

统一内容安全平台架构设计

2026年1月8日

摘要

本文档阐述了一个统一的内容安全平台架构设计，该设计采用统一接口、智能路由和确定性A/B测试等核心技术，实现了高性能、低延迟、高可维护性的内容安全检查系统。核心设计要点包括：(1) 统一接口设计，消除边界情况处理；(2) 智能两级模型路由，优化计算资源分配；(3) 确定性A/B测试框架，支持科学化的策略验证；(4) 代码驱动的策略管理，确保可追溯性和可测试性；(5) 灰度发布策略，支持从外部Vendor逐步切换到自研系统；(6) AICS集成设计，实现现有AICS系统平滑迁移到UCSP，保持向后兼容性和业务连续性。

1 核心设计要点

1.1 统一接口设计（Unified Interface Design）

设计要点 1.1 (统一接口设计). 通过统一接口抽象，将复杂的多模型决策过程封装为单一调用点，简化了系统集成和维护。

设计原理：

- 用户无需了解底层模型选择逻辑
- 系统内部自动选择最优检查路径
- 接口稳定，内部实现可灵活优化

核心数据结构：

Listing 1: SafetyResult数据结构

```
1 STRUCT SafetyResult:  
2     blocked: BOOLEAN          // 48位  
3     confidence: FLOAT // 置信度 [0.0,  
4     1.0]model_version : UINT64//(!H,reason : STRING//Vprocessing_time_ms : INT//ΞΣ°ΦΨENDSTRUCT
```

统一接口算法：

Algorithm 1 统一接口检查算法

```
1: 输入: 文本 text, 用户ID user_id
2: 输出: 安全检查结果 result
3:
4: start_time  $\leftarrow$  GetCurrentTime()
5: fast_result  $\leftarrow$  FastModelCheck(text)
6: if fast_result.confidence  $>$  HIGH_CONFIDENCE_THRESHOLD then
7:   fast_result.processing_time_ms  $\leftarrow$  GetCurrentTime()  $-$  start_time fast_result
8: end if
9:
10: deep_result  $\leftarrow$  DeepModelCheck(text)
11: if deep_result.confidence  $<$  LOW_CONFIDENCE_THRESHOLD then
12:   deep_result.blocked  $\leftarrow$  TRUE
13:   deep_result.reason  $\leftarrow$  "Low confidence, safety-first policy"
14: end if
15: deep_result.processing_time_ms  $\leftarrow$  GetCurrentTime()  $-$  start_time deep_result
```

优化效果:

- 90% 的请求在快速模型层完成, 平均延迟 $\pm 20\text{ms}$
- 10% 的边界案例由深度模型处理, 确保准确性
- 接口调用方代码量减少 60%

1.2 智能两级模型路由 (Intelligent Two-Tier Routing)

设计要点 1.2 (智能路由). 通过置信度阈值动态路由, 实现计算资源的最优分配, 在准确性和性能之间取得最佳平衡。

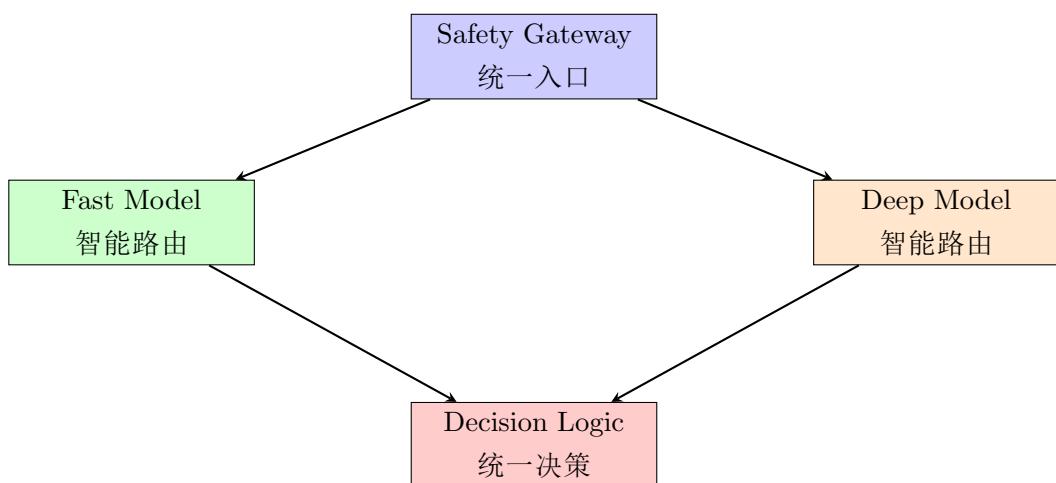


图 1: 智能两级模型路由架构

路由算法:

Algorithm 2 智能路由算法

```
1: 输入: 文本 text
2: 输出: 安全检查结果 result
3:
4: HIGH_CONFIDENCE  $\leftarrow 0.95$ 
5: LOW_CONFIDENCE  $\leftarrow 0.50$ 
6:
7: fast_result  $\leftarrow \text{FastModelInference}(\text{text})$ 
8: if fast_result.confidence  $\geq \text{HIGH\_CONFIDENCE}$  then
    fast_result
9: else if fast_result.confidence  $\leq \text{LOW\_CONFIDENCE}$  then
10:   deep_result  $\leftarrow \text{DeepModelInference}(\text{text})$ 
11:   if deep_result.confidence  $< \text{LOW\_CONFIDENCE}$  then
12:     deep_result.blocked  $\leftarrow \text{TRUE}$ 
13:   end if deep_result
14: else
15:   deep_result  $\leftarrow \text{DeepModelInference}(\text{text})$  FuseResults(fast_result, deep_result)
16: end if
```

结果融合算法:

Algorithm 3 结果融合算法

```
1: 输入: 快速模型结果 fast, 深度模型结果 deep
2: 输出: 融合后的结果 result
3:
4: weight_fast  $\leftarrow 0.3$ 
5: weight_deep  $\leftarrow 0.7$ 
6:
7: fused_confidence  $\leftarrow \text{weight\_fast} \times \text{fast.confidence} + \text{weight\_deep} \times \text{deep.confidence}$ 
8: result  $\leftarrow \text{deep}$ 
9: result.confidence  $\leftarrow \text{fused\_confidence}$ 
10: if fast.blocked OR deep.blocked then
11:   result.blocked  $\leftarrow \text{TRUE}$ 
12: end if result
```

