CI测试改造文件夹

```
.buildkite/
├── test-pipeline-optimized.yaml # 优化后的CI配置
├── nightly-pipeline.yaml # 夜间测试配置
└── scripts/
├── test-runner.sh # 测试执行脚本
├── test-selector.py # 智能测试选择
├── test-monitor.py # 测试时间监控
├── performance-regression.py # 性能回归检测
├── generate-report.py # 报告生成
└── ...

tests/
├── pytest_marks.py # 测试标记定义
pytest.ini # pytest配置
pyproject.toml # 项目配置
Makefile # 构建脚本
```

1. 新的测试标记配置文件

首先创建测试标记配置文件：

tests/pytest_marks.py

```
"""
测试标记定义和配置
"""
import pytest

# 时间分层标记
fast = pytest.mark.fast  # < 5分钟
standard = pytest.mark.standard  # 5-20分钟
extended = pytest.mark.extended  # 20-60分钟
nightly = pytest.mark.nightly  # > 60分钟

# 硬件需求标记
gpu_intensive = pytest.mark.gpu_intensive
memory_heavy = pytest.mark.memory_heavy
multi_gpu = pytest.mark.multi_gpu

# 功能模块标记
entrypoints = pytest.mark.entrypoints
spec_decode = pytest.mark.spec_decode
kernels = pytest.mark.kernels
models = pytest.mark.models
```

pytest.ini

```
[tool:pytest]
markers =
    fast: 快速测试 (< 5分钟)
    standard: 标准测试 (5-20分钟)
    extended: 扩展测试 (20-60分钟)
    nightly: 夜间测试 (> 60分钟)
    gpu_intensive: GPU密集型测试
    memory_heavy: 内存密集型测试
    multi_gpu: 多GPU测试
    entrypoints: 入口点测试
    spec_decode: 规范解码测试
    kernels: 内核测试
    models: 模型测试
timeout = 300
timeout_method = thread
```

2. 重构后的主要CI配置文件

.buildkite/test-pipeline-optimized.yaml

```
# 优化后的CI测试流水线配置
# 基于时间分层和功能模块的并行化策略

steps:
##### 快速检查层 (< 5分钟) #####

- label: "🚀 Fast Check - Documentation Build
  key: "fast-docs"
  mirror_hardwares: [amdexperimental]
  working_dir: "/vllm-workspace/test_docs"
  fast_check: true
  no_gpu: true
  timeout_in_minutes: 5
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  commands:
    - pip install -r ../requirements/docs.txt
    - mkdocs build

- label: "🚀 Fast Check - Core Unit Tests
  key: "fast-core"
  mirror_hardwares: [amdexperimental, amdproduction]
  fast_check: true
  timeout_in_minutes: 5
  retry:
    automatic:
```

```
      - exit_status: "*"
        limit: 1
  commands:
    - pytest -v -s -m "fast and not gpu_intensive" core/ --timeout=300

- label: "🚀 Fast Check - Basic Imports & Utils
  key: "fast-utils"
  mirror_hardwares: [amdexperimental, amdproduction]
  fast_check: true
  timeout_in_minutes: 5
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  commands:
    - python3 standalone_tests/lazy_imports.py
    - pytest -v -s -m "fast" test_utils.py --timeout=300

##### 标准测试层 (5-20分钟) #####

- label: "⚡ Standard - Entrypoints LLM"
  key: "std-entrypoints-llm"
  depends_on: ["fast-core", "fast-utils"]
  mirror_hardwares: [amdexperimental]
  timeout_in_minutes: 20
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - vllm/entrypoints/llm.py
    - tests/entrypoints/llm/
  commands:
    - export VLLM_WORKER_MULTIPROC_METHOD=spawn
    - pytest -v -s -m "standard and entrypoints" entrypoints/llm/ --timeout=1200

- label: "⚡ Standard - Entrypoints OpenAI"
  key: "std-entrypoints-openai"
  depends_on: ["fast-core", "fast-utils"]
  mirror_hardwares: [amdexperimental]
  timeout_in_minutes: 20
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - vllm/entrypoints/openai/
    - tests/entrypoints/openai/
  commands:
    - export VLLM_WORKER_MULTIPROC_METHOD=spawn
    - pytest -v -s -m "standard and entrypoints" entrypoints/openai/ --timeout=1200

- label: "⚡ Standard - Spec Decode Core"
```

```
  key: "std-spec-decode-core"
  depends_on: ["fast-core"]
  mirror_hardwares: [amdexperimental]
  timeout_in_minutes: 20
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - vllm/spec_decode/
    - tests/spec_decode/
  commands:
    - pytest -v -s -m "standard and spec_decode and not e2e" spec_decode/
--timeout=1200

- label: "⚡ Standard - Kernels Core %N"
  key: "std-kernels-core"
  depends_on: ["fast-core"]
  mirror_hardwares: [amdexperimental, amdproduction]
  timeout_in_minutes: 15
  parallelism: 3
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - csrc/
    - tests/kernels/core/
    - tests/kernels/attention/
    - tests/kernels/quantization/
  commands:
    - pytest -v -s -m "standard and kernels" kernels/
--shard-id=$BUILDKITE_PARALLEL_JOB
--num-shards=$BUILDKITE_PARALLEL_JOB_COUNT --timeout=900

- label: "⚡ Standard - Basic Models"
  key: "std-models-basic"
  depends_on: ["fast-core"]
  mirror_hardwares: [amdexperimental, amdproduction]
  timeout_in_minutes: 20
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - vllm/model_executor/models/
    - tests/models/
  commands:
    - pytest -v -s -m "standard and models and core_model" models/ --timeout=1200

##### 扩展测试层 (20-60分钟) #####

- label: "🔧 Extended - Spec Decode E2E
```

```
  key: "ext-spec-decode-e2e"
  depends_on: ["std-spec-decode-core"]
  mirror_hardwares: [amdexperimental]
  timeout_in_minutes: 45
  optional: true
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - vllm/spec_decode/
    - tests/spec_decode/e2e/
  commands:
    - pytest -v -s -m "extended and spec_decode and e2e" spec_decode/e2e/
--timeout=2700

- label: "🔧 Extended - Multi-GPU Tests
  key: "ext-multi-gpu"
  depends_on: ["std-kernels-core"]
  mirror_hardwares: [amdexperimental]
  num_gpus: 4
  timeout_in_minutes: 50
  optional: true
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - vllm/distributed/
    - tests/distributed/
  commands:
    - pytest -v -s -m "extended and multi_gpu" distributed/ --timeout=3000

- label: "🔧 Extended - Language Models
  key: "ext-models-language"
  depends_on: ["std-models-basic"]
  mirror_hardwares: [amdexperimental]
  timeout_in_minutes: 60
  optional: true
  retry:
    automatic:
      - exit_status: "*"
        limit: 1
  source_file_dependencies:
    - tests/models/language/
  commands:
    - pip install 'git+https://github.com/Dao-AILab/causal-conv1d@v1.5.0.post8'
    - pytest -v -s -m "extended and models and not core_model" models/language/
--timeout=3600

##### 夜间测试层 (> 60分钟) #####

- label: "🌙 Nightly - Performance Benchmarks
```

```
  key: "nightly-benchmarks"
  depends_on: ~
  mirror_hardwares: [amdexperimental]
  timeout_in_minutes: 120
  if: build.branch == "main" || build.env("FORCE_NIGHTLY") == "true"
  commands:
    - pytest -v -s -m "nightly and benchmarks" benchmarks/ --timeout=7200

- label: "🌙 Nightly - Large Model Tests
  key: "nightly-large-models"
  depends_on: ~
  gpu: a100
  num_gpus: 4
  timeout_in_minutes: 180
  if: build.branch == "main" || build.env("FORCE_NIGHTLY") == "true"
  commands:
    - pytest -v -s -m "nightly and models and large" models/ --timeout=10800

- label: "🌙 Nightly - Stress Tests
  key: "nightly-stress"
  depends_on: ~
  mirror_hardwares: [amdexperimental]
  timeout_in_minutes: 150
  if: build.branch == "main" || build.env("FORCE_NIGHTLY") == "true"
  commands:
    - pytest -v -s -m "nightly and stress" --timeout=9000

##### 条件触发的专项测试 #####

- label: "🎯 Conditional - AMD Hardware
  key: "cond-amd"
  depends_on: ["fast-core"]
  mirror_hardwares: [amdproduction]
  timeout_in_minutes: 30
  if: build.env("TEST_AMD") == "true" || build.branch == "main"
  commands:
    - pytest -v -s -m "standard and amd" --timeout=1800

- label: "🎯 Conditional - Memory Heavy
  key: "cond-memory"
  depends_on: ["std-models-basic"]
  gpu: a100
  timeout_in_minutes: 40
  if: build.env("TEST_MEMORY") == "true" || build.branch == "main"
  commands:
    - pytest -v -s -m "extended and memory_heavy" --timeout=2400""
```

3. 测试选择和执行脚本

.buildkite/scripts/test-selector.py

```
#!/usr/bin/env python3
"""
智能测试选择器 - 根据代码变更选择相关测试
"""
import os
import sys
import subprocess
import json
from pathlib import Path
from typing import List, Set

class TestSelector:
    def __init__(self):
        self.changed_files = self._get_changed_files()
        self.test_mapping = self._load_test_mapping()

    def _get_changed_files(self) -> List[str]:
        """获取变更的文件列表"""
        try:
            result = subprocess.run(
                ["git", "diff", "--name-only", "origin/main...HEAD"],
                capture_output=True, text=True, check=True
            )
            return result.stdout.strip().split('\n') if result.stdout.strip() else []
        except subprocess.CalledProcessError:
            return []

    def _load_test_mapping(self) -> dict:
        """加载文件到测试的映射关系"""
        return {
            "vllm/entrypoints/": ["entrypoints"],
            "vllm/spec_decode/": ["spec_decode"],
            "vllm/model_executor/": ["models", "kernels"],
            "csrc/": ["kernels"],
            "vllm/distributed/": ["distributed"],
            "tests/": ["all"]
        }

    def select_tests(self) -> Set[str]:
        """根据变更文件选择需要运行的测试"""
        if not self.changed_files:
            return {"fast", "standard"}

        selected_tests = set()

        for file_path in self.changed_files:
            for pattern, tests in self.test_mapping.items():
                if file_path.startswith(pattern):
                    if "all" in tests:
                        return {"fast", "standard", "extended"}
                    selected_tests.update(tests)
```

```
        # 默认至少运行快速测试
        if not selected_tests:
            selected_tests.add("fast")

        return selected_tests

    def generate_pipeline_filter(self) -> str:
        """生成Buildkite流水线过滤器"""
        selected = self.select_tests()

        filters = []
        if "fast" in selected:
            filters.append("fast-*")
        if "standard" in selected:
            filters.append("std-*")
        if "extended" in selected:
            filters.append("ext-*")

        return " ".join(filters)

if __name__ == "__main__":
    selector = TestSelector()
    filter_str = selector.generate_pipeline_filter()
    print(filter_str)
```

