml.safe_dump(config, f)
    print(f"[OK] Config updated with NVMe logic cache: {list(zones.values())}")

if __name__ == "__main__":
    LOGIC_CACHE.mkdir(exist_ok=True)
    print("[INFO] Detecting NVMe drives and logic RAM mounts...")
    drives = detect_nvmes()
    if not drives:
        print("[WARN] No NVMe or fallback drives detected. System unchanged.")
    else:
        zones = assign_as_logic_ram(drives)
        update_config(zones)
''')

# ========== fragment_teleporter.py ==========
write_file(["fragment_teleporter.py"], '''
import shutil
from pathlib import Path

CORE_DIR = Path("fragments/core")
TARGETS = [Path("fragments/node1"), Path("fragments/node2")]
TRANSFER_LOG = Path("logs/teleport_log.txt")
TRANSFER_LOG.parent.mkdir(parents=True, exist_ok=True)

for target in TARGETS:
    target.mkdir(parents=True, exist_ok=True)

class FragmentTeleporter:
    def __init__(self, limit=5):
        self.limit = limit

    def select_fragments(self):
        frags = list(CORE_DIR.glob("*.yaml"))
        return frags[:self.limit] if frags else []

    def teleport(self):
        selections = self.select_fragments()
        for i, frag_path in enumerate(selections):
            target = TARGETS[i % len(TARGETS)] / frag_path.name
            shutil.move(str(frag_path), target)
            with open(TRANSFER_LOG, 'a') as log:
                log.write(f"[TELEPORTED] {frag_path.name} -> {target}\\n")
            print(f"[Teleporter] {frag_path.name} -> {target}")

if __name__ == "__main__":
    teleporter = FragmentTeleporter(limit=10)
    teleporter.teleport()
''')

# ========== context_activator.py ==========
write_file(["context_activator.py"], '''
```

```
import yaml
from pathlib import Path

FRAGMENTS_DIR = Path("fragments/core")
ACTIVATION_LOG = Path("logs/context_activation.log")
ACTIVATION_LOG.parent.mkdir(parents=True, exist_ok=True)

class ContextActivator:
    def __init__(self, activation_threshold=0.75):
        self.threshold = activation_threshold

    def scan_fragments(self):
        activated = []
        for frag_file in FRAGMENTS_DIR.glob("*.yaml"):
            try:
                with open(frag_file, 'r') as f:
                    frag = yaml.safe_load(f)
                if frag.get("confidence", 0.5) >= self.threshold:
                    activated.append(frag)
            except Exception as e:
                print(f"Error reading {frag_file.name}: {e}")
        return activated

    def log_activations(self, activations):
        with open(ACTIVATION_LOG, 'a') as log:
            for frag in activations:
                log.write(f"[ACTIVATED] {frag['id']} :: {frag.get('claim', '???')}\\n")
        print(f"[ContextActivator] {len(activations)} fragment(s) activated.")

    def run(self):
        active = self.scan_fragments()
        self.log_activations(active)

if __name__ == "__main__":
    ctx = ContextActivator()
    ctx.run()
''')

# ========== subcon_layer_mapper.py ==========
write_file(["subcon_layer_mapper.py"], '''
import os
import yaml
from pathlib import Path

LAYER_MAP_PATH = Path("subcon_map.yaml")
FRAGMENTS_DIR = Path("fragments/core")
OUTPUT_PATH = Path("meta/subcon_layer_cache.yaml")
OUTPUT_PATH.parent.mkdir(parents=True, exist_ok=True)

class SubconLayerMapper:
    def __init__(self):
        self.layer_map = self.load_map()

    def load_map(self):
```

```
        if not LAYER_MAP_PATH.exists():
            print("[Mapper] No layer map found. Returning empty.")
            return {}
        with open(LAYER_MAP_PATH, 'r') as f:
            return yaml.safe_load(f)

    def extract_links(self):
        results = {}
        for file in FRAGMENTS_DIR.glob("*.yaml"):
            try:
                with open(file, 'r') as f:
                    frag = yaml.safe_load(f)
                tags = frag.get("tags", [])
                for tag in tags:
                    if tag in self.layer_map:
                        results.setdefault(tag, []).append(frag['id'])
            except Exception as e:
                print(f"[Mapper] Failed to read {file.name}: {e}")
        return results