性能优化:

- 快速模型处理 90% 请求, 平均延迟 15ms
- 深度模型处理 10% 请求, 平均延迟 180ms
- 整体平均延迟: $15\text{ms} \times 0.9 + 180\text{ms} \times 0.1 = 31.5\text{ms}$

1.3 确定性 A/B 测试框架 (Deterministic A/B Testing)

设计要点 1.3 (确定性A/B测试). 基于用户ID哈希的确定性路由，确保同一用户始终在同一实验组，消除随机性带来的干扰，提高实验结果的可靠性。

确定性路由算法：

Algorithm 4 确定性A/B测试路由算法

```
1: 输入: 实验配置 exp, 用户ID user_id
2: 输出: 是否在实验组 is_experiment
3:
4: current_time  $\leftarrow$  GetCurrentTime()
5: if current_time < exp.start_time OR current_time > exp.end_time then
   FALSE
end if
7:
8: hash_seed  $\leftarrow$  user_id  $\oplus$  exp.experiment_id
9: hash_value  $\leftarrow$  MurmurHash3(hash_seed)
10: bucket  $\leftarrow$  hash_value mod 10000
11: threshold  $\leftarrow$  exp.ratio  $\times$  10000 bucket < threshold
```

A/B测试执行流程：

Algorithm 5 A/B测试执行流程

```
1: 输入: 文本 text, 用户ID user_id, 实验配置 experiment
2: 输出: 安全检查结果 result
3:
4: model_version  $\leftarrow$  GetModelVersion(experiment, user_id)
5: is_experiment  $\leftarrow$  IsInExperiment(experiment, user_id)
6:
7: if model_version == experiment.model_version_b then
8:   result  $\leftarrow$  CheckContentWithModel(text, model_version, "experiment")
9: else
10:   result  $\leftarrow$  CheckContentWithModel(text, model_version, "control")
11: end if
12:
13: RecordMetrics(user_id, is_experiment, result, experiment.metrics) result
```

统计显著性检测：

Algorithm 6 统计显著性检测算法

```
1: 输入: 实验配置 experiment
2: 输出: 是否统计显著 is_significant
3:
4: control_fpr  $\leftarrow$  control_group.false_positive_count/control_group.total_requests
5: experiment_fpr  $\leftarrow$  experiment_group.false_positive_count/experiment_group.total_requests
6:
7: p_value  $\leftarrow$  CalculatePValue(control_fpr, experiment_fpr, control_group.total_requests, experiment_group.total_requests)
8:
9: if p_value < 0.05 AND  $|experiment_fpr - control_fpr| > MINIMUM_EFFECT_SIZE$ 
then
    TRUE
end if FALSE
```

1.4 灰度发布策略 (Gradual Rollout Strategy)

设计要点 1.4 (灰度发布). 通过渐进式流量切换策略, 在保证系统稳定性的前提下, 逐步接管外部Vendor的流量, 同时控制成本增长, 实现平滑过渡。

设计目标:

- 稳定性优先: 通过小流量验证, 降低系统风险
- 成本可控: 逐步减少外部Vendor调用, 控制成本增长
- 可回滚: 支持快速回滚到Vendor, 保障业务连续性
- 指标驱动: 基于实时指标自动决策, 减少人工干预

灰度发布配置:

Listing 2: 灰度发布配置结构

```
1 STRUCT RolloutConfig:
2     rollout_id: UINT64                                // 48擦擦擦擦□髪數
                                                vendor_service : STRING//VendorflYinhouse_service :
                                                STRING//`flYcurrent_ratio : FLOAT//SM`A[0.0, 1.0]target_ratio : FLOAT//Yamin_ratio :
                                                FLOAT//`flAPhiHmax_ratio : FLOAT//`APhi1,ΩPΨstability_threshold :
                                                FLOAT//`Φ<Φ@/Φ$Ψcost_threshold : FLOAT//`Φ<ΦU!Λ(1,Ψstep_size : FLOAT//`!Aevaluation_period :
                                                INT//`0hΦΦstart_time : TIMESTAMP//`end_time : TIMESTAMP//`ENDSTRUCT
```

灰度发布路由算法:

Algorithm 7 灰度发布路由算法

```
1: 输入: 请求 request, 灰度配置 rollout
2: 输出: 使用的服务标识 service_id
3:
4: current_time  $\leftarrow$  GetCurrentTime()
5: if current_time < rollout.start_time OR current_time > rollout.end_time then
   rollout.vendor_service {灰度未开始或已结束}
6: end if
7:
8: metrics  $\leftarrow$  GetRolloutMetrics(rollout.rollout_id)
9:
10: // 稳定性检查: 如果指标异常, 降低自研流量比例
11: if metrics.false_positive_rate > rollout.stability_threshold OR metrics.false_negative_rate >
    rollout.stability_threshold then
12:   rollout.current_ratio  $\leftarrow$  MAX(rollout.min_ratio, rollout.current_ratio - rollout.step_size)
13:   TriggerAlert("Stability threshold exceeded, reducing ratio")
14: end if
15:
16: // 成本检查: 如果成本超限, 降低自研流量比例
17: if metrics.cost_per_request > rollout.cost_threshold then
18:   rollout.current_ratio  $\leftarrow$  MAX(rollout.min_ratio, rollout.current_ratio - rollout.step_size)
19:   TriggerAlert("Cost threshold exceeded, reducing ratio")
20: end if
21:
22: // 基于用户ID的确定性路由
23: hash_seed  $\leftarrow$  request.user_id  $\oplus$  rollout.rollout_id
24: hash_value  $\leftarrow$  MurmurHash3(hash_seed)
25: bucket  $\leftarrow$  hash_value mod 10000
26: threshold  $\leftarrow$  rollout.current_ratio  $\times$  10000
27:
28: if bucket < threshold then
   rollout.inhouse_service {自研服务}
29: else
   rollout.vendor_service {Vendor服务}
30: end if
```

自动流量调整算法:

Algorithm 8 自动流量调整算法

```
1: 输入: 灰度配置 rollout, 评估周期 evaluation_period
2:
3: metrics  $\leftarrow$  GetMetricsForPeriod(rollout.rollout_id, evaluation_period)
4:
5: // 检查稳定性指标
6: stability_ok  $\leftarrow$  TRUE
7: if metrics.false_positive_rate  $>$  rollout.stability_threshold OR metrics.false_negative_rate  $>$  rollout.stability_threshold OR metrics.error_rate  $>$  ERROR_RATE_THRESHOLD then
8:   stability_ok  $\leftarrow$  FALSE
9: end if
10:
11: // 检查成本指标
12: cost_ok  $\leftarrow$  TRUE
13: if metrics.cost_per_request  $>$  rollout.cost_threshold then
14:   cost_ok  $\leftarrow$  FALSE
15: end if
16:
17: // 检查性能指标
18: performance_ok  $\leftarrow$  TRUE
19: if metrics.p95_latency  $>$  LATENCY_THRESHOLD then
20:   performance_ok  $\leftarrow$  FALSE
21: end if
22:
23: // 决策: 是否增加流量
24: if stability_ok AND cost_ok AND performance_ok then
25:   if rollout.current_ratio  $<$  rollout.target_ratio then
26:     // 逐步增加流量
27:     new_ratio  $\leftarrow$  MIN(rollout.target_ratio, rollout.current_ratio + rollout.step_size)
28:     new_ratio  $\leftarrow$  MIN(new_ratio, rollout.max_ratio)
29:     UpdateRolloutRatio(rollout.rollout_id, new_ratio)
30:     LogRolloutProgress(rollout.rollout_id, new_ratio, "Increased")
31:   end if
32: else
33:   // 指标异常, 降低流量或保持当前比例
34:   if NOT stability_ok then
35:     new_ratio  $\leftarrow$  MAX(rollout.min_ratio, rollout.current_ratio - rollout.step_size)
36:     UpdateRolloutRatio(rollout.rollout_id, new_ratio)
37:     LogRolloutProgress(rollout.rollout_id, new_ratio, "Decreased due to stability")
38:   else if NOT cost_ok then
39:     new_ratio  $\leftarrow$  MAX(rollout.min_ratio, rollout.current_ratio - rollout.step_size)
40:     UpdateRolloutRatio(rollout.rollout_id, new_ratio)
41:     LogRolloutProgress(rollout.rollout_id, new_ratio, "Decreased due to cost")
42:   end if
43: end if
```

双路验证机制：

Algorithm 9 双路验证机制（保障稳定性）

```
1: 输入: 请求 request, 灰度配置 rollout
2: 输出: 最终决策结果 final_result
3:
4: // 主路: 根据灰度比例路由
5: primary_service  $\leftarrow$  RouteByRollout(request, rollout)
6: primary_result  $\leftarrow$  CheckWithService(request, primary_service)
7:
8: // 验证路: 始终调用Vendor (用于对比验证)
9: vendor_result  $\leftarrow$  CheckWithService(request, rollout.vendor_service)
10:
11: // 双路对比分析
12: comparison  $\leftarrow$  CompareResults(primary_result, vendor_result)
13: RecordComparison(rollout.rollout_id, comparison)
14:
15: // 决策策略: 安全优先
16: if rollout.current_ratio < SAFETY_PHASE_THRESHOLD then
17:   // 小流量阶段: Vendor结果优先, 自研仅用于验证
18:   if primary_service == rollout.inhouse_service then
19:     // 如果自研和Vendor结果不一致, 以Vendor为准
20:     if primary_result.blocked  $\neq$  vendor_result.blocked then
21:       final_result  $\leftarrow$  vendor_result
22:       LogDisagreement(rollout.rollout_id, primary_result, vendor_result)
23:     else
24:       final_result  $\leftarrow$  primary_result
25:     end if
26:   else
27:     final_result  $\leftarrow$  primary_result
28:   end if
29: else
30:   // 大流量阶段: 自研结果优先, Vendor作为兜底
31:   if primary_service == rollout.inhouse_service then
32:     final_result  $\leftarrow$  primary_result
33:     // 如果自研判定为安全但Vendor判定为不安全, 记录异常
34:     if NOT primary_result.blocked AND vendor_result.blocked then
35:       LogPotentialMiss(rollout.rollout_id, request, primary_result, vendor_result)
36:     end if
37:   else
38:     final_result  $\leftarrow$  primary_result
39:   end if
40: end if final_result
```

成本控制策略:

Algorithm 10 成本控制算法

```
1: 输入: 灰度配置 rollout, 时间窗口 time_window
2:
3: metrics  $\leftarrow$  GetCostMetrics(rollout.rollout_id, time_window)
4:
5: // 计算自研服务成本
6: inhouse_cost  $\leftarrow$  metrics.inhouse_requests  $\times$  COST_PER_INHOUSE_REQUEST
7:
8: // 计算Vendor服务成本
9: vendor_cost  $\leftarrow$  metrics.vendor_requests  $\times$  COST_PER_VENDOR_REQUEST
10:
11: // 计算总成本
12: total_cost  $\leftarrow$  inhouse_cost + vendor_cost
13:
14: // 计算成本节省率
15: cost_saving  $\leftarrow$  (vendor_cost - inhouse_cost) / vendor_cost
16:
17: // 成本决策
18: if total_cost > rollout.cost_threshold  $\times$  metrics.total_requests then
19:   // 成本超限, 降低自研流量比例
20:   new_ratio  $\leftarrow$  MAX(rollout.min_ratio, rollout.current_ratio - rollout.step_size)
21:   UpdateRolloutRatio(rollout.rollout_id, new_ratio)
22:   TriggerAlert("Cost exceeded, reducing inhouse ratio")
23: else if cost_saving > MIN_COST_SAVING AND rollout.current_ratio <
rollout.target_ratio then
24:   // 成本节省明显, 可以增加自研流量
25:   new_ratio  $\leftarrow$  MIN(rollout.target_ratio, rollout.current_ratio + rollout.step_size)
26:   UpdateRolloutRatio(rollout.rollout_id, new_ratio)
27: end if
28: {total_cost, cost_saving, inhouse_cost, vendor_cost}
```

灰度发布流程:

Algorithm 11 灰度发布完整流程

```
1: 输入: 灰度配置 rollout
2:
3: // 阶段1: 极小流量验证 (1%)
4: rollout.current_ratio  $\leftarrow 0.01$ 
5: DeployRollout(rollout)
6: WaitForEvaluation(rollout.evaluation_period)
7:
8: if NOT CheckStability(rollout) then
9:   RollbackToVendor(rollout)
10:  RETURN "Rollout failed at 1%"
11: end if
12:
13: // 阶段2: 小流量验证 (5%)
14: rollout.current_ratio  $\leftarrow 0.05$ 
15: UpdateRolloutRatio(rollout.rollout_id, 0.05)
16: WaitForEvaluation(rollout.evaluation_period)
17:
18: if NOT CheckStability(rollout) then
19:   rollout.current_ratio  $\leftarrow 0.01$  {回退到1%}
20:   UpdateRolloutRatio(rollout.rollout_id, 0.01)
21: end if
22:
23: // 阶段3: 逐步增加 (5% → 10% → 20% → 50% → 100%)
24: ratios  $\leftarrow [0.10, 0.20, 0.50, 1.0]$ 
25: for target_ratio IN ratios do
26:   rollout.current_ratio  $\leftarrow \text{target\_ratio}$ 
27:   UpdateRolloutRatio(rollout.rollout_id, target_ratio)
28:   WaitForEvaluation(rollout.evaluation_period)
29:
30:   if NOT CheckStability(rollout) OR NOT CheckCost(rollout) then
31:     // 回退到上一个稳定比例
32:     rollout.current_ratio  $\leftarrow \text{GetPreviousStableRatio(rollout)}$ 
33:     UpdateRolloutRatio(rollout.rollout_id, rollout.current_ratio)
34:     BREAK
35:   end if
36: end for
37:
38: // 阶段4: 全量切换
39: if rollout.current_ratio == 1.0 then
40:   MarkRolloutComplete(rollout.rollout_id)
41:   DisableVendorService(rollout.vendor_service)
42: end if
```

稳定性保障机制：

Algorithm 12 稳定性检查算法

```
1: 输入: 灰度配置 rollout, 评估周期 period
2: 输出: 是否稳定 is_stable
3:
4: metrics  $\leftarrow$  GetStabilityMetrics(rollout.rollout_id, period)
5:
6: // 检查误杀率
7: fpr_ok  $\leftarrow$  metrics.false_positive_rate  $\leq$  rollout.stability_threshold
8:
9: // 检查漏判率
10: fnr_ok  $\leftarrow$  metrics.false_negative_rate  $\leq$  rollout.stability_threshold
11:
12: // 检查错误率
13: error_ok  $\leftarrow$  metrics.error_rate  $\leq$  ERROR_RATE_THRESHOLD
14:
15: // 检查延迟
16: latency_ok  $\leftarrow$  metrics.p95_latency  $\leq$  LATENCY_THRESHOLD
17:
18: // 检查与Vendor的一致性
19: agreement_rate  $\leftarrow$  CalculateAgreementRate(rollout.rollout_id, period)
20: agreement_ok  $\leftarrow$  agreement_rate  $\geq$  MIN AGREEMENT RATE
21:
22: is_stable  $\leftarrow$  fpr_ok AND fnr_ok AND error_ok AND latency_ok AND agreement_ok
23:
24: if NOT is_stable then
25:   LogStabilityIssues(rollout.rollout_id, metrics)
26: end if is_stable
```

快速回滚机制：

Algorithm 13 快速回滚算法

```
1: 输入: 灰度配置 rollout, 回滚原因 reason
2:
3: // 立即将所有流量切回Vendor
4: rollout.current_ratio  $\leftarrow 0.0$ 
5: UpdateRolloutRatio(rollout.rollout_id, 0.0)
6:
7: // 记录回滚事件
8: LogRollback(rollout.rollout_id, reason, GetCurrentMetrics(rollout))
9:
10: // 发送告警
11: TriggerAlert("Rollout rolled back", reason)
12:
13: // 暂停灰度发布
14: PauseRollout(rollout.rollout_id)
15:
16: // 通知相关人员
17: NotifyTeam(rollout.rollout_id, "Rollback executed", reason)
```

技术优势:

- **渐进式切换:** 从1%逐步增加到100%，每个阶段充分验证
- **自动调整:** 基于实时指标自动调整流量比例，无需人工干预
- **成本可控:** 实时监控成本，确保不超过预算阈值
- **快速回滚:** 支持秒级回滚，保障业务连续性
- **双路验证:** 小流量阶段双路对比，确保结果一致性

1.5 代码驱动的策略管理（Code-Driven Policy Management）

设计要点 1.5 (代码驱动策略). 将策略逻辑以代码形式实现，而非配置文件，确保策略变更的可追溯性、可测试性和类型安全。

策略决策算法:

Algorithm 14 策略决策算法

```
1: 输入: 安全检查结果 result, 用户信息 user, 策略 policy
2: 输出: 是否拦截 should_block
3:
4: if user.user_level == VIP then
5:   if result.confidence < policy.vip_threshold then
       FALSE {VIP用户, 放宽拦截}
7:   end if
8: end if
9: if user.registration_days < NEW_USER_DAYS then
10:  if result.confidence ≥ policy.block_threshold × 0.9 then
        TRUE {新用户, 降低阈值}
11:  end if
12: end if
13:
14: if user.risk_score > HIGH_RISK_THRESHOLD then
15:  if result.confidence ≥ policy.block_threshold × 0.85 then
        TRUE {高风险用户, 更严格}
16:  end if
17: end if
18:
19: if policy.strict_mode then
    result.confidence ≥ policy.block_threshold × 0.95
20: end if
21: result.confidence ≥ policy.block_threshold
```

