

PR #56351 完整报告

vllm-project/vllm

[CI][ROCm] Add opt-in TheRock builds for AMD CI

合并时间: 2026-09-16 13:09

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/56351>

执行摘要

- 一句话: AMD CI 新增可选 TheRock 镜像构建路径
- 推荐动作: 值得精读但优先级中等: 核心价值在 .buildkite/scripts/rocm/build-config.sh 的“一个共享选型 + 严格校验 + 三级变量回退”设计, 以及 configure_custom_rocm_stages 与 init_bake_files 中“自定义栈不污染稳定 tag / registry 缓存”的隔离策略, 这是 CI 复用与实验隔离的通用范式, 可直接迁移到其他硬件或镜像变体。建议后续补两块: (a) 为 validate_rocm_dockerfile、using_custom_rocm_dockerfiles 增加 shell 级单测 (本 PR 已主动移除回归测试, 存在回退缺口); (b) 把“自定义配方必须提供的阶段契约”写入文档或脚本头部注释, 减少未来按错误配方配置 CI 的排查成本。

功能与动机

PR body 明确目标是让 AMD CI 能用同一套维护中的 Buildkite jobs 跑 TheRock (AMD 新的 ROCm 打包形态), 而不必修改 Dockerfile 或强推分支: 'Add CI_ROCM_DOCKERFILE_BASE and CI_ROCM_DOCKERFILE to select alternative Dockerfiles in the normal AMD Buildkite pipeline', 并强调 'The regular ROCm Dockerfiles remain the defaults'. 讨论中 rasmith 直接问能否免掉 'git stuff and forced pushing', 作者确认这正是意图; micah-wil 补充该能力可用于 gfx1250 候选镜像与新的 ROCm preview, 说明动机不仅是 TheRock, 还包括通用候选镜像验证。

实现拆解

1. 新增共享选型脚本 (入口层): 新增 .buildkite/scripts/rocm/build-config.sh, 以 rocm_base_dockerfile() / rocm_ci_dockerfile() 解析 `${CI_ROCM_DOCKERFILE_BASE:-${ROCM_BASE_DOCKERFILE:-docker/Dockerfile.rocm_base}}` 这类三级回退, using_custom_rocm_dockerfiles() 判断是否偏离默认; validate_rocm_dockerfile() 做路径合法性 (禁止绝对路径、...、空白)、文件存在性、每个 FROM 必须带 AS 命名、以及必须包含调用方要求的阶段名; configure_rocm_build() 把结果导出为既有变量 ROCM_BASE_DOCKERFILE / CI_BASE_DOCKERFILE, 让下游脚本零改动复用, 并在自定义模式下关闭稳定 tag 推送 (ROCM_BASE_PUSH_STABLE_TAG=0、CI_BASE_PUSH_STABLE_TAG=0) 与依赖层导出。
2. 烘焙流水线接入自定义阶段: .buildkite/scripts/ci-bake-rocm.sh 新增 configure_custom_rocm_stages(), 要求自定义配方具备 ci_base test export_vllm export_test_smoke csrc-build rust-build 六个阶段, 并把全部阶段 (含辅助阶段) 写入

CI_BASE DOCKERFILE STAGES / ROCM_CSRC DOCKERFILE STAGES

/ROCM_RUST DOCKERFILE STAGES 以重算缓存键；当缺少 build_rocmshmem 阶段时 unset ROC_SHMEM_BRANCH ROC_SHMEM_CACHE_KEY DEEPEP_CACHE_KEY，避免烘焙引用不存在的 target。同时把 compute_rocm_csrc_content_hash、compute_rocm_rust_content_hash、write_rocm_build_arg_override、extract_dependency_pins、compute_dependency_cache_keys 里的硬编码 \${bake_dir}/Dockerfile.rocm 统一改为 \${CI_BASE DOCKERFILE:-\${bake_dir}/Dockerfile.rocm}。

