PR #44428 完整报告

vllm-project/vllm

[Feature] Add fault tolerance framework (simplified) for DP+EP external LB deployments

合并时间：2026-07-25 23:49

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/44428

执行摘要

- 一句话：为 DP+EP MoE 部署添加容错框架（简化版）
- 推荐动作：值得精读，尤其是 sentinel 模式将 FT 状态与主逻辑分离、以及外部驱动恢复的设计。建议关注认证绕过和状态广播问题的跟踪解决。生产部署前必须加固 API 认证。对于仅使用 vLLM 标准部署的用户，此 PR 无直接影响。

功能与动机

当 DP rank 死亡时，EP all2all 操作在幸存 rank 上无限阻塞，导致整个集群无响应。此 PR 旨在检测故障、中止请求并允许外部编排器通过 REST API 触发协调恢复。这是简化版本，移除了冗余抽象和未使用的配置字段（PR body）。

实现拆解

1. 新增 sentinel 类：在 vllm/v1/fault_tolerance/engine_core_sentinel.py 中创建 EngineCoreSentinel，管理 EngineCore 的 FT 状态（healthy/unhealthy/dead）和恢复命令调度；在 vllm/v1/worker/sentinel/gpu_worker_sentinel.py 中创建 WorkerSentinel，管理 Worker 侧的 FT 状态和恢复命令执行。两个 sentinel 通过 run_method 反射调度指令（如 retry）。
2. 新增配置和 CLI 参数：在 vllm/config/fault_tolerance.py 中定义 FaultToleranceConfig（含 engine_recovery_timeout_sec）；在 vllm/engine/arg_utils.py 中添加 --enable-fault-tolerance 和 --fault-tolerance-config 参数，并在 ParallelConfig.__post_init__ 中增加校验（外部 LB 模式等）。
3. 新增 REST API 端点：在 vllm/entrypoints/serve/fault_tolerance/api_router.py 中注册两个端点：POST /fault_tolerance/apply（接收恢复指令，立即返回 202，后台执行 _run_fault_recovery）和 GET /fault_tolerance/status（返回各引擎状态缓存）。API 仅在启用了 FT 时注册。
4. 修改 engine client 和 protocol：在 vllm/engine/protocol.py 和 vllm/v1/engine/core_client.py 中增加 handle_fault 和 get_status 抽象方法和具体实现。handle_fault 通过 call_utility_async 发送 FT 命令到 EngineCore，get_status 从引擎状态缓存读取。同时修改 async_llm.py 适配多 client 场景。
5. 修改 Worker 执行流：在 vllm/v1/worker/gpu/model_runner.py 中添加 check_ep_fault 检测；在 vllm/distributed/device_communicators/all2all.py 中为 DeepEP 和 Nixl 后端添

加 clean_buffers 方法；在 vllm/v1/worker/gpu_worker.py 中集成 WorkerSentinel。

6. E2E 测试和 CI：新增 tests/v1/fault_tolerance/test_fault_tolerance_e2e.py，使用 sitecustomize.py 注入故障到 DP 同步函数，验证故障检测和 retry 恢复流程。添加 .buildkite/test_areas/fault_tolerance.yaml，将测试接入 CI。

关键文件：

- vllm/v1/fault_tolerance/engine_core_sentinel.py（模块 容错核心；类别 source；类型 dependency-wiring；符号 EngineCoreSentinel, init, handle_command, on_fault）：核心 FT 状态机：管理 EngineCore 的健康状态、故障处理、恢复命令调度。提供了 fault_tolerant_wrapper 装饰器，是热路径的唯一集成点。
- vllm/entrypoints/serve/fault_tolerance/api_router.py（模块 API 入口；类别 source；类型 entrypoint；符号 _validate_payload, process_fault_tolerance_instruction, _run_fault_recovery, get_status）：FT REST API 入口：定义 POST /fault_tolerance/apply 和 GET /fault_tolerance/status 端点，实现验证、调度和状态查询。
- vllm/v1/worker/sentinel/gpu_worker_sentinel.py（模块 Worker 状态；类别 source；类型 dependency-wiring；符号 WorkerSentinel, init, handle_command, retry）：Worker 侧 FT 实现：管理 DP 配置、清理 worker 状态、重新初始化 CPU 组，支持 retry 恢复指令。
- tests/v1/fault_tolerance/test_fault_tolerance_e2e.py（模块 E2E 测试；类别 test；类型 test-coverage；符号 _patch, _wrapped, _hook, _install_fault_injection）：端到端验收测试：通过 sitecustomize.py 注入故障，验证故障检测和 retry 恢复流程，确保框架基本可用性。
- vllm/v1/engine/core_client.py（模块 客户端协议；类别 source；类型 dependency-wiring；符号 handle_fault, get_status）：修改了客户端协议：新增 handle_fault 和 get_status 抽象方法，以及在 AsyncMPClient 中实现 FT 命令发送和状态缓存更新。
- vllm/v1/fault_tolerance/utils.py（模块 数据模型；类别 source；类型 core-logic；符号 FaultToleranceResult, FaultToleranceRequest）：定义 FT 核心数据模型：FaultToleranceResult 和 FaultToleranceRequest，用于序列化 / 反序列化 FT 命令和结果。
- vllm/config/fault_tolerance.py（模块 配置；类别 source；类型 core-logic；符号 FaultToleranceConfig）：FT 配置类定义：FaultToleranceConfig，当前包含 engine_recovery_timeout_sec 字段。
- vllm/engine/arg_utils.py（模块 CLI 参数；类别 source；类型 core-logic）：添加 CLI 参数和配置校验：--enable-fault-tolerance, --fault-tolerance-config，并在 create_engine_config 中增加外部 LB 模式校验。