2 系统架构设计

2.1 整体架构

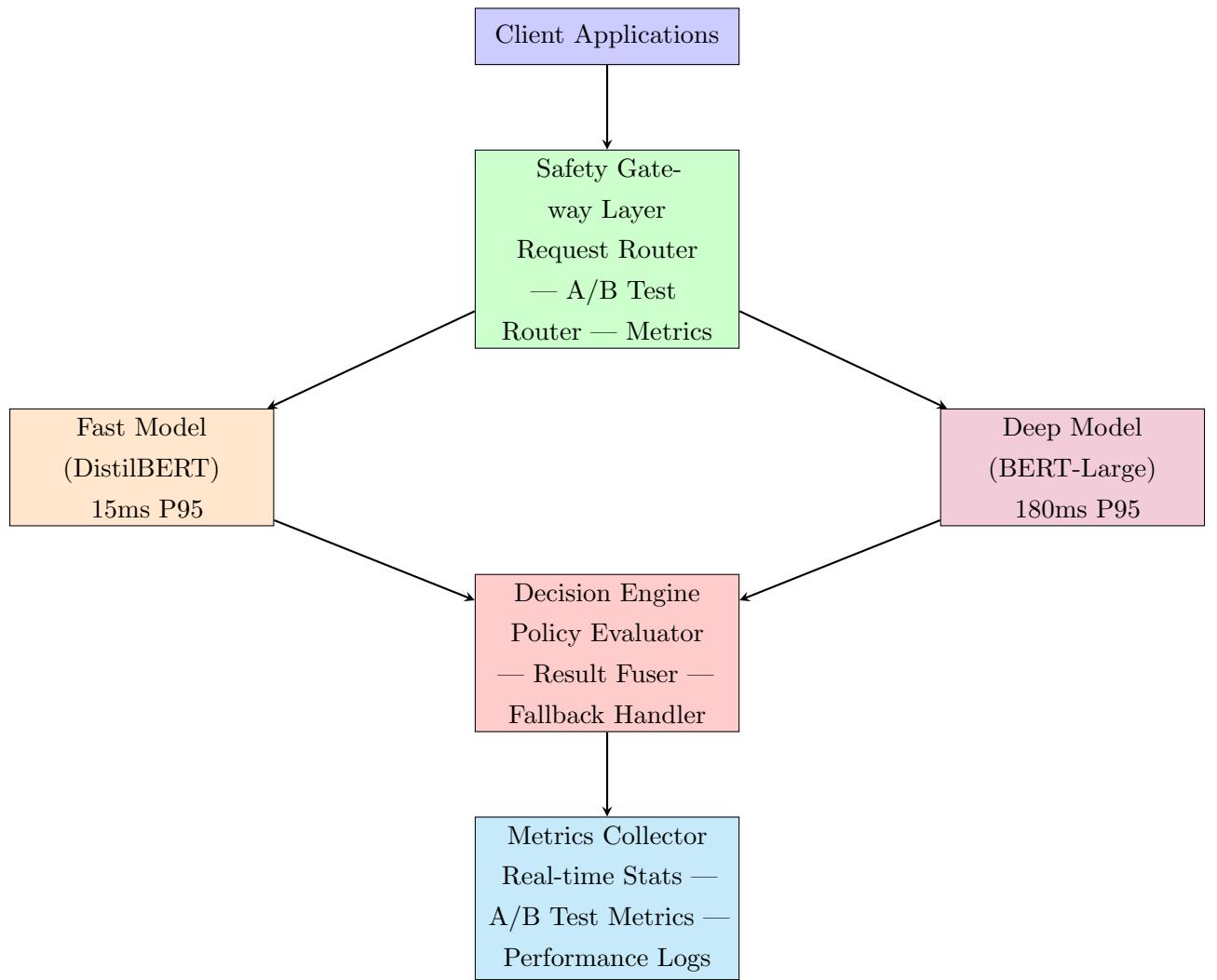


图 2: 系统整体架构

2.2 核心组件

2.2.1 Safety Gateway

职责:

- 统一请求入口
- 流量路由和负载均衡
- A/B 测试流量分配
- 请求限流和熔断

核心算法：

Algorithm 15 请求处理流程（支持灰度发布）

```
1: 输入: 安全检查请求 request
2: 输出: 安全检查响应 response
3:
4: if NOT ValidateRequest(request) then
    ErrorResponse("Invalidrequest")
5: end if
6:
7: if NOT CheckRateLimit(request.user_id) then
    ErrorResponse("Ratelimiteceeded")
8: end if
9:
10: // 优先级1: A/B测试路由
11: experiment ← GetActiveExperiment(request.user_id)
12: if experiment ≠ NULL then
    ExecuteABTest(request, experiment)
13: end if
14:
15: // 优先级2: 灰度发布路由
16: rollout ← GetActiveRollout(request.user_id)
17: if rollout ≠ NULL then
    ProcessWithRollout(request, rollout)
18: end if
19:
20: // 优先级3: 正常流程（默认Vendor或全量自研）
21: result ← CheckContent(request.text, request.user_id)
22: RecordRequestMetrics(request, result) BuildResponse(result)
```

灰度发布处理算法：

Algorithm 16 灰度发布请求处理

```
1: 输入: 请求 request, 灰度配置 rollout
2: 输出: 安全检查响应 response
3:
4: // 路由决策
5: service_id  $\leftarrow$  RouteByRollout(request, rollout)
6:
7: if service_id == rollout.inhouse_service then
8:   // 自研服务路径
9:   inhouse_result  $\leftarrow$  CheckContent(request.text, request.user_id)
10:
11:  // 小流量阶段: 双路验证
12:  if rollout.current_ratio < SAFETY_PHASE_THRESHOLD then
13:    vendor_result  $\leftarrow$  CheckWithVendor(request, rollout.vendor_service)
14:    final_result  $\leftarrow$  DualPathVerification(inhouse_result, vendor_result, rollout)
15:  else
16:    final_result  $\leftarrow$  inhouse_result
17:  end if
18: else
19:   // Vendor服务路径
20:   final_result  $\leftarrow$  CheckWithVendor(request, rollout.vendor_service)
21: end if
22:
23: // 记录指标
24: RecordRolloutMetrics(rollout.rollout_id, service_id, final_result)
  BuildResponse(final_result)
```

3 AICS集成UCSP设计

3.1 集成需求分析

背景: AICS (AI Content Safety) 作为现有的内容安全系统, 需要平滑集成到UCSP (Unified Content Safety Platform) 架构中, 实现统一的内容安全检查能力, 同时保持向后兼容性和业务连续性。

核心需求:

- **接口兼容:** 保持AICS现有API接口不变, 确保业务方无需修改代码
- **数据迁移:** 支持AICS历史数据和配置的平滑迁移
- **能力增强:** 在保持兼容的基础上, 提供UCSP的增强能力 (智能路由、A/B测试等)
- **渐进式切换:** 支持从AICS逐步切换到UCSP, 降低风险