3. 解析健壮性与缓存隔离：为了兼容 Rock Dockerfile 的小写风格，awk 判断改为 toupper(\$1) == "FROM"、阶段名 tolower()、sed 改为 [Aa][Rr][Gg]；init_bake_files() 在自定义模式下追加 --set '*.cache-to='，让实验构建只吃本地缓存、不覆盖同 commit/ 分支的标准 ROCm registry 缓存；init_config() 在自定义模式下强制要求 BASE_IMAGE 与 CI_BASE_IMAGE 由 base handoff 提供。
4. 基础镜像刷新的依赖元数据与缓存前缀：.buildkite/scripts/rocm/refresh-base-image.sh 引入 BASE_CACHE_PREFIX，自定义时取两个 Dockerfile 路径 sha256 的前 12 位作为命名空间 (rocm-base-<hash>-<scope>)；build_base_image() 先取 ROCM_SDK_VERSION，再用 rocm_version_from_base_image 兜底，并对 TRITON/PYTORCH/PYTORCH_VISION/PYTORCH_AUDIO 增加 *_BRANCH → *_VERSION 回退，从而同时适配源码构建 (Dockerfile.rocm) 与 wheel 安装 (Dockerfile.rock_base) 两种配方；main() 在自定义模式下拒绝 ROCM_BASE_REFRESH_SKIP=1 并先做 validate_rocm_dockerfile。
5. 运行期交接与 bake 定义：.buildkite/scripts/hardware_ci/run-amd-test.sh source 共享脚本，把 CI_ROCM DOCKERFILE_BASE、CI_ROCM DOCKERFILE、ROCM_BASE DOCKERFILE、CI_BASE DOCKERFILE 加入 clear_ci_orchestration_env 清理列表，保证测试阶段环境不被构建阶段残留污染；prepare_artifact_image() 在自定义模式下若 ci-base-image.txt 元数据缺失则直接 return 1，退回完整 CI 镜像而不是猜测 base image。docker/docker-bake-rocm.hcl 新增 variable "CI_BASE DOCKERFILE"（默认 docker/Dockerfile.rocm）并把 _common-rocm target 的 dockerfile 改为该变量。
6. 测试与验证配套：commit df17e694 明确 'Remove added Dockerfile regression tests'，最终 PR 不含新增自动化测试，也没有源码 / 测试联动信号；验证依赖 Buildkite 实跑 (/amd-ci run nightly → #12855、micah-wil 用三个环境变量实跑 #12883 成功) 以及作者声明的既有 ROCm CI 元数据 / 冒烟测试与 shell 行为检查。

关键文件：

- .buildkite/scripts/rocm/build-config.sh（模块 构建配置；类别 infra；类型 core-logic；符号 rocm_base_dockerfile, rocm_ci_dockerfile, using_custom_rocm_dockerfiles, rocm_dockerfile_stages）：本 PR 的核心新增文件，定义所有后续行为开关：Dockerfile 选择、是否自定义、路径与阶段校验、稳定 tag 与依赖层导出的抑制策略。
- .buildkite/scripts/ci-bake-rocm.sh（模块 镜像烘焙；类别 infra；类型 core-logic；符号 configure_custom_rocm_stages, init_config, validate_inputs, init_bake_files）：烘焙流水线的主战场：新增自定义阶段校验与缓存键重算，把多处硬编码 Dockerfile.rocm 改为可覆盖，并决定自定义构建不向 registry 导出缓存。

- `.buildkite/scripts/rocm/refresh-base-image.sh` (模块 基础镜像; 类别 infra; 类型 core-logic; 符号 `extract_arg_default`, `configure_rocm_base_layer_cache`, `build_base_image`, `main`) : 基础镜像刷新阶段: 把 registry 层缓存按 Dockerfile 组合分命名空间, 并为 SDK / Triton / PyTorch 增加 source-build 与 wheel 两套 ARG 回退。
- `.buildkite/scripts/hardware_ci/run-amd-test.sh` (模块 测试运行; 类别 infra; 类型 configuration; 符号 `clear_ci_orchestration_env`, `prepare_artifact_image`) : 运行期交接点: 清理构建阶段遗留的 Dockerfile 环境变量, 并在自定义栈缺少 `ci_base` 元数据时改为失败退回完整 CI 镜像。
- `docker/docker-bake-rocm.hcl` (模块 镜像定义; 类别 config; 类型 configuration; 符号 `CI_BASE_DOCKERFILE`) : `bake` 定义侧引入 `CI_BASE_DOCKERFILE` 变量, 使 `_common-rocm` 的 `dockerfile` 可由 CI 环境选择, 是整条覆盖链的最后一环。

