```
import asyncio
import time
from typing import Any, Callable, List, Tuple

import ray
import torch
from loguru import logger
from ray import serve

# Initialize Ray and Ray Serve
ray.init(ignore_reinit_error=True)
serve.start(detached=True)


# GPU Utility Function
def detect_gpus() -> List[str]:
    """
    Detects available GPU IDs on the system using torch.cuda.

    Returns:
        List[str]: A list of detected GPU IDs.
    """
    gpu_ids = [f"gpu-{i}" for i in range(torch.cuda.device_count())]
    logger.info(f"Detected GPUs: {gpu_ids}")
    return gpu_ids
```

```
@serve.deployment(route_prefix="/execute_job", num_replicas=2)
class GPUJobExecutor:
    def __init__(self):
        """Initialize GPUJobExecutor with a dynamic GPU pool and intelligent scheduler."""
        self.gpu_pool = detect_gpus()
        self.available_gpus = self.gpu_pool.copy()
        self.health_status = {
            gpu: True for gpu in self.gpu_pool
        }  # GPU health tracking
        logger.info(f"Initialized GPU pool: {self.gpu_pool}")
        asyncio.create_task(self.monitor_gpu_health())

    async def monitor_gpu_health(self):
        """
        Monitors GPU health periodically and marks GPUs as unavailable if issues are detected.
        """
        while True:
            for gpu in self.gpu_pool:
                if self._check_gpu_health(gpu):
                    self.health_status[gpu] = True
                else:
                    logger.warning(f"GPU {gpu} marked unhealthy")
                    self.health_status[gpu] = False
            await asyncio.sleep(5)  # Check every 5 seconds
```

```
def _check_gpu_health(self, gpu_id: str) -> bool:
    """
    Checks the health of a GPU based on simple metrics.
    This could be extended with more sophisticated monitoring in production.

    Args:
        gpu_id (str): GPU ID to check.

    Returns:
        bool: True if healthy, False if not.
    """
    gpu_index = int(gpu_id.split("-")[-1])
    memory_used = torch.cuda.memory_allocated(gpu_index)
    memory_limit = torch.cuda.get_device_properties(
        gpu_index
    ).total_memory
    # Basic health check: GPU is considered unhealthy if memory usage exceeds 90%
    return memory_used < 0.9 * memory_limit

async def execute(
    self,
    job_fn: Callable[[], Any],
    priority: int,
    *args,
    **kwargs,
) -> Any:
```

```
"""
Executes a job function with an available GPU in the pool.

Args:
    job_fn (Callable): The job function to execute.
    priority (int): Priority level for the job (1 is highest priority).
    *args: Positional arguments for the job function.
    **kwargs: Keyword arguments for the job function.

Returns:
    Any: The result of the job function execution.
"""
if not self.available_gpus:
    logger.warning(
        "No GPUs available. Job will wait for an available GPU."
    )
    await self._wait_for_available_gpu()

gpu_id = self._allocate_gpu(priority)
logger.info(
    f"Allocating {gpu_id} for job execution with priority {priority}."
)

try:
    result = await job_fn(*args, **kwargs)
    logger.info(f"Job completed successfully on {gpu_id}.")
```

```
        return result
    except Exception as e:
        logger.error(
            f"Error during job execution on {gpu_id}: {e}"
        )
        raise
    finally:
        self._release_gpu(gpu_id)

async def _wait_for_available_gpu(self):
    """
    Waits until a GPU becomes available.
    """
    while not self.available_gpus:
        logger.info("Waiting for an available GPU...")
        await asyncio.sleep(1)

def _allocate_gpu(self, priority: int) -> str:
    """
    Allocates the first healthy GPU from the pool with priority handling.

    Returns:
        str: The allocated GPU ID.
    """
    healthy_gpus = [
        gpu
        for gpu in self.available_gpus
        if self.health_status[gpu]
    ]
    if healthy_gpus:
        gpu_id = healthy_gpus.pop(0)
        self.available_gpus.remove(gpu_id)
        logger.info(
            f"{gpu_id} allocated with priority {priority}."
        )
        return gpu_id
    else:
        logger.warning(
            "No healthy GPUs available; waiting for recovery."
        )
        time.sleep(1)
        return self._allocate_gpu(priority)


def _release_gpu(self, gpu_id: str):
    """
    Releases a GPU back to the pool.


    Args:
        gpu_id (str): The GPU ID to release.
    """
    self.available_gpus.append(gpu_id)
    logger.info(f"{gpu_id} released back to pool.")
    async def __call__(
        self,
        job_fn: Callable[[], Any],
        priority: int = 1,
        *args,
        **kwargs,
    ) -> Any:
        return await self.execute(job_fn, priority, *args, **kwargs)


@serve.deployment
class JobScheduler:
    def __init__(self, executor_handle: Any):
        """
        JobScheduler to manage job execution with fault tolerance, job retries, and scaling.