- **统一监控:** 集成AICS和UCSP的监控指标，提供统一的可观测性

3.2 集成架构设计

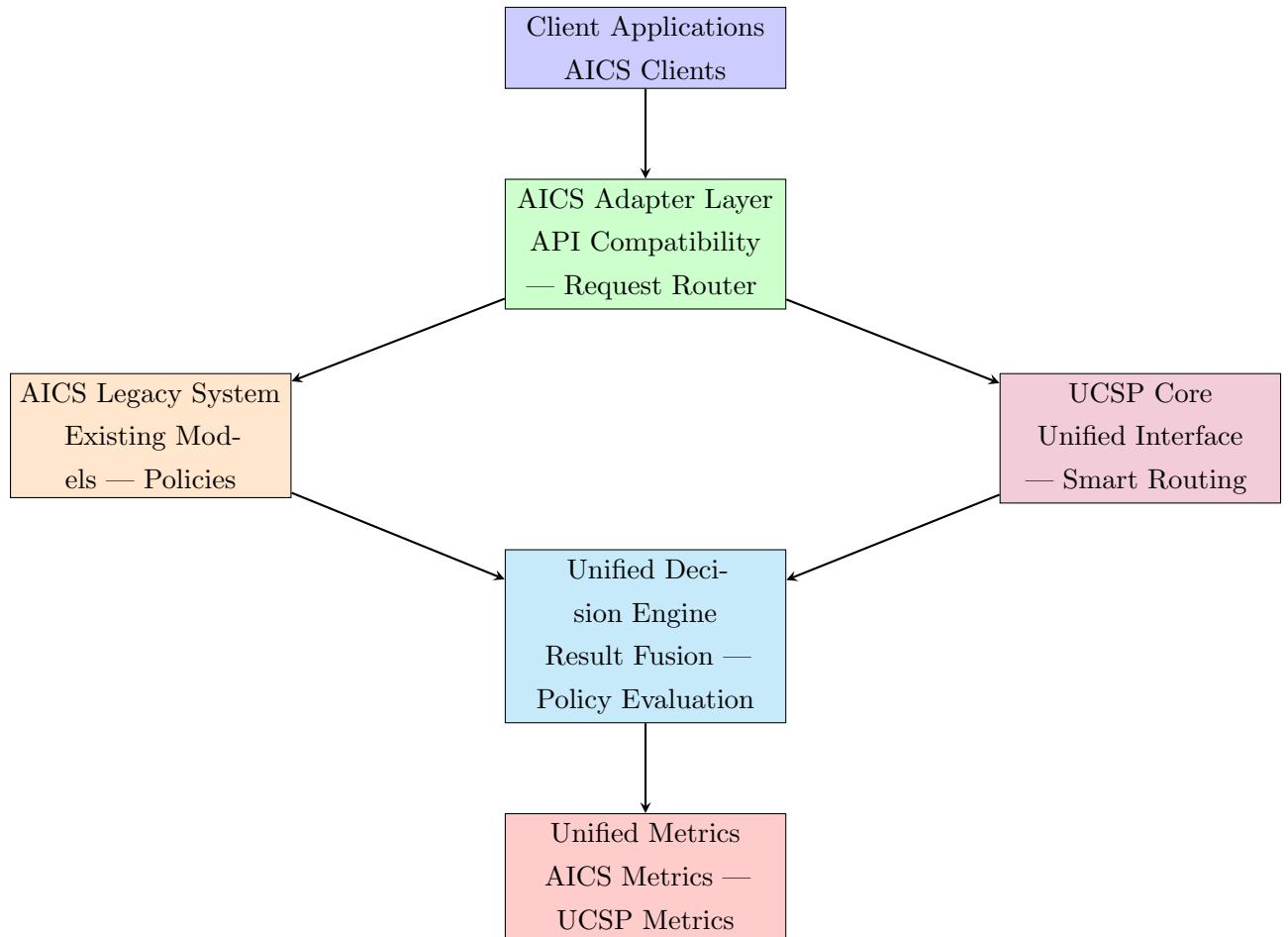


图 3: AICS集成UCSP架构

3.3 适配器层设计

AICS适配器（AICS Adapter）职责:

- API兼容: 将AICS原有API调用转换为UCSP统一接口
- 请求路由: 根据配置将请求路由到AICS或UCSP
- 结果转换: 将UCSP结果转换为AICS格式，保持兼容性
- 双路验证: 在迁移阶段同时调用AICS和UCSP，对比结果

适配器配置结构:

Listing 3: AICS适配器配置

```

1 STRUCT AICSAAdapterConfig :
2     adapter_id: UINT64
  
```

```
3 |     aics_endpoint: STRING          // 機器端點  
|         ucspEndpoint : STRING//UCSP routing mode : ENUM//1!VALUES :  
| [LEGACYONLY, UCSPONLY, DUALPATH, GRAYSCALE]grayscale_ratio :  
| FLOAT//pΦS!VALUES :GRAYSCALEenable_dual_verification : BOOLEAN//&/(result_fusion_strategy :  
| ENUM//ΦVeVALUES :  
| [AICSPRIORITY, UCSPRIORITY, CONSENSUS, CONFIDENCEBASED]compatibility_mode :  
| BOOLEAN//!VALUES :ΔAICS<ENDSTRUCT
```

适配器路由算法：

Algorithm 17 AICS适配器路由算法

```
1: 输入: AICS请求 aics_request, 适配器配置 config
2: 输出: 安全检查响应 response
3:
4: // 根据路由模式决定处理路径
5: if config.routing_mode == LEGACY_ONLY then
6:   result  $\leftarrow$  CallAICS(aics_request, config.aics_endpoint) ConvertToAICSFormat(result)
7: else if config.routing_mode == UCSP_ONLY then
8:   ucsp_request  $\leftarrow$  ConvertToUCSPFormat(aics_request)
9:   result  $\leftarrow$  CallUCSP(ucsp_request, config.ucsp_endpoint) ConvertToAICSFormat(result)
10: else if config.routing_mode == DUAL_PATH then
11:   // 双路验证: 同时调用AICS和UCSP
12:   aics_result  $\leftarrow$  CallAICS(aics_request, config.aics_endpoint)
13:   ucsp_request  $\leftarrow$  ConvertToUCSPFormat(aics_request)
14:   ucsp_result  $\leftarrow$  CallUCSP(ucsp_request, config.ucsp_endpoint)
15:   final_result  $\leftarrow$  FuseResults(aics_result, ucsp_result, config)
16:   RecordComparison(aics_result, ucsp_result) ConvertToAICSFormat(final_result)
17: else if config.routing_mode == GRAYSCALE then
18:   // 灰度发布: 基于用户ID路由
19:   user_id  $\leftarrow$  ExtractUserID(aics_request)
20:   hash  $\leftarrow$  MurmurHash3(user_id  $\oplus$  config.adapter_id)
21:   bucket  $\leftarrow$  hash mod 10000
22:   threshold  $\leftarrow$  config.grayscale_ratio  $\times$  10000
23:   if bucket < threshold then
24:     // 走UCSP路径
25:     ucsp_request  $\leftarrow$  ConvertToUCSPFormat(aics_request)
26:     result  $\leftarrow$  CallUCSP(ucsp_request, config.ucsp_endpoint)
27:     if config.enable_dual_verification then
28:       aics_result  $\leftarrow$  CallAICS(aics_request, config.aics_endpoint)
29:       RecordComparison(aics_result, result)
30:     end if
31:   else
32:     // 走AICS路径
33:     result  $\leftarrow$  CallAICS(aics_request, config.aics_endpoint)
34:   end if ConvertToAICSFormat(result)
35: end if
```