关键符号: `rocm_base_dockerfile`, `rocm_ci_dockerfile`, `using_custom_rocm_dockerfiles`, `rocm_dockerfile_stages`, `validate_rocm_dockerfile`, `configure_rocm_build`, `configure_custom_rocm_stages`, `init_config`, `init_bake_files`, `validate_inputs`, `compute_rocm_csrc_content_hash`, `compute_rocm_rust_content_hash`, `write_rocm_build_arg_override`, `extract_dependency_pins`, `compute_dependency_cache_keys`, `extract_arg_default`, `configure_rocm_base_layer_cache`, `build_base_image`, `clear_ci_orchestration_env`, `prepare_artifact_image`

关键源码片段

`.buildkite/scripts/rocm/build-config.sh`

本 PR 的核心新增文件, 定义所有后续行为开关: Dockerfile 选择、是否自定义、路径与阶段校验、稳定 tag 与依赖层导出的抑制策略。

```
#!/usr/bin/env bash
# AMD CI 构建链共用的 Dockerfile 选择逻辑。
# CI_ROCM_DOCKERFILE_BASE 与 CI_ROCM_DOCKERFILE 接受仓库相对路径,
# 跑 TheRock 时分别指向 docker/Dockerfile.rocm_base 与 docker/Dockerfile.rocm。
# 自定义配方必须保留 CI 目标名与产物布局, 额外的 COPY 输入也要配套 *_CONTENT_FILES 覆盖。

rocm_base_dockerfile() {
    # 优先级: CI_ROCM_DOCKERFILE_BASE > 旧变量 ROCM_BASE_DOCKERFILE > 官方默认
    local default="docker/Dockerfile.rocm_base"
    printf '%s\n' "${CI_ROCM_DOCKERFILE_BASE:-${ROCM_BASE_DOCKERFILE:-${default}}}"
}

rocm_ci_dockerfile() {
    local default="docker/Dockerfile.rocm"
    printf '%s\n' "${CI_ROCM_DOCKERFILE:-${CI_BASE_DOCKERFILE:-${default}}}"
}

using_custom_rocm_dockerfiles() {
    # 两个路径只要有一个偏离默认, 就视为自定义镜像栈,
```

```
# 后续的阶段校验、缓存隔离、禁止推送稳定 tag 都基于这个判断
[[ "$(rocm_base_dockerfile)" != "docker/Dockerfile.rocm_base" \
  || "$(rocm_ci_dockerfile)" != "docker/Dockerfile.rocm" ]]
}

rocm_dockerfile_stages() {
# awk 解析 FROM ... AS <stage>, 并统一转小写, 兼容 Rock Dockerfile 的小写风格
awk '
    toupper($1) == "FROM" {
        for (idx = 2; idx < NF; idx++) {
            if (tolower($idx) == "as") print tolower($(idx + 1))
        }
    }
' "$1"
}

validate_rocm_dockerfile() {
local dockerfile="$1"
local stage="" stages=""
shift

# 先拒绝对路径、'..' 与空白字符, 防止构建环境变量把构建指到仓库之外
case "${dockerfile}" in
/*|../*|*/../*|../.*|*[:space:]*)
    echo "ROCm Dockerfile must be a repository-relative path without '..' or whitespace:
${dockerfile}" >&2
    return 2
;;
esac

if [[ ! -f "${dockerfile}" ]]; then
    echo "ROCm Dockerfile not found: ${dockerfile}" >&2
    return 2
fi

# 每个 FROM 都必须带 AS 命名, 否则按阶段名计算的缓存键会失配
if (($#) && ! awk '
    toupper($1) == "FROM" {
        named = 0
        for (idx = 2; idx < NF; idx++) if (tolower($idx) == "as") named = 1
        if (!named) exit 1
    }
' "${dockerfile}"); then
    echo "ROCm Dockerfile must name every build stage: ${dockerfile}" >&2
    return 2
fi

# 调用方要求的阶段必须全部存在, 否则烘焙会找不到对应 target
stages=$(rocm_dockerfile_stages "${dockerfile}")
for stage in "$@"; do
    if ! grep -Fxq -- "${stage}" <<< "${stages}"; then
        echo "ROCm Dockerfile ${dockerfile} is missing required stage: ${stage}" >&2
    fi
done
```

```

    return 2
fi
done
}

configure_rocm_build() {
    # 把选中的 Dockerfile 回填到既有变量名，让下游脚本零改动复用
    ROCM_BASE_DOCKERFILE=$(rocm_base_dockerfile)
    CI_BASE_DOCKERFILE=$(rocm_ci_dockerfile)
    export ROCM_BASE_DOCKERFILE CI_BASE_DOCKERFILE
    using_custom_rocm_dockerfiles || return 0

    # 自定义栈未必有独立依赖阶段，关闭依赖层导出
    export ROCM_DEP_CACHE_EXPORT_MODE=never
    # 保留每次构建唯一的运行时 tag，同时阻止实验构建把镜像提升为稳定 tag
    export ROCM_BASE_PUSH_STABLE_TAG=0
    export CI_BASE_PUSH_STABLE_TAG=0
    echo "Custom ROCm Dockerfiles: ${ROCM_BASE_DOCKERFILE} and ${CI_BASE_DOCKERFILE}"
}