4. 超时和重试配置

.buildkite/scripts/test-runner.sh

```bash
#!/bin/bash
# 测试运行器 - 统一的测试执行脚本

set -e

# 配置参数
TEST_CATEGORY=${1:-"standard"}
TEST_MODULE=${2:-""}
TIMEOUT_MINUTES=${3:-20}
MAX_RETRIES=${4:-1}

# 计算超时秒数
TIMEOUT_SECONDS=$((TIMEOUT_MINUTES * 60))

# 设置pytest参数
PYTEST_ARGS="-v -s --timeout=${TIMEOUT_SECONDS}"
```

```
# 根据测试类别添加标记过滤
case $TEST_CATEGORY in
    "fast")
        PYTEST_ARGS="$PYTEST_ARGS -m 'fast'"
        ;;
    "standard")
        PYTEST_ARGS="$PYTEST_ARGS -m 'standard'"
        ;;
    "extended")
        PYTEST_ARGS="$PYTEST_ARGS -m 'extended'"
        ;;
    "nightly")
        PYTEST_ARGS="$PYTEST_ARGS -m 'nightly'"
        ;;
    *)
        echo "Unknown test category: $TEST_CATEGORY"
        exit 1
        ;;
esac

# 添加模块过滤
if [[ -n "$TEST_MODULE" ]]; then
    PYTEST_ARGS="$PYTEST_ARGS and $TEST_MODULE"
fi

# 执行测试，支持重试
RETRY_COUNT=0
while [ $RETRY_COUNT -le $MAX_RETRIES ]; do
    echo "Running tests (attempt $((RETRY_COUNT + 1))/$((MAX_RETRIES + 1))): pytest
$PYTEST_ARGS"

    if pytest $PYTEST_ARGS; then
        echo "Tests passed successfully"
        exit 0
    else
        RETRY_COUNT=$((RETRY_COUNT + 1))
        if [ $RETRY_COUNT -le $MAX_RETRIES ]; then
            echo "Test failed, retrying in 10 seconds..."
            sleep 10
        fi
    fi
done

echo "Tests failed after $((MAX_RETRIES + 1)) attempts"


exit 1
```

5. 测试时间监控脚本

.buildkite/scripts/test-monitor.py

```
#!/usr/bin/env python3
"""
```

```
测试时间监控和分析工具
"""
import json
import time
import subprocess
import sys
from pathlib import Path
from typing import Dict, List, Tuple
from datetime import datetime

class TestMonitor:
    def __init__(self):
        self.results_file = Path("test_timing_results.json")
        self.load_historical_data()

    def load_historical_data(self):
        """加载历史测试时间数据"""
        if self.results_file.exists():
            with open(self.results_file, 'r') as f:
                self.historical_data = json.load(f)
        else:
            self.historical_data = {}

    def save_results(self, results: Dict):
        """保存测试结果"""
        timestamp = datetime.now().isoformat()
        self.historical_data[timestamp] = results

        # 只保留最近30天的数据
        sorted_keys = sorted(self.historical_data.keys())
        if len(sorted_keys) > 30:
            for key in sorted_keys[:-30]:
                del self.historical_data[key]

        with open(self.results_file, 'w') as f:
            json.dump(self.historical_data, f, indent=2)

    def run_with_timing(self, test_command: str) -> Tuple[int, float]:
        """运行测试并记录时间"""
        start_time = time.time()

        try:
            result = subprocess.run(
                test_command.split(),
                capture_output=True,
                text=True,
                check=True
            )
            exit_code = 0
        except subprocess.CalledProcessError as e:
            exit_code = e.returncode

        end_time = time.time()
        duration = end_time - start_time

        return exit_code, duration
```

```python
    def analyze_trends(self) -> Dict:
        """分析测试时间趋势"""
        if not self.historical_data:
            return {}

        analysis = {}
        for timestamp, data in self.historical_data.items():
            for test_name, duration in data.items():
                if test_name not in analysis:
                    analysis[test_name] = []
                analysis[test_name].append(duration)

        # 计算平均时间和趋势
        trends = {}
        for test_name, durations in analysis.items():
            avg_duration = sum(durations) / len(durations)
            recent_avg = sum(durations[-5:]) / min(5, len(durations))
            trend = "increasing" if recent_avg > avg_duration * 1.1 else "stable"

            trends[test_name] = {
                "average_duration": avg_duration,
                "recent_average": recent_avg,
                "trend": trend,
                "sample_count": len(durations)
            }

        return trends

    def suggest_optimizations(self) -> List[str]:
        """基于分析结果提出优化建议"""
        trends = self.analyze_trends()
        suggestions = []

        for test_name, data in trends.items():
            if data["average_duration"] > 300:  # 5分钟
                suggestions.append(f"Consider splitting {test_name} - average duration:
{data['average_duration']:.1f}s")

            if data["trend"] == "increasing":
                suggestions.append(f"Monitor {test_name} - showing increasing duration
trend")

        return suggestions

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: test-monitor.py <test_command>")
        sys.exit(1)

    monitor = TestMonitor()
    test_command = " ".join(sys.argv[1:])

    exit_code, duration = monitor.run_with_timing(test_command)

    # 保存结果
    test_name = test_command.split()[-1] if test_command else "unknown"
    monitor.save_results({test_name: duration})
```

```
    # 输出分析
    print(f"Test completed in {duration:.2f}s with exit code {exit_code}")

    suggestions = monitor.suggest_optimizations()
    if suggestions:
        print("\nOptimization suggestions:")
        for suggestion in suggestions:
            print(f"  - {suggestion}")


    sys.exit(exit_code)
```

6. 动态测试分片脚本

.buildkite/scripts/test-sharding.py

```
#!/usr/bin/env python3
"""
动态测试分片工具 - 根据测试历史时间智能分片
"""
import json
import subprocess
import sys
from pathlib import Path
from typing import List, Dict, Tuple
import math

class TestSharding:
    def __init__(self):
        self.timing_data_file = Path("test_timing_results.json")
        self.load_timing_data()

    def load_timing_data(self):
        """加载测试时间数据"""
        if self.timing_data_file.exists():
            with open(self.timing_data_file, 'r') as f:
                data = json.load(f)
                # 获取最新的时间数据
                if data:
                    latest_key = max(data.keys())
                    self.timing_data = data[latest_key]
                else:
                    self.timing_data = {}
        else:
            self.timing_data = {}

    def discover_tests(self, test_path: str, markers: str = "") -> List[str]:
        """发现测试文件"""
        cmd = ["pytest", "--collect-only", "-q", test_path]
        if markers:
            cmd.extend(["-m", markers])

        try:
```

```
            result = subprocess.run(cmd, capture_output=True, text=True, check=True)
            test_files = []
            for line in result.stdout.split('\n'):
                if '::' in line and 'test_' in line:
                    test_file = line.split('::')[0]
                    if test_file not in test_files:
                        test_files.append(test_file)
            return test_files
        except subprocess.CalledProcessError:
            return []

    def estimate_test_duration(self, test_file: str) -> float:
        """估算测试文件执行时间"""
        # 使用历史数据或默认估算
        if test_file in self.timing_data:
            return self.timing_data[test_file]

        # 基于文件大小的简单估算
        try:
            file_path = Path(test_file)
            if file_path.exists():
                file_size = file_path.stat().st_size
                # 假设每KB代码需要1秒执行时间
                return max(30, file_size / 1024)
            else:
                return 60  # 默认1分钟
        except:
            return 60

    def create_shards(self, test_files: List[str], num_shards: int) -> List[List[str]]:
        """创建平衡的测试分片"""
        if not test_files:
            return []

        # 计算每个测试的预估时间
        test_durations = [(test, self.estimate_test_duration(test)) for test in test_files]
        test_durations.sort(key=lambda x: x[1], reverse=True)  # 按时间降序排列

        # 初始化分片
        shards = [[] for _ in range(num_shards)]
        shard_times = [0.0] * num_shards

        # 贪心算法分配测试到分片
        for test_file, duration in test_durations:
            # 找到当前时间最少的分片
            min_shard_idx = shard_times.index(min(shard_times))
            shards[min_shard_idx].append(test_file)
            shard_times[min_shard_idx] += duration

        return shards

    def generate_shard_commands(self, test_path: str, markers: str, num_shards: int) ->
List[str]:
        """生成分片测试命令"""
        test_files = self.discover_tests(test_path, markers)
        shards = self.create_shards(test_files, num_shards)
```

```
        commands = []
        for i, shard in enumerate(shards):
            if shard:  # 只有非空分片才生成命令
                test_list = " ".join(shard)
                cmd = f"pytest -v -s {test_list}"
                if markers:
                    cmd += f" -m '{markers}'"
                commands.append(cmd)

        return commands

if __name__ == "__main__":
    if len(sys.argv) < 4:
        print("Usage: test-sharding.py <test_path> <markers> <num_shards>")
        sys.exit(1)

    test_path = sys.argv[1]
    markers = sys.argv[2]
    num_shards = int(sys.argv[3])

    sharding = TestSharding()
    commands = sharding.generate_shard_commands(test_path, markers, num_shards)

    for i, cmd in enumerate(commands):
        print(f"# Shard {i+1}")
        print(cmd)


        print()
```