    def save_cache(self, data):
        with open(OUTPUT_PATH, 'w') as out:
            yaml.dump(data, out)
        print(f"[Mapper] Saved subcon layer associations -> {OUTPUT_PATH}")

    def run(self):
        links = self.extract_links()
        self.save_cache(links)

if __name__ == "__main__":
    mapper = SubconLayerMapper()
    mapper.run()
''')
# ========== validator.py ==========
write_file(["validator.py"], '''
import os
import time
from pathlib import Path
import yaml
from core.utils import load_yaml, hash_string, validate_fragment
from core.cortex_bus import send_message

CORE_DIR = Path("fragments/core")
OVERFLOW_DIR = Path("fragments/overflow")
OVERFLOW_DIR.mkdir(parents=True, exist_ok=True)

class Validator:
    def __init__(self, agent_id="validator_01"):
        self.agent_id = agent_id
        self.frags = {}

    def load_core_beliefs(self):
        for path in CORE_DIR.glob("*.yaml"):
            frag = load_yaml(path, validate_schema=validate_fragment)
```

```
            if frag:
                claim_hash = hash_string(frag['claim'])
                self.frags[claim_hash] = (path, frag)

    def contradicts(self, a, b):
        return a.lower().strip() == f"not {b.lower().strip()}"

    def run_validation(self):
        for hash_a, (path_a, frag_a) in self.frags.items():
            for hash_b, (path_b, frag_b) in self.frags.items():
                if hash_a == hash_b:
                    continue
                if self.contradicts(frag_a['claim'], frag_b['claim']):
                    contradiction_id = f"{hash_a[:6]}_{hash_b[:6]}"
                    filename = f"contradiction_{contradiction_id}.yaml"
                    contradiction_path = OVERFLOW_DIR / filename
                    if not contradiction_path.exists():
                        contradiction = {
                            'source_1': frag_a['claim'],
                            'source_2': frag_b['claim'],
                            'path_1': str(path_a),
                            'path_2': str(path_b),
                            'detected_by': self.agent_id,
                            'timestamp': int(time.time())
                        }
                        with open(contradiction_path, 'w') as out:
                            yaml.dump(contradiction, out)
                        send_message({
                            'from': self.agent_id,
                            'type': 'contradiction_found',
                            'payload': {
                                'claim_1': frag_a['claim'],
                                'claim_2': frag_b['claim'],
                                'paths': [str(path_a), str(path_b)]
                            },
                            'timestamp': int(time.time())
                        })

    def run(self):
        self.load_core_beliefs()
        self.run_validation()

if __name__ == "__main__":
    Validator().run()
''')

# ========== inject_profiler.py ==========
write_file(["inject_profiler.py"], '''
import time
import psutil

class InjectProfiler:
    def __init__(self, label="logic_injection"):
        self.label = label
```

```
        self.snapshots = []

    def snapshot(self):
        mem = psutil.virtual_memory().percent
        cpu = psutil.cpu_percent(interval=0.1)
        self.snapshots.append((time.time(), mem, cpu))

    def report(self):
        print(f"[Profiler:{self.label}] Total snapshots: {len(self.snapshots)}")
        for t, mem, cpu in self.snapshots:
            print(f" - {round(t, 2)}s :: MEM {mem}% | CPU {cpu}%")

if __name__ == "__main__":
    p = InjectProfiler()
    for _ in range(5):
        p.snapshot()
        time.sleep(1)
    p.report()
''')

# ========== async_swarm_launcher.py ==========
write_file(["async_swarm_launcher.py"], '''
import asyncio
import subprocess

TASKS = [
    "fragment_decay_engine.py",
    "dreamwalker.py",
    "validator.py",
    "mutation_engine.py"
]

async def run_script(name):
    proc = await asyncio.create_subprocess_exec("python", name)
    await proc.wait()

async def main():
    coros = [run_script(task) for task in TASKS]
    await asyncio.gather(*coros)

if __name__ == "__main__":
    asyncio.run(main())
''')

# ========== requirements_installer.py ==========
write_file(["requirements_installer.py"], '''
required = [
    "torch",
    "numpy",
    "ray",
    "optuna",
    "matplotlib",
    "psutil",
    "pyyaml"
```