```

[.buildkite/scripts/ci-bake-rocm.sh](#)

烘焙流水线的主战场：新增自定义阶段校验与缓存键重算，把多处硬编码 Dockerfile.rocm 改为可覆盖，并决定自定义构建不向 registry 导出缓存。

```

# 在 prepare_ci_build_context 之后调用：自定义配方必须提供 CI 链所需的全部阶段
configure_custom_rocm_stages() {
    using_custom_rocm_dockerfiles || return 0
    local context_root="${ROCM_BUILD_CONTEXT_ROOT:-.}"
    local dockerfile="${context_root}/${CI_BASE_DOCKERFILE}"
    local stages=""

    (cd "${context_root}" && \
        validate_rocm_dockerfile "${CI_BASE_DOCKERFILE}" \
        ci_base test export_vllm export_test_smoke csrc-build rust-build) \
    || return $?

    # 自定义布局下把所有阶段都纳入哈希（包括额外辅助阶段），保证缓存键覆盖完整
    stages=$(rocm_dockerfile_stages "${dockerfile}" | paste -sd ' ')
    CI_BASE_DOCKERFILE_STAGES="${CI_BASE_DOCKERFILE_STAGES:-${stages}}"
    ROCM_CSRC_DOCKERFILE_STAGES="${ROCM_CSRC_DOCKERFILE_STAGES:-${stages}}"
    ROCM_RUST_DOCKERFILE_STAGES="${ROCM_RUST_DOCKERFILE_STAGES:-${stages}}"
    # Rock 栈没有 build_rocshmem 阶段，清掉相关依赖变量，避免烘焙引用不存在的 target
    if [[ "${stages}" != *"build_rocshmem"* ]]; then
        unset ROCSHMEM_BRANCH ROCSHMEM_CACHE_KEY DEEPEEP_CACHE_KEY 2>/dev/null ||
        true
        unset ROCSHMEM_BRANCH ROCSHMEM_CACHE_KEY DEEPEEP_CACHE_KEY
    fi
}