7. 夜间测试配置

.buildkite/nightly-pipeline.yaml

```
# 夜间测试流水线配置
steps:

- label: "🌙 Nightly - Full Test Suite"
  key: "nightly-full"
  timeout_in_minutes: 240
  agents:
    queue: "gpu-large"
  commands:
    - echo "Starting full nightly test suite"
    - .buildkite/scripts/test-runner.sh nightly "" 240 0

- label: "🌙 Nightly - Performance Regression"
  key: "nightly-perf"
  depends_on: ["nightly-full"]
  timeout_in_minutes: 180
  agents:
    queue: "gpu-large"
  commands:
    - echo "Running performance regression tests"
    - python3 .buildkite/scripts/performance-regression.py
```

```yaml
- label: "🌙 Nightly - Memory Leak Detection"
  key: "nightly-memory"
  depends_on: ["nightly-full"]
  timeout_in_minutes: 120
  agents:
    queue: "gpu-large"
  commands:
    - echo "Running memory leak detection"
    - pytest -v -s -m "nightly and memory_heavy" --timeout=7200

- label: "🌙 Nightly - Large Model Tests"
  key: "nightly-large-models"
  depends_on: ["nightly-full"]
  gpu: a100
  num_gpus: 8
  timeout_in_minutes: 300
  commands:
    - echo "Testing large models"
    - pytest -v -s -m "nightly and large_model" --timeout=18000

- label: "🌙 Nightly - Stress Tests"
  key: "nightly-stress"
  depends_on: ["nightly-full"]
  timeout_in_minutes: 180
  parallelism: 4
  commands:
    - echo "Running stress tests - shard $$BUILDKITE_PARALLEL_JOB"
    - python3 .buildkite/scripts/test-sharding.py tests/ "nightly and stress" 4 | sed -n
"$((BUILDKITE_PARALLEL_JOB + 1))p" | bash

- label: "🌙 Nightly - Report Generation"
  key: "nightly-report"
  depends_on: ["nightly-perf", "nightly-memory", "nightly-large-models", "nightly-stress"]
  timeout_in_minutes: 30
  commands:
    - echo "Generating nightly test report"
    - python3 .buildkite/scripts/generate-report.py


    - buildkite-agent artifact upload "nightly-report.html"
```

8. 性能回归检测脚本

.buildkite/scripts/performance-regression.py

```python
#!/usr/bin/env python3
"""
性能回归检测工具
"""
import json
import subprocess
import sys
import time
from pathlib import Path
```

```python
from typing import Dict, List, Tuple
from datetime import datetime, timedelta

class PerformanceRegression:
    def __init__(self):
        self.benchmark_results_file = Path("benchmark_results.json")
        self.regression_threshold = 0.15  # 15%性能下降阈值
        self.load_historical_benchmarks()

    def load_historical_benchmarks(self):
        """加载历史基准测试数据"""
        if self.benchmark_results_file.exists():
            with open(self.benchmark_results_file, 'r') as f:
                self.historical_data = json.load(f)
        else:
            self.historical_data = {}

    def run_benchmark_suite(self) -> Dict:
        """运行基准测试套件"""
        benchmarks = {
            "throughput": self._run_throughput_benchmark(),
            "latency": self._run_latency_benchmark(),
            "serving": self._run_serving_benchmark()
        }
        return benchmarks

    def _run_throughput_benchmark(self) -> Dict:
        """运行吞吐量基准测试"""
        cmd = [
            "python", "benchmarks/benchmark_throughput.py",
            "--model", "meta-llama/Llama-2-7b-hf",
            "--dataset-name", "sonnet",
            "--num-prompts", "100",
            "--backend", "vllm"
        ]

        start_time = time.time()
        try:
            result = subprocess.run(cmd, capture_output=True, text=True, check=True)
            duration = time.time() - start_time

            # 解析输出获取性能指标
            output_lines = result.stdout.split('\n')
            throughput = None
            for line in output_lines:
                if "Throughput:" in line:
                    throughput = float(line.split(':')[1].strip().split()[0])
                    break

            return {
                "throughput": throughput,
                "duration": duration,
                "status": "success"
            }
        except subprocess.CalledProcessError as e:
            return {
                "throughput": None,
```

```
            "duration": time.time() - start_time,
            "status": "failed",
            "error": str(e)
        }

def _run_latency_benchmark(self) -> Dict:
    """运行延迟基准测试"""
    cmd = [
        "python", "benchmarks/benchmark_latency.py",
        "--model", "meta-llama/Llama-2-7b-hf",
        "--input-len", "32",
        "--output-len", "128",
        "--num-iters", "10"
    ]

    start_time = time.time()
    try:
        result = subprocess.run(cmd, capture_output=True, text=True, check=True)
        duration = time.time() - start_time

        # 解析延迟指标
        output_lines = result.stdout.split('\n')
        latency_p50 = None
        latency_p99 = None

        for line in output_lines:
            if "P50 latency:" in line:
                latency_p50 = float(line.split(':')[1].strip().split()[0])
            elif "P99 latency:" in line:
                latency_p99 = float(line.split(':')[1].strip().split()[0])

        return {
            "latency_p50": latency_p50,
            "latency_p99": latency_p99,
            "duration": duration,
            "status": "success"
        }
    except subprocess.CalledProcessError as e:
        return {
            "latency_p50": None,
            "latency_p99": None,
            "duration": time.time() - start_time,
            "status": "failed",
            "error": str(e)
        }

def _run_serving_benchmark(self) -> Dict:
    """运行服务基准测试"""
    # 启动vLLM服务器
    server_cmd = [
        "python", "-m", "vllm.entrypoints.openai.api_server",
        "--model", "meta-llama/Llama-2-7b-hf",
        "--port", "8000"
    ]

    server_process = subprocess.Popen(server_cmd)
    time.sleep(30)  # 等待服务器启动
```

```
        try:
            # 运行客户端基准测试
            client_cmd = [
                "python", "benchmarks/benchmark_serving.py",
                "--backend", "vllm",
                "--model", "meta-llama/Llama-2-7b-hf",
                "--dataset-name", "random",
                "--num-prompts", "50",
                "--request-rate", "2"
            ]

            start_time = time.time()
            result = subprocess.run(client_cmd, capture_output=True, text=True, check=True)
            duration = time.time() - start_time

            # 解析服务性能指标
            output_lines = result.stdout.split('\n')
            request_throughput = None
            mean_ttft = None

            for line in output_lines:
                if "Request throughput:" in line:
                    request_throughput = float(line.split(':')[1].strip().split()[0])
                elif "Mean TTFT:" in line:
                    mean_ttft = float(line.split(':')[1].strip().split()[0])

            return {
                "request_throughput": request_throughput,
                "mean_ttft": mean_ttft,
                "duration": duration,
                "status": "success"
            }
        except subprocess.CalledProcessError as e:
            return {
                "request_throughput": None,
                "mean_ttft": None,
                "duration": time.time() - start_time,
                "status": "failed",
                "error": str(e)
            }
        finally:
            server_process.terminate()
            server_process.wait()

    def detect_regressions(self, current_results: Dict) -> List[Dict]:
        """检测性能回归"""
        regressions = []

        if not self.historical_data:
            return regressions

        # 获取最近7天的基准数据作为基线
        baseline_date = datetime.now() - timedelta(days=7)
        baseline_results = {}

        for timestamp, data in self.historical_data.items():
```

```
            if datetime.fromisoformat(timestamp) >= baseline_date:
                for metric, value in data.items():
                    if metric not in baseline_results:
                        baseline_results[metric] = []
                    if value is not None:
                        baseline_results[metric].append(value)

        # 计算基线平均值并检测回归
        for metric, current_value in current_results.items():
            if current_value is None or metric not in baseline_results:
                continue

            if not baseline_results[metric]:
                continue

            baseline_avg = sum(baseline_results[metric]) / len(baseline_results[metric])

            # 对于吞吐量和请求吞吐量，值越大越好
            if metric in ["throughput", "request_throughput"]:
                if current_value < baseline_avg * (1 - self.regression_threshold):
                    regressions.append({
                        "metric": metric,
                        "current_value": current_value,
                        "baseline_avg": baseline_avg,
                        "regression_percent": (baseline_avg - current_value) / baseline_avg
* 100,
                        "type": "performance_degradation"
                    })

            # 对于延迟，值越小越好
            elif metric in ["latency_p50", "latency_p99", "mean_ttft"]:
                if current_value > baseline_avg * (1 + self.regression_threshold):
                    regressions.append({
                        "metric": metric,
                        "current_value": current_value,
                        "baseline_avg": baseline_avg,
                        "regression_percent": (current_value - baseline_avg) / baseline_avg
* 100,
                        "type": "latency_regression"
                    })

        return regressions

    def save_results(self, results: Dict):
        """保存基准测试结果"""
        timestamp = datetime.now().isoformat()
        self.historical_data[timestamp] = results

        # 只保留最近30天的数据
        cutoff_date = datetime.now() - timedelta(days=30)
        filtered_data = {
            ts: data for ts, data in self.historical_data.items()
            if datetime.fromisoformat(ts) >= cutoff_date
        }

        with open(self.benchmark_results_file, 'w') as f:
            json.dump(filtered_data, f, indent=2)
```

```python
    def generate_report(self, results: Dict, regressions: List[Dict]) -> str:
        """生成性能报告"""
        report = ["# Performance Regression Report", ""]
        report.append(f"**Generated at:** {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
        report.append("")

        # 当前结果
        report.append("## Current Benchmark Results")
        for metric, value in results.items():
            if value is not None:
                report.append(f"- **{metric}**: {value:.4f}")
        report.append("")

        # 回归检测
        if regressions:
            report.append("## ⚠️ Performance Regressions Detected")
            for regression in regressions:
                report.append(f"### {regression['metric']}")
                report.append(f"- Current: {regression['current_value']:.4f}")
                report.append(f"- Baseline: {regression['baseline_avg']:.4f}")
                report.append(f"- Regression: {regression['regression_percent']:.2f}%")
                report.append(f"- Type: {regression['type']}")
                report.append("")
        else:
            report.append("## ✅ No Performance Regressions Detected")
            report.append("")

        return "\n".join(report)

if __name__ == "__main__":
    detector = PerformanceRegression()

    print("Running performance benchmarks...")
    results = detector.run_benchmark_suite()

    print("Detecting regressions...")
    regressions = detector.detect_regressions(results)

    print("Saving results...")
    detector.save_results(results)

    print("Generating report...")
    report = detector.generate_report(results, regressions)

    # 保存报告
    with open("performance_report.md", 'w') as f:
        f.write(report)

    print(report)

    # 如果检测到回归，退出码为1
    if regressions:
        print(f"\n❌ Detected {len(regressions)} performance regressions!")
        sys.exit(1)
    else:
        print("\n✅ No performance regressions detected.")
```

```
        sys.exit(0)
```