关键符号：EngineCoreSentinel.handle_command, EngineCoreSentinel.on_fault, EngineCoreSentinel.retry, WorkerSentinel.retry, WorkerSentinel._clean_worker_state, api_router.process_fault_tolerance_instruction, api_router._run_fault_recovery, api_router.get_status, core_client.AsyncMPClient.handle_fault, core_client.AsyncMPClient.get_status

关键源码片段

vllm/entrypoints/serve/fault_tolerance/api_router.py

FT REST API 入口：定义 POST /fault_tolerance/apply 和 GET /fault_tolerance/status 端点，实现验证、调度和状态查询。

```
# vllm/entrypoints/serve/fault_tolerance/api_router.py
import json
import uuid
from http import HTTPStatus
from fastapi import APIRouter, BackgroundTasks, Depends, FastAPI, HTTPException, Request
from fastapi.responses import JSONResponse
from vllm.engine.protocol import EngineClient
from vllm.entrypoints.openai.engine.protocol import ErrorResponse
from vllm.entrypoints.serve.utils.api_utils import validate_json_request
from vllm.logger import init_logger
from vllm.v1.fault_tolerance.utils import FaultToleranceRequest

logger = init_logger(__name__)
router = APIRouter()
_ALLOWED_INSTRUCTIONS = {"retry"}

def _validate_payload(body: dict) -> tuple[str, dict]:
    """验证请求体：必需 'instruction' 字段，可选 'params' 字段。"""
    if not isinstance(body, dict):
        raise HTTPException(400, "Request body must be a JSON object.")
    instruction = body.get("instruction")
    if not instruction:
        raise HTTPException(400, "'instruction' is required.")
    if instruction not in _ALLOWED_INSTRUCTIONS:
        raise HTTPException(400, f"Invalid instruction: '{instruction}'.")
    params = body.get("params", {})
    if not isinstance(params, dict):
        raise HTTPException(400, "'params' must be an object.")
    return instruction, params

@router.post("/fault_tolerance/apply", dependencies=[Depends(validate_json_request)],
             responses={HTTPStatus.ACCEPTED.value: {"model": dict}, HTTPStatus.BAD_REQUEST.
             value: {"model": ErrorResponse}})
async def process_fault_tolerance_instruction(raw_request: Request, background_tasks:
BackgroundTasks):
    """接收恢复指令，立即返回202接受，后台异步执行恢复。"""
    try:
        body = await raw_request.json()
    except json.JSONDecodeError as e:
        raise HTTPException(400, "Invalid JSON format") from e
    instruction, params = _validate_payload(body)
    ft_request = FaultToleranceRequest(instruction=instruction, params=params, request_id=
    str(uuid.uuid4()))
    client: EngineClient = raw_request.app.state.engine_client
```

```
    # 后台运行恢复，避免阻塞编排器对其他 rank 的调度
    background_tasks.add_task(_run_fault_recovery, client, ft_request)
    return JSONResponse(
        status_code=HTTPStatus.ACCEPTED.value,
        content={"message": "Request accepted; poll /fault_tolerance/status for updates.",
        "request_id": ft_request.request_id},
        background=background_tasks,
    )

async def _run_fault_recovery(client: EngineClient, ft_request: FaultToleranceRequest) -> None:
    """执行恢复，并发回202后异步完成。"""
    try:
        result = await client.handle_fault(ft_request)
    except Exception:
        logger.exception("[FT] Recovery dispatch failed.")
        return
    if not result.success:
        logger.error("[FT] Recovery failed for request %s: %s", ft_request.request_id, result.
        reason)

@router.get("/fault_tolerance/status")
async def get_status(raw_request: Request):
    """返回所有引擎的健康状态。"""
    client: EngineClient = raw_request.app.state.engine_client
    return JSONResponse(content=await client.get_status())

def register_fault_tolerance_api_router(app: FastAPI):
    """仅在启用FT时注册该路由。"""
    app.include_router(router)
```