```

```

init_config() {
    # 共享选型脚本在此处生效，TARGET 解析之前就完成变量回填
    source "$(dirname "${BASH_SOURCE[0]}")/rocm/build-config.sh"
    configure_rocm_build
    TARGET="${1:-test-ci}"
    # 自定义栈必须拿到 base handoff 产出的镜像引用，否则直接失败而不是静默回退
    if using_custom_rocm_dockerfiles; then
        if [[ -z "${BASE_IMAGE:-}" ]]; then
            echo "Custom ROCm builds require BASE_IMAGE from the selected base handoff" >&2
            return 1
        fi
        if ! is_ci_base_target && [[ -z "${CI_BASE_IMAGE:-}" ]]; then
            echo "Custom ROCm builds require CI_BASE_IMAGE from the selected ci_base handoff"
            >&2
            return 1
        fi
    fi
    # 阶段集合由 configure_custom_rocm_stages 决定，这里不再强行套用默认值
    if ! using_custom_rocm_dockerfiles; then
        CI_BASE_DOCKERFILE_STAGES="${CI_BASE_DOCKERFILE_STAGES:-${DEFAULT_CI_BASE_DOCKERFILE_STAGES}}"
    fi
    export PYTORCH_ROCM_ARCH CI_BASE_DOCKERFILE
}

init_bake_files() {
    BAKE_FILES=(-f "${VLLM_BAKE_FILE}" -f "${CI_HCL_PATH}")
    if using_custom_rocm_dockerfiles; then
        # 清空 cache-to: 自定义栈只复用本地缓存，避免同 commit 或分支下
        # 标准 ROCm 的 registry 缓存被实验镜像覆盖
        BAKE_FILES+=(--set '*.cache-to=')
    fi
}

```

[.buildkite/scripts/rocm/refresh-base-image.sh](#)

基础镜像刷新阶段：把 registry 层缓存按 Dockerfile 组合分命名空间，并为 SDK / Triton / PyTorch 增加 source-build 与 wheel 两套 ARG 回退。

```

# source 共享脚本后完成选型；DOCKERFILE 直接取回填后的 ROCM_BASE_DOCKERFILE
source "$(dirname "${BASH_SOURCE[0]}")/build-config.sh"
configure_rocm_build

# 自定义 Dockerfile 组合派生独立缓存命名空间：前缀取两个路径 sha256 的前 12 位，
# 标准化栈继续保持 rocm-base，实验构建无法覆盖标准缓存的 trusted ref
BASE_CACHE_PREFIX="rocm-base"
if using_custom_rocm_dockerfiles; then
    BASE_CACHE_PREFIX="rocm-base-$(printf '%s\n%s\n' \
        "${ROCM_BASE_DOCKERFILE}" "${CI_BASE_DOCKERFILE}" | sha256sum | cut -c1-12)"
fi

```

```

ROCM_BASE_TRUSTED_LAYER_CACHE_REF="${CACHE_REPO}:${BASE_CACHE_PREFIX}-main"

configure_rocm_base_layer_cache() {
    # 层缓存引用统一带上前缀，自定义栈与标准栈互不可见
    scope=$(rocm_base_layer_cache_scope)
    ROCM_BASE_LAYER_CACHE_REF="${CACHE_REPO}:${BASE_CACHE_PREFIX}-${scope}"
    ROCM_BASE_CACHE_ARGS+=(
        --cache-from "type=registry,ref=${ROCM_BASE_LAYER_CACHE_REF}"
    )
}

extract_arg_default() {
    # sed 改为对 ARG 大小写不敏感，兼容 Rock Dockerfile 里的 arg 写法
    local arg_name="$1"
    sed -n -E "s/^[[:space:]]*[Aa][Rr][Gg][[:space:]]+${arg_name}=\"?([^\"][:space:]]+)\"?.*$/p" \
        "${DOCKERFILE}" | head -1
}

build_base_image() {
    # 依赖元数据双通道解析：先看源码构建风格 (*_BRANCH / ROCM_SDK_VERSION) ,
    # 再回退到 wheel 风格 (*_VERSION) , 从而同时兼容 Dockerfile.rocm 与 Dockerfile.rock_base
    rocm_version="$(extract_arg_default ROCM_SDK_VERSION)"
    rocm_version="${rocm_version:-$(rocm_version_from_base_image "${base_image_arg}")}"
    triton_arg="$(extract_arg_default TRITON_BRANCH)"
    pytorch_arg="$(extract_arg_default PYTORCH_BRANCH)"
    pytorch_vision_arg="$(extract_arg_default PYTORCH_VISION_BRANCH)"
    pytorch_audio_arg="$(extract_arg_default PYTORCH_AUDIO_BRANCH)"
    triton_arg="${triton_arg:-$(extract_arg_default TRITON_VERSION)}"
    pytorch_arg="${pytorch_arg:-$(extract_arg_default TORCH_VERSION)}"
    pytorch_vision_arg="${pytorch_vision_arg:-$(extract_arg_default TORCHVISION_VERSION)}"
    pytorch_audio_arg="${pytorch_audio_arg:-$(extract_arg_default TORCHAUDIO_VERSION)}"
}

main() {
    if using_custom_rocm_dockerfiles; then
        # 自定义栈不允许跳过 base 刷新，否则拿不到与配方匹配的基础镜像
        if [[ "${ROCM_BASE_REFRESH_SKIP:-0}" == "1" ]]; then
            echo "Custom ROCm Dockerfiles require ROCM_BASE_REFRESH_SKIP=0" >&2
            return 2
        fi
        validate_rocm_dockerfile "${DOCKERFILE}" || return $?
    fi
    # ... 后续沿用原有的 metadata_set / 刷新流程 ...
}