9. 报告生成脚本

.buildkite/scripts/generate-report.py

```python
#!/usr/bin/env python3
"""
CI测试报告生成器
"""
import json
import os
import sys
from pathlib import Path
from datetime import datetime
from typing import Dict, List

class ReportGenerator:
    def __init__(self):
        self.results_dir = Path("test_results")
        self.results_dir.mkdir(exist_ok=True)

    def collect_test_results(self) -> Dict:
        """收集所有测试结果"""
        results = {
            "fast_tests": self._collect_junit_results("fast_*.xml"),
            "standard_tests": self._collect_junit_results("std_*.xml"),
            "extended_tests": self._collect_junit_results("ext_*.xml"),
            "nightly_tests": self._collect_junit_results("nightly_*.xml"),
            "performance_results": self._load_performance_results(),
            "timing_analysis": self._load_timing_analysis()
        }
        return results

    def _collect_junit_results(self, pattern: str) -> Dict:
        """收集JUnit格式的测试结果"""
        import glob
        import xml.etree.ElementTree as ET

        results = {
            "total_tests": 0,
            "passed": 0,
            "failed": 0,
            "skipped": 0,
            "duration": 0.0,
            "failures": []
        }

        for xml_file in glob.glob(pattern):
            try:
                tree = ET.parse(xml_file)
                root = tree.getroot()

                # 解析testsuite元素
```

```
                for testsuite in root.findall('.//testsuite'):
                    results["total_tests"] += int(testsuite.get('tests', 0))
                    results["failed"] += int(testsuite.get('failures', 0))
                    results["skipped"] += int(testsuite.get('skipped', 0))
                    results["duration"] += float(testsuite.get('time', 0))

                # 收集失败的测试
                for testcase in root.findall('.//testcase'):
                    failure = testcase.find('failure')
                    if failure is not None:
                        results["failures"].append({
                            "name": testcase.get('name'),
                            "classname": testcase.get('classname'),
                            "message": failure.get('message', ''),
                            "type": failure.get('type', '')
                        })
            except Exception as e:
                print(f"Error parsing {xml_file}: {e}")

        results["passed"] = results["total_tests"] - results["failed"] - results["skipped"]
        return results

    def _load_performance_results(self) -> Dict:
        """加载性能测试结果"""
        perf_file = Path("performance_report.md")
        if perf_file.exists():
            return {"report_available": True, "file": str(perf_file)}
        return {"report_available": False}

    def _load_timing_analysis(self) -> Dict:
        """加载测试时间分析"""
        timing_file = Path("test_timing_results.json")
        if timing_file.exists():
            with open(timing_file, 'r') as f:
                data = json.load(f)
                if data:
                    latest_key = max(data.keys())
                    return data[latest_key]
        return {}


    def generate_html_report(self, results: Dict) -> str:


        """生成HTML格式的报告"""


        html = f"""


<!DOCTYPE html>


<html lang="zh-CN">


<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>vLLM CI测试报告</title>
    <style>
        body {{ font-family: Arial, sans-serif; margin: 20px; }}
        .header {{ background: #f5f5f5; padding: 20px; border-radius: 5px; }}
        .summary {{ display: flex; gap: 20px; margin: 20px 0; }}
        .card {{ background: white; border: 1px solid #ddd; padding: 15px; border-radius:
5px; flex: 1; }}
        .success {{ border-left: 4px solid #4CAF50; }}
        .warning {{ border-left: 4px solid #FF9800; }}
        .error {{ border-left: 4px solid #F44336; }}
        .test-details {{ margin: 20px 0; }}
        table {{ width: 100%; border-collapse: collapse; margin: 10px 0; }}
        th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}
        th {{ background-color: #f2f2f2; }}
        .failure {{ background-color: #ffebee; }}
    </style>
</head>
```

```
<body>
    <div class="header">
        <h1>vLLM CI测试报告</h1>
        <p>生成时间: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}</p>
        <p>构建ID: {os.environ.get('BUILDKITE_BUILD_NUMBER', 'N/A')}</p>
        <p>分支: {os.environ.get('BUILDKITE_BRANCH', 'N/A')}</p>
    </div>

    <div class="summary">
        {self._generate_test_summary_cards(results)}
    </div>

    <div class="test-details">
        <h2>测试详情</h2>
        {self._generate_test_details_table(results)}
    </div>

    <div class="performance-section">
        <h2>性能分析</h2>
```

```
            {self._generate_performance_section(results)}
        </div>

        <div class="timing-analysis">
            <h2>时间分析</h2>
            {self._generate_timing_analysis(results)}
        </div>
    </body>
    </html>
    """
            return html

    def _generate_test_summary_cards(self, results: Dict) -> str:
        """生成测试摘要卡片"""
        cards = []

        for test_type, data in results.items():
            if test_type in ["fast_tests", "standard_tests", "extended_tests",
"nightly_tests"]:
```

```
            total = data.get("total_tests", 0)

            passed = data.get("passed", 0)

            failed = data.get("failed", 0)



            card_class = "success" if failed == 0 else "error"

            card_html = f"""

            <div class="card {card_class}">

                <h3>{test_type.replace('_', ' ').title()}</h3>

                <p>总计: {total}</p>

                <p>通过: {passed}</p>

                <p>失败: {failed}</p>

                <p>耗时: {data.get('duration', 0):.2f}s</p>

            </div>

            """

            cards.append(card_html)



    return "".join(cards)



def _generate_test_details_table(self, results: Dict) -> str:
```

```
    """生成测试详情表格"""
    table_html = """
    <table>
        <thead>
            <tr>
                <th>测试类型</th>
                <th>总计</th>
                <th>通过</th>
                <th>失败</th>
                <th>跳过</th>
                <th>耗时(s)</th>
                <th>状态</th>
            </tr>
        </thead>
        <tbody>
    """

    for test_type, data in results.items():
```

```
            if test_type in ["fast_tests", "standard_tests", "extended_tests",
"nightly_tests"]:

                total = data.get("total_tests", 0)

                passed = data.get("passed", 0)

                failed = data.get("failed", 0)

                skipped = data.get("skipped", 0)

                duration = data.get("duration", 0)



                status = "✅ 通过" if failed == 0 else "❌ 失败"

                row_class = "" if failed == 0 else "failure"



                table_html += f"""

                <tr class="{row_class}">

                    <td>{test_type.replace('_', ' ').title()}</td>

                    <td>{total}</td>

                    <td>{passed}</td>

                    <td>{failed}</td>

                    <td>{skipped}</td>

                    <td>{duration:.2f}</td>
```

```
                <td>{status}</td>

            </tr>

            """



    table_html += "</tbody></table>"

    return table_html



def _generate_performance_section(self, results: Dict) -> str:

    """生成性能分析部分"""

    perf_data = results.get("performance_results", {})



    if perf_data.get("report_available"):

        return f"""

        <p>性能回归报告已生成: <a href="{perf_data['file']}">{perf_data['file']}</a></p>

        """

    else:

        return "<p>性能测试未运行或报告不可用</p>"



def _generate_timing_analysis(self, results: Dict) -> str:
```

```
    """生成时间分析部分"""
    timing_data = results.get("timing_analysis", {})

    if not timing_data:
        return "<p>时间分析数据不可用</p>"

    # 找出最慢的测试
    sorted_tests = sorted(timing_data.items(), key=lambda x: x[1], reverse=True)[:10]

    table_html = """
    <h3>最慢的10个测试</h3>
    <table>
        <thead>
            <tr>
                <th>测试名称</th>
                <th>耗时(s)</th>
            </tr>
        </thead>
        <tbody>
```

```
        """

        for test_name, duration in sorted_tests:
            table_html += f"""
            <tr>
                <td>{test_name}</td>
                <td>{duration:.2f}</td>
            </tr>
            """

        table_html += "</tbody></table>"
        return table_html

    def generate_markdown_report(self, results: Dict) -> str:
        """生成Markdown格式的报告"""
        report = [
            "# vLLM CI测试报告",
            "",
            f"**生成时间:** {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}",
```

```
            f"**构建ID:** {os.environ.get('BUILDKITE_BUILD_NUMBER', 'N/A')}",
            f"**分支:** {os.environ.get('BUILDKITE_BRANCH', 'N/A')}",
            "",
            "## 测试摘要",
            ""
        ]

        # 添加测试摘要表格
        report.append("| 测试类型 | 总计 | 通过 | 失败 | 跳过 | 耗时(s) | 状态 |")
        report.append("|---------|------|------|------|------|---------|------|")

        for test_type, data in results.items():
            if test_type in ["fast_tests", "standard_tests", "extended_tests",
"nightly_tests"]:
                total = data.get("total_tests", 0)
                passed = data.get("passed", 0)
                failed = data.get("failed", 0)
                skipped = data.get("skipped", 0)
                duration = data.get("duration", 0)
```

```
                status = "✅" if failed == 0 else "❌"

                report.append(f"| {test_type.replace('_', ' ').title()} | {total} |
{passed} | {failed} | {skipped} | {duration:.2f} | {status} |")

        # 添加失败测试详情
        report.extend(["", "## 失败测试详情", ""])

        has_failures = False
        for test_type, data in results.items():
            if test_type in ["fast_tests", "standard_tests", "extended_tests",
"nightly_tests"]:
                failures = data.get("failures", [])
                if failures:
                    has_failures = True
                    report.append(f"### {test_type.replace('_', ' ').title()}")
                    report.append("")
                    for failure in failures:
                        report.append(f"- **{failure['name']}**: {failure['message']}")
                    report.append("")
```

```
        if not has_failures:
            report.append("🎉 没有失败的测试！")

        return "\n".join(report)

if __name__ == "__main__":
    generator = ReportGenerator()

    print("收集测试结果...")
    results = generator.collect_test_results()

    print("生成HTML报告...")
    html_report = generator.generate_html_report(results)
    with open("nightly-report.html", 'w', encoding='utf-8') as f:
        f.write(html_report)

    print("生成Markdown报告...")
    md_report = generator.generate_markdown_report(results)
```

```
    with open("nightly-report.md", 'w', encoding='utf-8') as f:
        f.write(md_report)

    print("报告生成完成！")
    print(f"HTML报告: nightly-report.html")
    print(f"Markdown报告: nightly-report.md")
```