vllm/v1/worker/sentinel/gpu_worker_sentinel.py

Worker 侧 FT 实现：管理 DP 配置、清理 worker 状态、重新初始化 CPU 组，支持 retry 恢复指令。

```
# vllm/v1/worker/sentinel/gpu_worker_sentinel.py
from typing import TYPE_CHECKING, cast
import torch
from vllm.config import set_current_vllm_config
from vllm.distributed import get_dp_group, stateless_destroy_torch_distributed_process_group,
stateless_init_torch_distributed_process_group
from vllm.logger import init_logger
from vllm.model_executor.layers.fused_moe.all2all_utils import get_ep_all2all_manager
from vllm.v1.fault_tolerance.utils import FaultToleranceRequest
from vllm.v1.serial_utils import run_method

if TYPE_CHECKING:
    from vllm.v1.worker.gpu.model_runner import GPUModelRunner as GPUModelRunnerV2
    from vllm.v1.worker.gpu_worker import Worker
```

```
logger = init_logger(__name__)
# 支持 FT 的 all2all 后端集合（需要超时 + rank 屏蔽能力）
FT_BACKEND_SET = frozenset({"deepep_low_latency", "nixl_ep"})

class WorkerSentinel:
    """管理单个worker的FT状态（mask张量、DP配置）。
    方法通过collective_rpc从EngineCoreSentinel调用。
    """
    def __init__(self, worker: "Worker"):
        self.worker = worker
        self.dp_rank = worker.parallel_config.data_parallel_rank
        self.dp_size = worker.parallel_config.data_parallel_size
        self.data_parallel_master_ip = worker.parallel_config.data_parallel_master_ip
        all2all_backend = worker.parallel_config.all2all_backend
        # 初始化时校验后端是否支持 FT
        if all2all_backend not in FT_BACKEND_SET:
            raise ValueError(f"Fault tolerance requires an FT-capable all2all backend (one of
            {sorted(FT_BACKEND_SET)}), but got '{all2all_backend}'.")

    def handle_command(self, ft_request: FaultToleranceRequest):
        """根据指令名称调度。"""
        with set_current_vllm_config(self.worker.vllm_config):
            return run_method(self, ft_request.instruction, (ft_request,), {})

    def retry(self, ft_request: FaultToleranceRequest):
        """执行恢复：同步GPU、清理worker状态、重置EP all2all缓冲区、重新初始化DP CPU组。"""
        torch.accelerator.synchronize()
        params = ft_request.params
        self._clean_worker_state()
        if self.dp_size > 1:
            get_ep_all2all_manager().clean_buffers()
            old_cpu_group = get_dp_group().cpu_group
            stateless_destroy_torch_distributed_process_group(old_cpu_group)
            world_size = self.worker.parallel_config.world_size
            port = params["new_stateless_dp_group_ports"][self.worker.rank % world_size]
            get_dp_group().cpu_group = stateless_init_torch_distributed_process_group(
                self.data_parallel_master_ip, port, self.dp_rank, self.dp_size, backend="gloo"
            )

    def _clean_worker_state(self):
        """清理模型运行时的中间状态，准备重新开始服务。"""
        model_runner = self.worker.model_runner
        model_runner.execute_model_state = None
        if self.worker.use_v2_model_runner:
            runner = cast("GPUModelRunnerV2", model_runner)
            for req_id in list(runner.req_states.req_id_to_index):
                runner._remove_request(req_id)
        else:
            model_runner.kv_connector_output = None
```

```
input_batch = model_runner.input_batch
cached_req_ids = list(input_batch.req_id_to_index)
for req_id in cached_req_ids:
    model_runner.requests.pop(req_id, None)
    model_runner.num_prompt_logprobs.pop(req_id, None)
    input_batch.remove_request(req_id)
input_batch.condense()
input_batch.refresh_metadata()
input_batch.req_prompt_embeds.clear()
```