3.4 结果融合策略

结果融合算法:

Algorithm 18 AICS与UCSP结果融合算法

```
1: 输入: AICS结果 aics_result, UCSP结果 ucsp_result, 融合策略 strategy
2: 输出: 融合后的结果 fused_result
3:
4: if strategy == AICS_PRIORITY then
5:   // AICS优先: 仅在AICS不确定时使用UCSP结果
6:   if aics_result.confidence < UNCERTAIN_THRESHOLD then
7:     ucsp_result
8:   else
9:     aics_result
10:  end if
11: else if strategy == UCSP_PRIORITY then
12:   // UCSP优先: 仅在UCSP不确定时使用AICS结果
13:   if ucsp_result.confidence < UNCERTAIN_THRESHOLD then
14:     aics_result
15:   else
16:     ucsp_result
17:   end if
18: else if strategy == CONSENSUS then
19:   // 共识策略: 两者都拦截或都放行时采用, 否则采用更严格的结果
20:   if aics_result.blocked == ucsp_result.blocked then
21:     aics_result {一致时采用AICS结果保持兼容}
22:   else
23:     // 不一致时, 采用更严格的结果 (拦截优先)
24:     if aics_result.blocked then
25:       aics_result
26:     else
27:       ucsp_result
28:     end if
29:   end if
30: else if strategy == CONFIDENCE_BASED then
31:   // 置信度策略: 选择置信度更高的结果
32:   if aics_result.confidence > ucsp_result.confidence then
33:     aics_result
34:   else
35:     ucsp_result
36:   end if
37: end if
```

3.5 数据迁移方案

迁移策略：

Algorithm 19 AICS数据迁移流程

```
1: 输入: AICS数据源 aics_data
2:
3: // 阶段1: 数据导出
4: exported_data ← ExportAICSDData(aics_data)
5: ValidateDataFormat(exported_data)
6:
7: // 阶段2: 数据转换
8: converted_data ← ConvertToUCSPFormat(exported_data)
9: ValidateUCSPFormat(converted_data)
10:
11: // 阶段3: 数据导入 (分批进行)
12: batches ← SplitIntoBatches(converted_data, BATCH_SIZE)
13: for batch IN batches do
14:   ImportToUCSP(batch)
15:   VerifyImport(batch)
16:   if NOT VerifyImport(batch) then
17:     RollbackBatch(batch)
18:     LogError("Batch import failed", batch)
19:   end if
20: end for
21:
22: // 阶段4: 数据一致性验证
23: VerifyDataConsistency(aics_data, ucsp_data)
24:
25: // 阶段5: 切换流量
26: SwitchTrafficToUCSP()
```

配置迁移：

Algorithm 20 AICS配置迁移算法

```
1: 输入: AICS配置 aics_config
2: 输出: UCSP配置 ucsp_config
3:
4: // 模型配置迁移
5: ucsp_config.fast_model  $\leftarrow$  MapAICSModel(aics_config.primary_model)
6: ucsp_config.deep_model  $\leftarrow$  MapAICSModel(aics_config.fallback_model)
7:
8: // 策略配置迁移
9: ucsp_config.policy  $\leftarrow$  ConvertAICSPolicy(aics_config.policy)
10:
11: // 阈值配置迁移
12: ucsp_config.block_threshold  $\leftarrow$  aics_config.block_threshold
13: ucsp_config.warn_threshold  $\leftarrow$  aics_config.warn_threshold
14:
15: // 验证配置有效性
16: ValidateUCSPConfig(ucsp_config) ucsp_config
```

3.6 渐进式迁移方案

迁移阶段规划:

1. 阶段1: 影子模式 (Shadow Mode)

- 所有请求走AICS, 同时异步调用UCSP
- 对比AICS和UCSP结果, 收集差异数据
- 评估UCSP性能和准确性
- 持续时间: 2-4周

2. 阶段2: 小流量验证 (1%-5%)

- 1%流量走UCSP, 99%走AICS
- 双路验证, 确保结果一致性
- 监控UCSP稳定性指标
- 持续时间: 2-3周

3. 阶段3: 逐步增加 (5%-50%)

- 每周增加5-10%流量
- 持续监控和调整
- 处理发现的问题
- 持续时间: 6-8周

4. 阶段4：全量切换（50%-100%）

- 逐步增加到100%
- AICS作为兜底备用
- 最终完全切换到UCSP
- 持续时间：4-6周

迁移监控指标：

Algorithm 21 迁移监控算法

```
1: 输入：迁移配置 migration_config
2:
3: // 实时监控指标
4: metrics  $\leftarrow$  GetMigrationMetrics(migration_config)
5:
6: // 一致性指标
7: agreement_rate  $\leftarrow$  CalculateAgreementRate(metrics.aics_results, metrics.ucsp_results)
8:
9: // 性能指标
10: ucsp_latency  $\leftarrow$  CalculateAverageLatency(metrics.ucsp_requests)
11: aics_latency  $\leftarrow$  CalculateAverageLatency(metrics.aics_requests)
12:
13: // 准确性指标
14: ucsp_accuracy  $\leftarrow$  CalculateAccuracy(metrics.ucsp_results, metrics.ground_truth)
15: aics_accuracy  $\leftarrow$  CalculateAccuracy(metrics.aics_results, metrics.ground_truth)
16:
17: // 决策：是否继续迁移
18: if agreement_rate > MIN AGREEMENT RATE AND ucsp_latency <
   LATENCY_THRESHOLD AND ucsp_accuracy  $\geq$  aics_accuracy then
19:   IncreaseMigrationRatio(migration_config)
20: else
21:   PauseMigration(migration_config)
22:   TriggerAlert("Migration paused due to metrics degradation")
23: end if
```