10. 测试标记自动化脚本

.buildkite/scripts/auto-mark-tests.py

```
#!/usr/bin/env python3
"""
自动为测试添加时间分层标记
"""
import ast
import os
import sys
from pathlib import Path
from typing import Dict, List, Set
```

```
import re



class TestMarker:

    def __init__(self):

        self.timing_data = self._load_timing_data()

        self.test_files = []



    def _load_timing_data(self) -> Dict:

        """加载历史测试时间数据"""

        timing_file = Path("test_timing_results.json")

        if timing_file.exists():

            import json

            with open(timing_file, 'r') as f:

                data = json.load(f)

                if data:

                    latest_key = max(data.keys())

                    return data[latest_key]

        return {}
```

```python
def discover_test_files(self, test_dir: str) -> List[Path]:
    """发现所有测试文件"""
    test_path = Path(test_dir)
    test_files = []

    for file_path in test_path.rglob("test_*.py"):
        test_files.append(file_path)

    return test_files

def analyze_test_file(self, file_path: Path) -> Dict:
    """分析测试文件，提取测试函数信息"""
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()

        tree = ast.parse(content)
        test_functions = []
```

```
for node in ast.walk(tree):
    if isinstance(node, ast.FunctionDef) and node.name.startswith('test_'):
        # 检查现有的装饰器
        existing_marks = set()
        for decorator in node.decorator_list:
            if isinstance(decorator, ast.Attribute):
                if isinstance(decorator.value, ast.Attribute):
                    # pytest.mark.xxx
                    if (decorator.value.attr == 'mark' and
                        isinstance(decorator.value.value, ast.Name) and
                        decorator.value.value.id == 'pytest'):
                        existing_marks.add(decorator.attr)

        test_functions.append({
            'name': node.name,
            'line': node.lineno,
            'existing_marks': existing_marks
        })
```

```
            return {
                'file_path': file_path,
                'content': content,
                'test_functions': test_functions
            }
        except Exception as e:
            print(f"Error analyzing {file_path}: {e}")
            return None


def estimate_test_duration(self, file_path: Path, test_name: str) -> str:
    """估算测试执行时间并返回对应标记"""
    # 使用历史数据
    test_key = f"{file_path}::{test_name}"
    if test_key in self.timing_data:
        duration = self.timing_data[test_key]
        if duration < 300:  # 5分钟
            return "fast"
        elif duration < 1200:  # 20分钟
            return "standard"
```

```
            elif duration < 3600:  # 60分钟
                return "extended"
            else:
                return "nightly"

        # 基于文件路径和测试名称的启发式规则
        file_str = str(file_path).lower()
        test_name_lower = test_name.lower()

        # 快速测试的模式
        if any(pattern in file_str for pattern in ['unit', 'utils', 'basic']):
            return "fast"

        # 夜间测试的模式
        if any(pattern in test_name_lower for pattern in ['stress', 'large', 'benchmark',
'e2e']):
            return "nightly"

        # 扩展测试的模式
```

```
        if any(pattern in file_str for pattern in ['integration', 'distributed',
'multi_gpu']):

            return "extended"



        # 默认为标准测试

        return "standard"




    def determine_module_marks(self, file_path: Path) -> List[str]:
        """根据文件路径确定模块标记"""
        file_str = str(file_path).lower()
        marks = []

        if 'entrypoints' in file_str:
            marks.append('entrypoints')
        if 'spec_decode' in file_str:
            marks.append('spec_decode')
        if 'kernels' in file_str:
            marks.append('kernels')
        if 'models' in file_str:
            marks.append('models')
        if 'distributed' in file_str:
            marks.append('multi_gpu')
        if any(pattern in file_str for pattern in ['gpu', 'cuda', 'rocm']):
            marks.append('gpu_intensive')
        if 'memory' in file_str:
            marks.append('memory_heavy')

        return marks

    def add_marks_to_test(self, file_info: Dict) -> str:
        """为测试文件添加标记"""
        content = file_info['content']
        file_path = file_info['file_path']
        test_functions = file_info['test_functions']

        # 检查是否已经导入pytest
        if 'import pytest' not in content:
            content = 'import pytest\n' + content

        lines = content.split('\n')
        new_lines = []
```

```
        for i, line in enumerate(lines):
            new_lines.append(line)

            # 检查是否是测试函数定义
            for test_func in test_functions:
                if i + 1 == test_func['line']:  # 函数定义行
                    # 确定需要添加的标记
                    time_mark = self.estimate_test_duration(file_path, test_func['name'])
                    module_marks = self.determine_module_marks(file_path)

                    all_marks = [time_mark] + module_marks
                    existing_marks = test_func['existing_marks']

                    # 只添加不存在的标记
                    new_marks = [mark for mark in all_marks if mark not in existing_marks]

                    if new_marks:
                        # 在函数定义前添加装饰器
                        indent = len(line) - len(line.lstrip())
                        for mark in new_marks:
                            decorator_line = ' ' * indent + f'@pytest.mark.{mark}'
                            new_lines.insert(-1, decorator_line)

        return '\n'.join(new_lines)

    def process_test_files(self, test_dir: str, dry_run: bool = True):
        """处理所有测试文件"""
        test_files = self.discover_test_files(test_dir)

        for file_path in test_files:
            print(f"Processing {file_path}...")

            file_info = self.analyze_test_file(file_path)
            if not file_info:
                continue

            new_content = self.add_marks_to_test(file_info)

            if dry_run:
                print(f"Would update {file_path}")
                # 显示差异
                original_lines = file_info['content'].split('\n')
                new_lines = new_content.split('\n')

                import difflib
                diff = difflib.unified_diff(
                    original_lines, new_lines,
                    fromfile=str(file_path), tofile=str(file_path),
                    lineterm=''
                )
                for line in diff:
                    print(line)
            else:
                # 实际写入文件
                with open(file_path, 'w', encoding='utf-8') as f:
                    f.write(new_content)
                print(f"Updated {file_path}")
```

```python
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="自动为测试添加时间分层标记")
    parser.add_argument("test_dir", help="测试目录路径")
    parser.add_argument("--dry-run", action="store_true", help="只显示变更，不实际修改文件")

    args = parser.parse_args()

    marker = TestMarker()


    marker.process_test_files(args.test_dir, args.dry_run)
```

11. CI流水线触发器脚本

.buildkite/scripts/pipeline-trigger.py

```python
#!/usr/bin/env python3
"""
智能CI流水线触发器
"""
import os
import sys
import json
import subprocess
from pathlib import Path
from typing import Dict, List, Set

class PipelineTrigger:
    def __init__(self):
        self.branch = os.environ.get('BUILDKITE_BRANCH', 'main')
        self.commit = os.environ.get('BUILDKITE_COMMIT', '')
        self.pull_request = os.environ.get('BUILDKITE_PULL_REQUEST', 'false')

    def get_changed_files(self) -> List[str]:
        """获取变更的文件列表"""
        if self.pull_request != 'false':
            # PR模式：比较与目标分支的差异
            cmd = ["git", "diff", "--name-only", "origin/main...HEAD"]
        else:
            # 推送模式：比较与上一次提交的差异
            cmd = ["git", "diff", "--name-only", "HEAD~1", "HEAD"]

        try:
            result = subprocess.run(cmd, capture_output=True, text=True, check=True)
            return result.stdout.strip().split('\n') if result.stdout.strip() else []
        except subprocess.CalledProcessError:
            return []

    def determine_test_scope(self, changed_files: List[str]) -> Dict[str, bool]:
        """根据变更文件确定测试范围"""
        scope = {
            "fast": True,  # 总是运行快速测试
```

```
        "standard": False,
        "extended": False,
        "nightly": False,
        "performance": False
    }

    # 如果没有变更文件，运行标准测试
    if not changed_files:
        scope["standard"] = True
        return scope

    # 分析变更文件类型
    core_changes = False
    test_changes = False
    doc_changes = False

    for file_path in changed_files:
        if file_path.startswith('vllm/'):
            core_changes = True
        elif file_path.startswith('tests/'):
            test_changes = True
        elif file_path.startswith('docs/') or file_path.endswith('.md'):
            doc_changes = True

    # 根据变更类型确定测试范围
    if core_changes:
        scope["standard"] = True
        # 核心变更需要扩展测试
        if any(path.startswith(('vllm/model_executor/', 'vllm/engine/', 'csrc/'))
               for path in changed_files):
            scope["extended"] = True

    if test_changes:
        scope["standard"] = True

    # 主分支或发布分支运行性能测试
    if self.branch in ['main', 'release']:
        scope["performance"] = True

    # 夜间构建或手动触发运行全量测试
    if (os.environ.get('BUILDKITE_SOURCE') == 'schedule' or
        os.environ.get('FORCE_FULL_TESTS') == 'true'):
        scope.update({
            "standard": True,
            "extended": True,
            "nightly": True,
            "performance": True
        })

    return scope

def generate_pipeline_steps(self, test_scope: Dict[str, bool]) -> List[Dict]:
    """生成流水线步骤"""
    steps = []

    # 快速测试步骤
    if test_scope["fast"]:
```

```
            steps.extend([
                {
                    "label": "🚀 Fast Check - Documentation",
                    "key": "fast-docs",
                    "command": ".buildkite/scripts/test-runner.sh fast docs 5 1",
                    "timeout_in_minutes": 5
                },
                {
                    "label": "🚀 Fast Check - Core Units",
                    "key": "fast-core",
                    "command": ".buildkite/scripts/test-runner.sh fast core 5 1",
                    "timeout_in_minutes": 5
                }
            ])

        # 标准测试步骤
        if test_scope["standard"]:
            steps.extend([
                {
                    "label": "⚡ Standard - Entrypoints",
                    "key": "std-entrypoints",
                    "depends_on": ["fast-core"],
                    "command": ".buildkite/scripts/test-runner.sh standard entrypoints 20
1",
                    "timeout_in_minutes": 20,
                    "parallelism": 2
                },
                {
                    "label": "⚡ Standard - Models",
                    "key": "std-models",
                    "depends_on": ["fast-core"],
                    "command": ".buildkite/scripts/test-runner.sh standard models 20 1",
                    "timeout_in_minutes": 20,
                    "parallelism": 3
                }
            ])

        # 扩展测试步骤
        if test_scope["extended"]:
            steps.extend([
                {
                    "label": "🔧 Extended - Multi-GPU",
                    "key": "ext-multi-gpu",
                    "depends_on": ["std-models"],
                    "command": ".buildkite/scripts/test-runner.sh extended multi_gpu 60 1",
                    "timeout_in_minutes": 60,
                    "agents": {"queue": "gpu-multi"}
                },
                {
                    "label": "🔧 Extended - Large Models",
                    "key": "ext-large-models",
                    "depends_on": ["std-models"],
                    "command": ".buildkite/scripts/test-runner.sh extended large_model 60
1",
                    "timeout_in_minutes": 60,
                    "agents": {"queue": "gpu-large"}
                }
```

```
            ])

        # 夜间测试步骤
        if test_scope["nightly"]:
            steps.extend([
                {
                    "label": "🌙 Nightly - Stress Tests",
                    "key": "nightly-stress",
                    "command": ".buildkite/scripts/test-runner.sh nightly stress 180 0",
                    "timeout_in_minutes": 180
                }
            ])

        # 性能测试步骤
        if test_scope["performance"]:
            steps.append({
                "label": "📊 Performance Regression",
                "key": "performance",
                "depends_on": ["std-models"] if test_scope["standard"] else None,
                "command": "python3 .buildkite/scripts/performance-regression.py",
                "timeout_in_minutes": 120
            })

        return steps

    def output_pipeline(self, steps: List[Dict]):
        """输出流水线配置"""
        pipeline = {"steps": steps}

        # 输出到文件或标准输出
        output_file = os.environ.get('PIPELINE_OUTPUT_FILE')
        if output_file:
            with open(output_file, 'w') as f:
                json.dump(pipeline, f, indent=2)
        else:
            print(json.dumps(pipeline, indent=2))

if __name__ == "__main__":
    trigger = PipelineTrigger()

    print("Analyzing changed files...", file=sys.stderr)
    changed_files = trigger.get_changed_files()
    print(f"Changed files: {changed_files}", file=sys.stderr)

    print("Determining test scope...", file=sys.stderr)
    test_scope = trigger.determine_test_scope(changed_files)
    print(f"Test scope: {test_scope}", file=sys.stderr)

    print("Generating pipeline steps...", file=sys.stderr)
    steps = trigger.generate_pipeline_steps(test_scope)

    print("Outputting pipeline configuration...", file=sys.stderr)


    trigger.output_pipeline(steps)
```