评论区精华

1. 认证绕过风险（depthfirst-app[bot] 指出）：/fault_tolerance/apply 和 /fault_tolerance/status 端点未受 AuthenticationMiddleware 保护（GUARDED_PREFIX 仅覆盖 /v1、/v2、/inference）。配置 API key 后这些端点仍可无认证访问，未授权攻击者可能触发集群恢复。
2. 状态广播只到 client 0（tzulingk 指出）：_push_status 硬编码 client_idx 为 0，导致多 client 场景下其他 client 无法收到状态更新。作者回复说当前外部 LB 默认 client_count=1，多 client 作为遗留问题。
3. 恢复流程应为异步（tlrmchlsmth 提出）：初始实现 POST /fault_tolerance/apply 同步等待恢复完成，但跨 rank 集体操作要求编排器并发发送 POST。修改后返回 202 Accepted，编排器通过轮询 /fault_tolerance/status 观察结果。
4. 故障模拟的现实性（tzulingk 质疑）：测试仅在软件层面注入异常（allreduce 后 raise），未模拟真正的进程死亡。作者承认这是简化版，预期用在瞬态故障场景；永久 rank 死亡需后续 PR 实现 scale-down（#46370）。
5. MTTR 可观测性（tlrmchlsmth 提出）：需要 Prometheus 指标记录故障次数和恢复状态，作为后续工作。
6. run_method 任意方法调用风险（depthfirst-app[bot]）：EngineCoreSentinel.handle_command 通过 run_method 反射调用指令方法，但本地无 allowlist，依赖 HTTP 层的 _ALLOWED_INSTRUCTIONS 限制，结合认证绕过可能造成任意方法调用。

- FT 端点认证绕过 (security): 已识别为风险，但未在本次 PR 中修复。需要后续将 FT 端点纳入 GUARDED_PREFIX 或增加独立认证。
- 状态推送硬编码 client 0 (correctness): 已添加 api_server_count=1 的校验，多 client 拓扑目前不支持。
- 恢复流程应由同步改为异步 (design): 已修改为异步：POST 返回 202，后台执行 _run_fault_recovery，编排器轮询 /fault_tolerance/status。
- 故障模拟是否足够现实 (testing): 当前设计暂不考虑进程级故障，retry 仅针对瞬态故障。scale-down 路径在 #46370 中实现。
- MTTR 可观测性 (design): 作为后续工作，本次 PR 未添加。
- run_method 任意方法调用风险 (security): 已添加本地校验：handle_command 中检查指令是否在 _ALLOWED_INSTRUCTIONS 中？未明确看到。但实际上指令通过 run_method 执行，而 run_method 内部使用 getattr，所以仍然存在反射风险。HTTP 层已有校验，但

后续若引入新 RPC 通道可能绕过。建议在 sentinel 内也增加 allowlist。

风险与影响

- 风险：
  1. 认证绕过（高危）：FT 端点未受认证中间件保护，若生产环境配置了 API key，攻击者可滥用 /fault_tolerance/apply 发起空转恢复，中断所有请求。需要将 FT 端点纳入认证范围或增加独立认证。
  2. 状态广播局限：_push_status 硬编码 client 0，多 API server/ 前端部署时其他 client 无法更新健康缓存，可能提供过时状态。当前仅适用于单 client 拓扑。
  3. 故障模拟不完全：E2E 测试仅注入软件异常，未覆盖真正的进程死亡（如 kill -9）。幸存 rank 依赖 Gloo 和 all2all 内核超时检测，超时窗口内行为未充分验证。
  4. retry 幂等性：若 /fault_tolerance/apply 响应丢失，编排器重试时可能对健康的引擎再次执行 retry（该问题已通过在 handle_command 中检查 UNHEALTHY 状态缓解，但 rank 间状态不一致时仍可能误判）。
  5. 配置项耦合：--fault-tolerance-config 传递时会自动启用 --enable-fault-tolerance，可能让用户意外开启 FT。已添加日志警告。 - 影响：用户：仅为使用外部 LB 模式、DP+EP 部署且启用 FT-capable all2all 后端的用户提供价值。用户需增加 FT 配置和编排器集成，获得瞬态故障自动恢复能力。系统：在 EngineCore 和 Worker 中新增 sentinel 对象，热路径增加少量检查（wrapper 装饰器，check_ep_fault），性能影响可忽略。团队：需维护两个核心 sentinel 和 API 层，增加认证和可观测性负担。后续可能有 scale-down PR 叠加。
- 风险标记：认证绕过，状态广播局限，故障模拟不全面，缺少进程级故障测试，run_method 反射风险

关联脉络

- PR #46370 Fault tolerance scale down (follow-up): 作者在讨论中提及此后续 PR，用于处理永久 DP rank 死亡后的 scale-down 恢复策略，与当前 retry 策略互补。
- PR #43202 Elastic EP (deferred): 讨论中提及，Elastic EP 与 FT 正交，但后续可以适应 FT scale-up。
- PR #34833 Earlier PR with FT discussion: 作者引用该 PR 中的讨论作为 auto-enable fault-tolerance 设计的动机。