```

评论区精华

讨论集中在两点。其一是设计收敛：micah-wil 在 [run-amd-test.sh](#) 的 hunk 上质疑是否真的需要 [VLLM_USE_ROCK](#) 开关，'Basically, I am wondering if we can boil this PR down to

just two additional env vars (CI_ROCM_DOCKERFILE_BASE and CI_ROCM_DOCKERFILE) to override the Dockerfiles', 并指出这对 gfx1250、新 ROCm preview 同样有用；最终实现采纳了该建议，commit [c33bf976](#) 'Use Dockerfile overrides for custom AMD CI builds' 正是这次重构。其二是依赖元数据可靠性：rasmith 连问 'Where does this come from? Could it end up being empty?' 与 'Hmm, we have both of these (TRITON_BRANCH and TRITON_VERSION)?'，作者答复 TRITON_VERSION 来自所选 base Dockerfile 的 ARG (Rock 侧 pin 在 3.8.0+git4cff872c.rocm10.0.0)，且空值只影响依赖元数据、不会回流构建或参与缓存镜像选择。验证侧由 micah-wil 实跑通过并给出链接，rasmith、tjtanaa 均 APPROVED，未留下未解决阻塞项。

- 是否保留 VLLM_USE_ROCK 开关，还是只用两个 Dockerfile 环境变量 (design): 作者采纳该建议，commit [c33bf976](#) 'Use Dockerfile overrides for custom AMD CI builds' 把实现重构为纯 Dockerfile 覆盖机制，最终 PR body 与代码中只保留 CI_ROCM_DOCKERFILE_BASE / CI_ROCM_DOCKERFILE 两个变量，机制因此对任意兼容 Dockerfile 对都可复用。
- ROCM_SDK_VERSION 取值来源与空值风险 (question): 确认两套当前配方都不会留空，风险降级为元数据可观测性问题，未做强制断言，review 通过。
- TRITON_BRANCH 与 TRITON_VERSION 双变量是否必要 (design): 双通道回退被视为兼容两套配方的必要成本，保留 *_BRANCH → *_VERSION 的解析顺序。
- 自定义 Dockerfile 栈的实跑验证 (testing): 随后更新 'Edit: Seems to have run successfully here <https://buildkite.com/vllm/amd-ci/builds/12883>', 端到端验证通过并获得 APPROVED。
- 移除已加入的 Dockerfile 回归测试 (testing): PR 以脚本校验 (validate_rocm_dockerfile 返回码 2) 加人工 Buildkite 实跑替代自动化测试，build-config.sh 的校验分支缺少 CI 兜底。