12. 测试结果聚合脚本

.buildkite/scripts/aggregate-results.py

```python
#!/usr/bin/env python3
"""
测试结果聚合和分析工具
"""
import json
import glob
import sys
from pathlib import Path
from typing import Dict, List
import xml.etree.ElementTree as ET

class ResultsAggregator:
    def __init__(self):
        self.results = {
            "summary": {},
            "details": {},
            "trends": {}
        }

    def collect_junit_results(self) -> Dict:
        """收集JUnit XML结果"""
        junit_files = glob.glob("test-results/*.xml")

        total_results = {
            "total_tests": 0,
            "passed": 0,
            "failed": 0,
            "skipped": 0,
            "duration": 0.0,
            "test_suites": []
        }

        for xml_file in junit_files:
            try:
                tree = ET.parse(xml_file)
                root = tree.getroot()

                suite_info = {
                    "name": Path(xml_file).stem,
                    "tests": 0,
                    "failures": 0,
                    "skipped": 0,
                    "time": 0.0,
                    "test_cases": []
                }

                for testsuite in root.findall('.//testsuite'):
                    suite_info["tests"] += int(testsuite.get('tests', 0))
                    suite_info["failures"] += int(testsuite.get('failures', 0))
                    suite_info["skipped"] += int(testsuite.get('skipped', 0))
                    suite_info["time"] += float(testsuite.get('time', 0))
```

```
                # 收集测试用例详情
                for testcase in root.findall('.//testcase'):
                    case_info = {
                        "name": testcase.get('name'),
                        "classname": testcase.get('classname'),
                        "time": float(testcase.get('time', 0)),
                        "status": "passed"
                    }

                    if testcase.find('failure') is not None:
                        case_info["status"] = "failed"
                        case_info["failure"] = testcase.find('failure').get('message', '')
                    elif testcase.find('skipped') is not None:
                        case_info["status"] = "skipped"
                   suite_info["test_cases"].append(case_info)

                total_results["test_suites"].append(suite_info)
                total_results["total_tests"] += suite_info["tests"]
                total_results["failed"] += suite_info["failures"]
                total_results["skipped"] += suite_info["skipped"]
                total_results["duration"] += suite_info["time"]

            except Exception as e:
                print(f"Error parsing {xml_file}: {e}")

        total_results["passed"] = total_results["total_tests"] - total_results["failed"] -
total_results["skipped"]
        return total_results

    def analyze_test_trends(self, current_results: Dict) -> Dict:
        """分析测试趋势"""
        trends = {
            "duration_trend": "stable",
            "failure_rate_trend": "stable",
            "slowest_tests": [],
            "most_failed_tests": []
        }

        # 找出最慢的测试
        all_test_cases = []
        for suite in current_results.get("test_suites", []):
            all_test_cases.extend(suite.get("test_cases", []))

        # 按执行时间排序
        sorted_by_time = sorted(all_test_cases, key=lambda x: x.get("time", 0),
reverse=True)
        trends["slowest_tests"] = sorted_by_time[:10]

        # 统计失败的测试
        failed_tests = [test for test in all_test_cases if test.get("status") == "failed"]
        trends["most_failed_tests"] = failed_tests

        return trends

    def generate_summary_report(self, results: Dict, trends: Dict) -> str:
        """生成摘要报告"""
        report = []
```

```
        report.append("# CI测试结果摘要")
        report.append("")

        # 基本统计
        total_tests = results.get("total_tests", 0)
        passed = results.get("passed", 0)
        failed = results.get("failed", 0)
        skipped = results.get("skipped", 0)
        duration = results.get("duration", 0)

        report.append(f"**总测试数:** {total_tests}")
        report.append(f"**通过:** {passed}")
        report.append(f"**失败:** {failed}")
        report.append(f"**跳过:** {skipped}")
        report.append(f"**总耗时:** {duration:.2f}秒")
        report.append("")

        # 成功率
        success_rate = (passed / total_tests * 100) if total_tests > 0 else 0
        report.append(f"**成功率:** {success_rate:.1f}%")
        report.append("")

        # 最慢的测试
        if trends.get("slowest_tests"):
            report.append("## 最慢的测试")
            for i, test in enumerate(trends["slowest_tests"][:5], 1):
                report.append(f"{i}. {test['classname']}::{test['name']} -
{test['time']:.2f}s")
            report.append("")

        # 失败的测试
        if trends.get("most_failed_tests"):
            report.append("## 失败的测试")
            for test in trends["most_failed_tests"]:
                report.append(f"- {test['classname']}::{test['name']}")
                if test.get("failure"):
                    report.append(f"  错误: {test['failure'][:100]}...")
            report.append("")

        return "\n".join(report)

    def save_results(self, results: Dict, trends: Dict):
        """保存结果到文件"""
        output_data = {
            "timestamp": datetime.now().isoformat(),
            "results": results,
            "trends": trends
        }

        with open("aggregated_results.json", 'w') as f:
            json.dump(output_data, f, indent=2)

        # 生成摘要报告
        summary = self.generate_summary_report(results, trends)
        with open("test_summary.md", 'w') as f:
            f.write(summary)
```

```python
if __name__ == "__main__":
    from datetime import datetime

    aggregator = ResultsAggregator()

    print("收集JUnit测试结果...")
    results = aggregator.collect_junit_results()

    print("分析测试趋势...")
    trends = aggregator.analyze_test_trends(results)

    print("保存结果...")
    aggregator.save_results(results, trends)

    print("聚合完成！")
    print(f"总测试数: {results['total_tests']}")
    print(f"通过: {results['passed']}")
    print(f"失败: {results['failed']}")


    print(f"总耗时: {results['duration']:.2f}秒")
```

13. 智能测试重试脚本

.buildkite/scripts/smart-retry.py

```python
#!/usr/bin/env python3
"""
智能测试重试工具 - 基于失败模式决定重试策略
"""
import json
import subprocess
import sys
import time
from pathlib import Path
from typing import Dict, List, Tuple
import re

class SmartRetry:
    def __init__(self):
        self.retry_patterns = {
            "network_timeout": {
                "patterns": ["timeout", "connection", "network", "dns"],
                "max_retries": 3,
                "delay": 30
            },
            "resource_exhaustion": {
                "patterns": ["out of memory", "cuda out of memory", "disk space"],
                "max_retries": 2,
                "delay": 60
            },
            "flaky_test": {
                "patterns": ["race condition", "timing", "random"],
                "max_retries": 2,
                "delay": 10
```

```
        },
        "infrastructure": {
            "patterns": ["buildkite", "agent", "docker"],
            "max_retries": 1,
            "delay": 120
        }
    }

def analyze_failure(self, error_output: str) -> str:
    """分析失败原因并返回失败类型"""
    error_lower = error_output.lower()

    for failure_type, config in self.retry_patterns.items():
        for pattern in config["patterns"]:
            if pattern in error_lower:
                return failure_type

    return "unknown"

def should_retry(self, failure_type: str, attempt: int) -> Tuple[bool, int]:
    """判断是否应该重试以及延迟时间"""
    if failure_type not in self.retry_patterns:
        return False, 0

    config = self.retry_patterns[failure_type]
    if attempt >= config["max_retries"]:
        return False, 0

    return True, config["delay"]

def run_with_smart_retry(self, command: List[str], max_attempts: int = 3) -> int:
    """使用智能重试运行命令"""
    attempt = 0

    while attempt < max_attempts:
        attempt += 1
        print(f"Attempt {attempt}/{max_attempts}: {' '.join(command)}")

        try:
            result = subprocess.run(
                command,
                capture_output=True,
                text=True,
                check=True
            )
            print("Command succeeded!")
            return 0

        except subprocess.CalledProcessError as e:
            print(f"Command failed with exit code {e.returncode}")
            print(f"Error output: {e.stderr}")

            # 分析失败原因
            failure_type = self.analyze_failure(e.stderr)
            print(f"Detected failure type: {failure_type}")

            # 决定是否重试
```

```
                should_retry, delay = self.should_retry(failure_type, attempt)

                if should_retry and attempt < max_attempts:
                    print(f"Retrying in {delay} seconds...")
                    time.sleep(delay)
                else:
                    print("No more retries, giving up")
                    return e.returncode

        return 1

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: smart-retry.py <command> [args...]")
        sys.exit(1)

    retry_tool = SmartRetry()
    command = sys.argv[1:]

    exit_code = retry_tool.run_with_smart_retry(command)


    sys.exit(exit_code)
```