        Args:
            executor_handle (RayServeHandle): Handle to GPUJobExecutor deployment.
        """
        self.executor_handle = executor_handle
        self.max_retries = 3
        self.job_queue: List[Tuple[int, Callable]] = (
            []
        )  # Prioritized job queue
        self.min_executors = 2  # Minimum number of executors running
    self.max_executors = 6  # Maximum number of executors allowed
    asyncio.create_task(self.scale_cluster())

async def schedule_job(
    self,
    job_fn: Callable[[], Any],
    priority: int = 1,
    *args,
    **kwargs,
) -> Any:
    """
    Schedule and execute a job with retries in case of failure.

    Args:
        job_fn (Callable): The job function to execute.
        priority (int): Priority level for the job (1 is highest priority).
        *args: Positional arguments for the job function.
        **kwargs: Keyword arguments for the job function.

    Returns:
        Any: The result of the job function execution.
    """
    self.job_queue.append((priority, job_fn))
    self.job_queue.sort(key=lambda x: x[0])  # Sort by priority

    for attempt in range(self.max_retries):
            try:
                _, next_job = self.job_queue.pop(0)
                result = await self.executor_handle.execute.remote(
                    next_job, priority, *args, **kwargs
                )
                logger.info(
                    f"Job completed successfully on attempt {attempt + 1}"
                )
                return result
            except Exception as e:
                logger.warning(
                    f"Job failed on attempt {attempt + 1}: {e}"
                )
                if attempt == self.max_retries - 1:
                    logger.error(
                        "Max retries reached. Job failed permanently."
                    )
                    raise
                await asyncio.sleep(2**attempt)  # Exponential backoff
                logger.info(
                    f"Retrying job execution, attempt {attempt + 2}"
                )

    async def scale_cluster(self):
        """
        Dynamically scales GPU nodes in the cluster based on load.
```

```
"""
while True:
    queue_size = len(self.job_queue)
    num_executors = len(
        await self.executor_handle.list_replicas.remote()
    )

    if queue_size > 5 and num_executors < self.max_executors:
        logger.info("Scaling up resources for high load.")
        new_executor = (
            GPUJobExecutor.bind()
        )  # Create a new executor
        ray.get(
            new_executor.deploy()
        )  # Deploy additional executor
    elif (
        queue_size == 0 and num_executors > self.min_executors
    ):
        logger.info("Scaling down resources for low load.")
        await self.executor_handle.remove_replicas.remote(
            1
        )  # Remove an executor
    await asyncio.sleep(
        10
    )  # Adjust scaling check frequency as needed
```

```python
# # Deployment
# gpu_executor = GPUJobExecutor.bind()
# job_scheduler = JobScheduler.bind(gpu_executor)

# # Example job function
# async def example_job_fn():
#     await asyncio.sleep(2)  # Simulates job duration
#     return "Job completed"

# # Usage: Send request to job scheduler
# async def run():
#     job_handle = job_scheduler.schedule_job.remote(example_job_fn, priority=1)
#     result = await ray.get(job_handle)
#     print(result)

# # Deployments are initialized via Ray Serve:
# serve.run(run)



def gpu_scheduler(
    function: Callable[[], Any], priority: int = 1, *args, **kwargs
) -> Any:
    """
    Schedule and execute a function on available GPUs using Ray and Ray Serve.
    Args:
        function (Callable[[], Any]): The function to execute on GPU
          priority (int, optional): Priority level for job scheduling. Lower numbers indicate higher priority. Defaults to 1.
        *args: Additional positional arguments to pass to the function
        **kwargs: Additional keyword arguments to pass to the function

    Returns:
        Any: Result of the executed function

    Raises:
        RuntimeError: If there is an error initializing the GPU executor or job scheduler
        TimeoutError: If the job execution times out
        Exception: If there is an error executing the function
    """
    try:
        # Initialize GPU executor and job scheduler
        gpu_executor = GPUJobExecutor.bind()
        job_scheduler = JobScheduler.bind(gpu_executor)

        # Schedule and execute the job
        job_handle = job_scheduler.schedule_job.remote(
            function, priority, *args, **kwargs
        )
        result = ray.get(job_handle, timeout=3600)  # 1 hour timeout
        logger.info(
            f"Job completed successfully with result: {result}"
        )
        return result

    except TimeoutError as e:
        logger.error(f"Job execution timed out: {e}")
        raise TimeoutError(
            "GPU job execution timed out after 1 hour"
        ) from e

    except Exception as e:
        logger.error(f"Error executing GPU job: {e}")
        raise RuntimeError(
            f"Failed to execute GPU job: {str(e)}"
        ) from e
```