```
]

def install_all():
    import subprocess
    for r in required:
        print(f"[INSTALL] Installing: {r}")
        subprocess.run(["pip", "install", r])

if __name__ == "__main__":
    install_all()
''')

==== redis_publisher.py ====
# redis_publisher.py
import redis
import time

CHANNEL = "symbolic_channel"
client = redis.Redis(host='127.0.0.1', port=6379)

def publish_loop():
    i = 0
    while True:
        message = f"? Belief-{i} has entered the bus."
        client.publish(CHANNEL, message)
        print(f"[PUB] {message}")
        time.sleep(2)
        i += 1

if __name__ == "__main__":
    print(f"? Publishing on channel: {CHANNEL}")
    publish_loop()

==== redis_subscriber.py ====
# redis_subscriber.py
import redis

CHANNEL = "symbolic_channel"
client = redis.Redis(host='127.0.0.1', port=6379)
pubsub = client.pubsub()
pubsub.subscribe(CHANNEL)

print(f"???? Listening on channel: {CHANNEL}")
for message in pubsub.listen():
    if message['type'] == 'message':
        print(f"[SUB] {message['data'].decode('utf-8')}")

==== regex_to_grammar.py ====
import json, subprocess, sys, os

assert len(sys.argv) >= 2
[_, pattern, *rest] = sys.argv

print(subprocess.check_output(
```

```
    [
        "python",
        os.path.join(
        os.path.dirname(os.path.realpath(__file__)),
        "json_schema_to_grammar.py"),
        *rest,
        "-",
        "--raw-pattern",
    ],
    text=True,
    input=json.dumps({
        "type": "string",
        "pattern": pattern,
    }, indent=2)))

==== requirements.py ====
tensorflow
keras
numpy
pandas
sklearn
matplotlib
seaborn

==== requirements_installer.py ====
required = [
    "torch",
    "numpy",
    "ray",
    "optuna",
    "matplotlib",
    "psutil",
    "pyyaml"
]

def install_all():
    import subprocess
    for r in required:
        print(f"[INSTALL] Installing: {r}")
        subprocess.run(["pip", "install", r])

if __name__ == "__main__":
    install_all()

==== rpc_parameter_server.py ====
import argparse
import logging
import os
import sys
import time
from threading import Lock

import torch
import torch.distributed.autograd as dist_autograd
```

```
import torch.distributed.rpc as rpc
import torch.multiprocessing as mp
import torch.nn as nn
import torch.nn.functional as F
from torch import optim
from torch.distributed.optim import DistributedOptimizer
from torchvision import datasets, transforms

logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)
logger = logging.getLogger()

# Constants
TRAINER_LOG_INTERVAL = 5  # How frequently to log information
TERMINATE_AT_ITER = 300  # for early stopping when debugging
PS_AVERAGE_EVERY_N = 25  # How often to average models between trainers

# --------- MNIST Network to train, from pytorch/examples -----


class Net(nn.Module):
    def __init__(self, num_gpus=0):
        super(Net, self).__init__()
        logger.info(f"Using {num_gpus} GPUs to train")
        self.num_gpus = num_gpus
        device = torch.device(
            "cuda:0" if torch.cuda.is_available() and self.num_gpus > 0 else "cpu"
        )
        logger.info(f"Putting first 2 convs on {str(device)}")
        # Put conv layers on the first cuda device
        self.conv1 = nn.Conv2d(1, 32, 3, 1).to(device)
        self.conv2 = nn.Conv2d(32, 64, 3, 1).to(device)
        # Put rest of the network on the 2nd cuda device, if there is one
        if "cuda" in str(device) and num_gpus > 1:
            device = torch.device("cuda:1")

        logger.info(f"Putting rest of layers on {str(device)}")
        self.dropout1 = nn.Dropout2d(0.25).to(device)
        self.dropout2 = nn.Dropout2d(0.5).to(device)
        self.fc1 = nn.Linear(9216, 128).to(device)
        self.fc2 = nn.Linear(128, 10).to(device)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.max_pool2d(x, 2)

        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        # Move tensor to next device if necessary
        next_device = next(self.fc1.parameters()).device
        x = x.to(next_device)

        x = self.fc1(x)
```