14. 测试环境清理脚本

.buildkite/scripts/cleanup-test-env.py

```
#!/usr/bin/env python3
"""
测试环境清理工具
"""
import os
import shutil
import subprocess
import sys
import time
from pathlib import Path
from typing import List

class TestEnvironmentCleaner:
    def __init__(self):
        self.cleanup_patterns = [
            "*.pyc",
            "__pycache__",
            ".pytest_cache",
            "*.log",
            "test_results",
            "coverage_reports",
            ".coverage*",
            "*.tmp"
        ]

    def cleanup_python_cache(self):
        """清理Python缓存文件"""
```

```
        print("Cleaning Python cache files...")

        for pattern in ["**/__pycache__", "**/*.pyc", "**/*.pyo"]:
            for path in Path(".").glob(pattern):
                if path.is_file():
                    path.unlink()
                    print(f"Removed file: {path}")
                elif path.is_dir():
                    shutil.rmtree(path)
                    print(f"Removed directory: {path}")

    def cleanup_test_artifacts(self):
        """清理测试产生的文件"""
        print("Cleaning test artifacts...")

        artifacts = [
            "test_results",
            "coverage_reports",
            ".coverage",
            ".pytest_cache",
            "htmlcov",
            "*.log",
            "*.tmp"
        ]

        for pattern in artifacts:
            for path in Path(".").glob(pattern):
                if path.exists():
                    if path.is_file():
                        path.unlink()
                        print(f"Removed file: {path}")
                    elif path.is_dir():
                        shutil.rmtree(path)
                        print(f"Removed directory: {path}")

    def cleanup_gpu_processes(self):
        """清理GPU进程"""
        print("Cleaning GPU processes...")

        try:
            # 查找使用GPU的Python进程
            result = subprocess.run(
                ["nvidia-smi", "--query-compute-apps=pid,process_name",
"--format=csv,noheader,nounits"],
                capture_output=True,
                text=True
            )

            if result.returncode == 0:
                lines = result.stdout.strip().split('\n')
                for line in lines:
                    if line.strip():
                        parts = line.split(', ')
                        if len(parts) >= 2:
                            pid = parts[0].strip()
                            process_name = parts[1].strip()
```

```
                        if 'python' in process_name.lower():
                            print(f"Killing GPU process: {pid} ({process_name})")
                            try:
                                subprocess.run(["kill", "-9", pid], check=True)
                            except subprocess.CalledProcessError:
                                print(f"Failed to kill process {pid}")

        except FileNotFoundError:
            print("nvidia-smi not found, skipping GPU cleanup")
        except Exception as e:
            print(f"Error during GPU cleanup: {e}")

    def cleanup_docker_containers(self):
        """清理Docker容器"""
        print("Cleaning Docker containers...")

        try:
            # 停止所有测试相关的容器
            result = subprocess.run(
                ["docker", "ps", "-q", "--filter", "label=test=vllm"],
                capture_output=True,
                text=True
            )

            if result.returncode == 0 and result.stdout.strip():
                container_ids = result.stdout.strip().split('\n')
                for container_id in container_ids:
                    print(f"Stopping container: {container_id}")
                    subprocess.run(["docker", "stop", container_id])
                    subprocess.run(["docker", "rm", container_id])

        except FileNotFoundError:
            print("Docker not found, skipping container cleanup")
        except Exception as e:
            print(f"Error during Docker cleanup: {e}")

    def cleanup_shared_memory(self):
        """清理共享内存"""
        print("Cleaning shared memory...")

        try:
            # 清理/dev/shm中的测试文件
            shm_path = Path("/dev/shm")
            if shm_path.exists():
                for item in shm_path.glob("vllm_test_*"):
                    if item.is_file():
                        item.unlink()
                        print(f"Removed shared memory file: {item}")
        except Exception as e:
            print(f"Error during shared memory cleanup: {e}")

    def cleanup_all(self):
        """执行完整清理"""
        print("Starting comprehensive test environment cleanup...")

        self.cleanup_python_cache()
        self.cleanup_test_artifacts()
```

```
        self.cleanup_gpu_processes()
        self.cleanup_docker_containers()
        self.cleanup_shared_memory()

        print("Cleanup completed!")

if __name__ == "__main__":
    cleaner = TestEnvironmentCleaner()


    cleaner.cleanup_all()
```

15. 最终的主配置文件更新pyproject.toml（更新测试配置部分）

```
[tool.pytest.ini_options]
minversion = "6.0"
addopts = [
    "-ra",
    "--strict-markers",
    "--strict-config",
    "--cov=vllm",
    "--cov-report=term-missing:skip-covered",
    "--cov-report=html:htmlcov",
    "--cov-report=xml",
    "--junit-xml=test_results/junit.xml",
]
testpaths = ["tests"]
markers = [
    "fast: 快速测试 (< 5分钟)",
    "standard: 标准测试 (5-20分钟)",
    "extended: 扩展测试 (20-60分钟)",
    "nightly: 夜间测试 (> 60分钟)",
    "gpu_intensive: GPU密集型测试",
    "memory_heavy: 内存密集型测试",
    "multi_gpu: 多GPU测试",
    "entrypoints: 入口点测试",
    "spec_decode: 规范解码测试",
    "kernels: 内核测试",
    "models: 模型测试",
    "benchmarks: 基准测试",
    "stress: 压力测试",
    "large_model: 大模型测试",
    "e2e: 端到端测试",
    "amd: AMD硬件测试",
    "core_model: 核心模型测试",
    "cpu_model: CPU模型测试",
    "distributed: 分布式测试",
    "skip_global_cleanup: 跳过全局清理",
    "optional: 可选测试",
    "split: 分片测试",
    "skip_v1: 跳过V1测试"
]
timeout = 300
```

```
timeout_method = "thread"
```

16. 测试执行脚本包装器

.buildkite/scripts/run-tests.sh

```
#!/bin/bash
# 主测试执行脚本 - 统一入口点

set -e

# 默认参数
TEST_TYPE="standard"
TEST_FILTER=""
TIMEOUT_MINUTES=20
PARALLEL_JOBS=1
SHARD_ID=""
NUM_SHARDS=""

# 解析命令行参数
while [[ $# -gt 0 ]]; do
    case $1 in
        --type)
            TEST_TYPE="$2"
            shift 2
            ;;
        --filter)
            TEST_FILTER="$2"
            shift 2
            ;;
        --timeout)
            TIMEOUT_MINUTES="$2"
            shift 2
            ;;
        --parallel)
            PARALLEL_JOBS="$2"
            shift 2
            ;;
        --shard-id)
            SHARD_ID="$2"
            shift 2
            ;;
        --num-shards)
            NUM_SHARDS="$2"
            shift 2
            ;;
        *)
            echo "Unknown option $1"
            exit 1
            ;;
    esac
done

# 设置环境变量
```

```
export VLLM_WORKER_MULTIPROC_METHOD=spawn
export CUDA_VISIBLE_DEVICES=${CUDA_VISIBLE_DEVICES:-"0"}

# 创建结果目录
mkdir -p test_results

# 构建pytest命令
PYTEST_CMD="pytest -v -s"

# 添加标记过滤
case $TEST_TYPE in
    "fast")
        PYTEST_CMD="$PYTEST_CMD -m 'fast'"
        ;;
    "standard")
        PYTEST_CMD="$PYTEST_CMD -m 'standard'"
        ;;
    "extended")
        PYTEST_CMD="$PYTEST_CMD -m 'extended'"
        ;;
    "nightly")
        PYTEST_CMD="$PYTEST_CMD -m 'nightly'"
        ;;
    *)
        echo "Unknown test type: $TEST_TYPE"
        exit 1
        ;;
esac

# 添加额外过滤器
if [[ -n "$TEST_FILTER" ]]; then
    PYTEST_CMD="$PYTEST_CMD and $TEST_FILTER"
fi

# 添加分片支持
if [[ -n "$SHARD_ID" && -n "$NUM_SHARDS" ]]; then
    PYTEST_CMD="$PYTEST_CMD --shard-id=$SHARD_ID --num-shards=$NUM_SHARDS"
fi

# 添加超时和输出配置
TIMEOUT_SECONDS=$((TIMEOUT_MINUTES * 60))
PYTEST_CMD="$PYTEST_CMD --timeout=$TIMEOUT_SECONDS"
PYTEST_CMD="$PYTEST_CMD --junit-xml=test_results/${TEST_TYPE}_${SHARD_ID:-0}.xml"
PYTEST_CMD="$PYTEST_CMD
--cov-report=xml:test_results/coverage_${TEST_TYPE}_${SHARD_ID:-0}.xml"

# 执行测试前清理
echo "Cleaning test environment..."
python3 .buildkite/scripts/cleanup-test-env.py

# 执行测试
echo "Running tests: $PYTEST_CMD"
if [[ "$PARALLEL_JOBS" -gt 1 ]]; then
    # 并行执行
    $PYTEST_CMD -n $PARALLEL_JOBS
else
    # 串行执行
```

```
    $PYTEST_CMD
fi

# 收集结果
echo "Collecting test results..."
python3 .buildkite/scripts/aggregate-results.py


echo "Test execution completed!"
```