风险与影响

- 风险：1) 外部输入决定构建内容：CI_ROCM_DOCKERFILE* 来自 Buildkite 构建环境，虽然有 validate_rocm_dockerfile 的仓库相对路径 / 阶段名校验，但一旦误配仍可能烘焙出非预期镜像；缓解措施是自定义模式下 ROCM_BASE_PUSH_STABLE_TAG=0、CI_BASE_PUSH_STABLE_TAG=0，不会把实验镜像提升为稳定 tag。2) 缓存冷启动代价：自定义栈清空 cache-to (--set '*.cache-to=') 并把 registry 缓存前缀换成 rocm-base-<hash>，避免了污染标准缓存，但实验镜像无法导出 registry 缓存，重复构建会明显变慢。3) 解析语义被放宽：toupper(\$1) == "FROM"、[Aa][Rr][Gg]、阶段名统一小写，改变了 ci-bake-rocm.sh / refresh-base-image.sh 原解析器的匹配范围，若未来出现仅大小写不同的阶段名会发生键合并。4) 依赖元数据可为空：ROCM_SDK_VERSION / TRITON_VERSION 等提取失败时只会写空元数据 (rasmith 的关注点)，缺少显式断言，排查问题时可观测性偏弱。5) 测试兜底缺失：commit [df17e694](#) 删除了早期加入的 Dockerfile 回归测试，最终只有脚本与配置改动，改动 build-config.sh 校验分支时没有 CI 用例保护。6) 与 main 的 Rock Dockerfile 冲突：合并提交 [6233a5b](#) 说明需要手工消化 Rock Dockerfile 重叠 (采用 main 的 smoke-test 阶段)，rock 相关文件与主线耦合仍在演进，存在再次冲突风险。7) 运行期容错语义变化：自定义模式下

prepare_artifact_image() 在元数据缺失时由“回退推导 base image”改为直接失败并退回完整 CI 镜像，属于有意的 fail-loud，但改变了既有容错行为。

- 影响：对 AMD CI：在不 fork、不强推、不改 Dockerfile 的前提下多出一条受支持的 TheRock / 候选镜像验证路径，micah-wil 已在 build 12883 上实跑成功，并明确该机制可复用用于 gfx1250 镜像与新的 ROCm preview。对默认路径：未设置环境变量时，using_custom_rocm_dockerfiles 短路掉几乎所有新逻辑（CI_BASE_DOCKERFILE_STAGES 仍取默认、BAKE_FILES 不追加 --set、缓存前缀保持 rocm-base），设计目标是零回归。对团队：AMD CI 维护者成为新变量的所有者，并隐含承接“自定义配方必须保留 CI 目标名、产物布局、额外 COPY 需配套 *_CONTENT_FILES”的契约，违反契约会以校验退出码 2 被拒绝；对不参与 AMD CI 的工程师影响可忽略。对下游 Rock 相关工作（如 TheRock 镜像下的内核测试）提供了统一的触发入口，降低临时补丁的维护成本。
- 风险标记：CI 基础设施变更，缺少自动化回归测试，外部环境变量决定构建镜像，缓存命名空间隔离，脚本解析语义放宽，人工实跑验证

关联脉络

- PR #57112 [ROCm][CI] Fix MLA RoPE fused-kernel tests for TheRock image: 同属 TheRock 镜像在 AMD CI 中的落地链路：本 PR 提供触发与构建 TheRock 运行时的统一入口，57112 则调整该镜像下 MLA RoPE fused kernel 测试的 ULP 容差，二者共同构成 TheRock 从构建到测试的闭环。
- PR #56545 [Build][NVIDIA] Update public Rubin dependencies and MSA compatibility: 同样是在 docker/ 构建层引入实验性或预发布依赖线并做兼容适配；可对照其 requirements/rubin-prerelease.txt 与 docker/Dockerfile 的隔离方式，与本 PR 的 CI_ROCM_DOCKERFILE 覆盖 + 稳定 tag/ 缓存隔离策略属于同一类“实验构建不污染主线”问题。
- PR #53940 [ROCm][Kimi-K3] Enable a4w4 flydsl kernels for KimiK3: ROCm 侧通过环境变量 (vllm/envs.py) 切换默认内核实现，与本 PR 通过环境变量切换镜像栈的做法同属 AMD 栈的可切换默认值模式，都在验证成本与默认路径稳定性之间取平衡，可作为该 PR 后续扩展模式的参考。