17. 测试依赖管理脚本

.buildkite/scripts/setup-test-deps.py

```python
#!/usr/bin/env python3
"""
测试依赖管理和安装脚本
"""
import subprocess
import sys
import os
from pathlib import Path
from typing import List, Dict

class TestDependencyManager:
    def __init__(self):
        self.base_requirements = [
            "pytest>=7.0.0",
            "pytest-timeout>=2.1.0",
            "pytest-xdist>=3.0.0",
            "pytest-cov>=4.0.0",
            "pytest-mock>=3.10.0"
        ]

        self.test_type_deps = {
            "entrypoints": [
                "requests>=2.28.0",
                "openai>=1.0.0",
                "fastapi>=0.100.0"
            ],
            "models": [
                "transformers>=4.30.0",
                "torch>=2.0.0",
                "torchvision>=0.15.0"
            ],
            "kernels": [
                "triton>=2.0.0",
                "flash-attn>=2.0.0"
            ],
            "benchmarks": [
                "matplotlib>=3.5.0",
                "pandas>=1.5.0",
                "seaborn>=0.11.0"
            ]
```

```
    }

def detect_test_types(self, test_paths: List[str]) -> List[str]:
    """检测需要运行的测试类型"""
    test_types = set()

    for path in test_paths:
        if "entrypoints" in path:
            test_types.add("entrypoints")
        elif "models" in path:
            test_types.add("models")
        elif "kernels" in path:
            test_types.add("kernels")
        elif "benchmarks" in path:
            test_types.add("benchmarks")

    return list(test_types)

def install_base_dependencies(self):
    """安装基础测试依赖"""
    print("Installing base test dependencies...")

    for req in self.base_requirements:
        try:
            subprocess.run([
                sys.executable, "-m", "pip", "install", req
            ], check=True)
            print(f"✓ Installed {req}")
        except subprocess.CalledProcessError as e:
            print(f"✗ Failed to install {req}: {e}")
            return False

    return True

def install_test_type_dependencies(self, test_types: List[str]):
    """安装特定测试类型的依赖"""
    print(f"Installing dependencies for test types: {test_types}")

    for test_type in test_types:
        if test_type in self.test_type_deps:
            print(f"Installing {test_type} dependencies...")

            for req in self.test_type_deps[test_type]:
                try:
                    subprocess.run([
                        sys.executable, "-m", "pip", "install", req
                    ], check=True)
                    print(f"✓ Installed {req}")
                except subprocess.CalledProcessError as e:
                    print(f"✗ Failed to install {req}: {e}")

def setup_gpu_dependencies(self):
    """设置GPU相关依赖"""
    if os.environ.get("CUDA_VISIBLE_DEVICES"):
        print("Setting up GPU dependencies...")

        gpu_deps = [
```

```
                "nvidia-ml-py>=11.0.0",
                "pynvml>=11.0.0"
            ]

            for req in gpu_deps:
                try:
                    subprocess.run([
                        sys.executable, "-m", "pip", "install", req
                    ], check=True)
                    print(f"✓ Installed {req}")
                except subprocess.CalledProcessError as e:
                    print(f"✗ Failed to install {req}: {e}")

    def verify_installation(self) -> bool:
        """验证依赖安装"""
        print("Verifying test dependencies...")

        try:
            import pytest
            import pytest_timeout
            import pytest_xdist
            import pytest_cov
            print("✓ All base dependencies verified")
            return True
        except ImportError as e:
            print(f"✗ Missing dependency: {e}")
            return False

if __name__ == "__main__":
    manager = TestDependencyManager()

    # 获取测试路径参数
    test_paths = sys.argv[1:] if len(sys.argv) > 1 else ["tests/"]

    # 检测测试类型
    test_types = manager.detect_test_types(test_paths)
    print(f"Detected test types: {test_types}")

    # 安装依赖
    if not manager.install_base_dependencies():
        sys.exit(1)

    manager.install_test_type_dependencies(test_types)
    manager.setup_gpu_dependencies()

    # 验证安装
    if not manager.verify_installation():
        sys.exit(1)



    print("✓ All test dependencies installed successfully!")
```

18. CI状态监控脚本

.buildkite/scripts/ci-monitor.py

```python
#!/usr/bin/env python3
"""
CI状态监控和通知脚本
"""
import json
import requests
import os
import sys
from datetime import datetime, timedelta
from typing import Dict, List

class CIMonitor:
    def __init__(self):
        self.buildkite_token = os.environ.get('BUILDKITE_API_TOKEN')
        self.org_slug = os.environ.get('BUILDKITE_ORGANIZATION_SLUG', 'vllm-project')
        self.pipeline_slug = os.environ.get('BUILDKITE_PIPELINE_SLUG', 'vllm')
        self.webhook_url = os.environ.get('SLACK_WEBHOOK_URL')

    def get_recent_builds(self, hours: int = 24) -> List[Dict]:
        """获取最近的构建信息"""
        if not self.buildkite_token:
            print("No Buildkite API token provided")
            return []

        url =
f"https://api.buildkite.com/v2/organizations/{self.org_slug}/pipelines/{self.pipeline_slug}
/builds"
        headers = {"Authorization": f"Bearer {self.buildkite_token}"}

        try:
            response = requests.get(url, headers=headers)
            response.raise_for_status()
            builds = response.json()

            # 过滤最近的构建
            cutoff_time = datetime.now() - timedelta(hours=hours)
            recent_builds = []

            for build in builds:
                created_at = datetime.fromisoformat(build['created_at'].replace('Z',
'+00:00'))
                if created_at >= cutoff_time:
                    recent_builds.append(build)

            return recent_builds
        except Exception as e:
            print(f"Error fetching builds: {e}")
            return []

    def analyze_build_trends(self, builds: List[Dict]) -> Dict:
        """分析构建趋势"""
        if not builds:
            return {}

        total_builds = len(builds)
```

```
        passed_builds = len([b for b in builds if b['state'] == 'passed'])
        failed_builds = len([b for b in builds if b['state'] == 'failed'])

        # 计算平均构建时间
        completed_builds = [b for b in builds if b['finished_at']]
        avg_duration = 0

        if completed_builds:
            durations = []
            for build in completed_builds:
                start = datetime.fromisoformat(build['created_at'].replace('Z', '+00:00'))
                end = datetime.fromisoformat(build['finished_at'].replace('Z', '+00:00'))
                durations.append((end - start).total_seconds())

            avg_duration = sum(durations) / len(durations)

        # 分析失败原因
        failure_reasons = {}
        for build in builds:
            if build['state'] == 'failed':
                # 这里可以进一步分析具体的失败步骤
                failure_reasons[build['id']] = build.get('message', 'Unknown failure')

        return {
            "total_builds": total_builds,
            "passed_builds": passed_builds,
            "failed_builds": failed_builds,
            "success_rate": (passed_builds / total_builds * 100) if total_builds > 0 else
0,
            "avg_duration_minutes": avg_duration / 60,
            "failure_reasons": failure_reasons
        }

    def generate_status_report(self, trends: Dict) -> str:
        """生成状态报告"""
        if not trends:
            return "No build data available"

        report = [
            "# CI Status Report",
            f"**Generated:** {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}",
            "",
            "## Build Statistics (Last 24 hours)",
            f"- **Total Builds:** {trends['total_builds']}",
            f"- **Passed:** {trends['passed_builds']}",
            f"- **Failed:** {trends['failed_builds']}",


            f"- **Success Rate:** {trends['success_rate']:.1f}


            f"- **Average Duration:** {trends['avg_duration_minutes']:.1f} minutes",


            ""
```

```
]


# 失败分析
if trends['failure_reasons']:
    report.append("## Recent Failures")
    for build_id, reason in list(trends['failure_reasons'].items())[:5]:
        report.append(f"- Build {build_id}: {reason}")
    report.append("")


# 健康状态
success_rate = trends['success_rate']
if success_rate >= 90:
    status = "🟢 Healthy"
elif success_rate >= 70:
    status = "🟡 Warning"
else:
    status = "🔴 Critical"


report.append(f"## Overall Status: {status}")
```

```
    return "\n".join(report)



def send_slack_notification(self, message: str):
    """发送Slack通知"""
    if not self.webhook_url:
        print("No Slack webhook URL configured")
        return

    payload = {
        "text": "vLLM CI Status Update",
        "blocks": [
            {
                "type": "section",
                "text": {
                    "type": "mrkdwn",
                    "text": message
                }
            }
```

```
            ]
        }

        try:
            response = requests.post(self.webhook_url, json=payload)
            response.raise_for_status()
            print("Slack notification sent successfully")
        except Exception as e:
            print(f"Failed to send Slack notification: {e}")

def check_and_alert(self):
    """检查CI状态并发送警报"""
    builds = self.get_recent_builds()
    trends = self.analyze_build_trends(builds)

    if not trends:
        return

    # 生成报告
```

```
        report = self.generate_status_report(trends)
        print(report)

        # 检查是否需要发送警报
        success_rate = trends.get('success_rate', 100)
        if success_rate < 70:  # 成功率低于70%时发送警报
            alert_message = f"⚠️ CI Success Rate Alert: {success_rate:.1f}%\n\n{report}"
            self.send_slack_notification(alert_message)

if __name__ == "__main__":
    monitor = CIMonitor()
    monitor.check_and_alert()
```

19. Makefile

Makefile

```
# vLLM CI改造 - 统一构建和测试管理

.PHONY: help install-deps setup-ci test-fast test-standard test-extended test-nightly
clean-env
```

```
# 默认目标

help:
	@echo "vLLM CI改造工具"
	@echo ""
	@echo "可用命令:"
	@echo "  install-deps     - 安装测试依赖"
	@echo "  setup-ci         - 设置CI环境"
	@echo "  test-fast        - 运行快速测试"
	@echo "  test-standard    - 运行标准测试"
	@echo "  test-extended    - 运行扩展测试"
	@echo "  test-nightly     - 运行夜间测试"
	@echo "  mark-tests       - 自动标记测试"
	@echo "  clean-env        - 清理测试环境"
	@echo "  monitor-ci       - 监控CI状态"
	@echo "  generate-report  - 生成测试报告"

# 安装依赖

install-deps:
	@echo "Installing test dependencies..."
```

```
	python3 .buildkite/scripts/setup-test-deps.py tests/

# 设置CI环境
setup-ci:
	@echo "Setting up CI environment..."
	mkdir -p test_results
	chmod +x .buildkite/scripts/*.sh
	chmod +x .buildkite/scripts/*.py

# 快速测试
test-fast:
	@echo "Running fast tests..."
	.buildkite/scripts/run-tests.sh --type fast --timeout 5

# 标准测试
test-standard:
	@echo "Running standard tests..."
	.buildkite/scripts/run-tests.sh --type standard --timeout 20
```

```
# 扩展测试
test-extended:
	@echo "Running extended tests..."
	.buildkite/scripts/run-tests.sh --type extended --timeout 60

# 夜间测试
test-nightly:
	@echo "Running nightly tests..."
	.buildkite/scripts/run-tests.sh --type nightly --timeout 180

# 自动标记测试
mark-tests:
	@echo "Auto-marking tests..."
	python3 .buildkite/scripts/auto-mark-tests.py tests/ --dry-run

# 应用测试标记
apply-marks:
	@echo "Applying test marks..."
	python3 .buildkite/scripts/auto-mark-tests.py tests/
```

```
# 清理环境
clean-env:
	@echo "Cleaning test environment..."
	python3 .buildkite/scripts/cleanup-test-env.py

# 监控CI
monitor-ci:
	@echo "Monitoring CI status..."
	python3 .buildkite/scripts/ci-monitor.py

# 生成报告
generate-report:
	@echo "Generating test report..."
	python3 .buildkite/scripts/generate-report.py

# 性能回归检测
check-performance:
	@echo "Checking for performance regressions..."
```

```
        python3 .buildkite/scripts/performance-regression.py



# 完整的CI流程

ci-full: clean-env install-deps test-fast test-standard generate-report



# 开发者快速检查

dev-check: clean-env test-fast



# 发布前检查

pre-release: clean-env install-deps test-fast test-standard test-exten
```