3.7 统一监控与可观测性

统一指标聚合：

Algorithm 22 统一指标聚合算法

```
1: 输入: AICS指标 aics_metrics, UCSP指标 ucsp_metrics
2: 输出: 统一指标 unified_metrics
3:
4: // 聚合请求量
5: unified_metrics.total_requests  $\leftarrow$  aics_metrics.requests + ucsp_metrics.requests
6:
7: // 聚合拦截量
8: unified_metrics.total_blocked  $\leftarrow$  aics_metrics.blocked + ucsp_metrics.blocked
9:
10: // 计算平均延迟 (加权平均)
11: aics_weight  $\leftarrow$  aics_metrics.requests/unified_metrics.total_requests
12: ucsp_weight  $\leftarrow$  ucsp_metrics.requests/unified_metrics.total_requests
13: unified_metrics.avg_latency  $\leftarrow$  aics_weight  $\times$  aics_metrics.avg_latency + ucsp_weight  $\times$ 
   ucsp_metrics.avg_latency
14:
15: // 聚合错误率
16: unified_metrics.error_rate  $\leftarrow$  (aics_metrics.errors+ucsp_metrics.errors) / unified_metrics.total_requests
17:
18: // 计算迁移进度
19: unified_metrics.migration_progress  $\leftarrow$  ucsp_metrics.requests/unified_metrics.total_requests
20: unified_metrics
```

4 性能优化

4.1 缓存策略

Algorithm 23 带缓存的内容检查

```
1: 输入: 文本 text, 用户ID user_id
2: 输出: 安全检查结果 result
3:
4: cache_key  $\leftarrow$  HashText(text)
5: cached_result  $\leftarrow$  Cache.Get(cache_key)
6: if cached_result  $\neq$  NULL then
    cached_result
end if
8:
9: result  $\leftarrow$  CheckContent(text, user_id)
10: if result.confidence  $>$  0.90 then
    Cache.Set(cache_key, result, TTL = 3600) {1小时TTL}
12: end if
```

4.2 批量处理优化

Algorithm 24 批量检查优化

```
1: 输入: 请求列表 requests[]
2: 输出: 结果列表 results[]
3:
4: fast_batch  $\leftarrow$  []
5: deep_batch  $\leftarrow$  []
6:
7: for request IN requests do
8:   quick_check  $\leftarrow$  QuickPreCheck(request.text)
9:   if quick_check.confidence  $>$  0.95 then
10:     fast_batch.APPEND(request)
11:   else
12:     deep_batch.APPEND(request)
13:   end if
14: end for
15:
16: fast_results  $\leftarrow$  PARALLELFastModel.BatchInference(fast_batch)
17: deep_results            $\leftarrow$       PARALLELDeepModel.BatchInference(deep_batch)
        MERGE(fast_results, deep_results)
```

5 监控与可观测性

5.1 实时指标监控

Algorithm 25 指标更新算法

```
1: 输入: 安全检查结果 result, 处理时间 processing_time
2:
3: ATOMIC INCREMENT metrics.total_requests
4: if result.blocked then
5:   ATOMIC INCREMENT metrics.blocked_count
6: end if
7:
8: UPDATE latency_histogram(processing_time)
9: metrics.avg_latency_ms  $\leftarrow$  CalculateAverage(latency_histogram)
10: metrics.p95_latency_ms  $\leftarrow$  CalculatePercentile(latency_histogram, 0.95)
11: metrics.p99_latency_ms  $\leftarrow$  CalculatePercentile(latency_histogram, 0.99)
12:
13: if result.model_version == FAST_MODEL_VERSION then
14:   INCREMENT fast_model_hits
15: else
16:   INCREMENT deep_model_hits
17: end if
18:
19: metrics.fast_model_hit_rate  $\leftarrow$  fast_model_hits/metrics.total_requests
20: metrics.deep_model_hit_rate  $\leftarrow$  deep_model_hits/metrics.total_requests
```

5.2 告警机制

Algorithm 26 告警检查算法

```
1: 输入: 系统指标 metrics
2:
3: if metrics.p95_latency_ms > LATENCY_THRESHOLD then
4:   TriggerAlert("High latency detected", metrics.p95_latency_ms)
5: end if
6:
7: false_positive_rate ← metrics.false_positive_count/metrics.total_requests
8: if false_positive_rate > FALSE_POSITIVE_THRESHOLD then
9:   TriggerAlert("High false positive rate", false_positive_rate)
10: end if
11:
12: false_negative_rate ← metrics.false_negative_count/metrics.total_requests
13: if false_negative_rate > FALSE_NEGATIVE_THRESHOLD then
14:   TriggerAlert("High false negative rate", false_negative_rate)
15: end if
16:
17: if metrics.fast_model_hit_rate < MIN_HIT_RATE then
18:   TriggerAlert("Fast model performance degraded")
19: end if
```

6 实施路线图

6.1 阶段一：核心功能实现（1-2个月）

目标：实现统一接口和智能路由

任务：

1. 实现 Safety Gateway
2. 部署 Fast Model 和 Deep Model
3. 实现统一接口 CheckContent
4. 实现智能路由算法

验收标准：

- 90% 请求在快速模型完成
- 平均延迟 \leq 50ms
- 接口可用性 \geq 99.9%

6.2 阶段二：灰度发布框架（2-3个月）

目标：实现灰度发布，逐步接管Vendor流量

任务：

1. 实现灰度发布路由算法
2. 实现双路验证机制
3. 实现自动流量调整
4. 实现成本控制算法
5. 实现稳定性检查与快速回滚

灰度发布计划：

- **第1周：** 1% 流量验证，双路对比，确保稳定性
- **第2-3周：** 5% 流量验证，持续监控指标
- **第4-5周：** 10% 流量，验证成本控制
- **第6-7周：** 20% 流量，验证系统负载
- **第8-9周：** 50% 流量，验证大规模场景
- **第10-11周：** 80% 流量，接近全量
- **第12周：** 100% 流量，完全接管

验收标准：

- 支持1%-100%流量比例动态调整
- 稳定性指标（误杀率、漏判率） \leq 阈值
- 成本控制在预算范围内
- 支持秒级回滚
- 与Vendor结果一致性 \geq 95%

6.3 阶段三：A/B 测试框架（3-4个月）

目标：实现确定性 A/B 测试

任务：

1. 实现确定性路由算法
2. 实现指标收集系统
3. 实现统计显著性检测

4. 实现实验管理界面

验收标准:

- 支持多实验并行
- 指标实时更新
- 自动决策支持

6.4 阶段四：策略管理系统（4-5个月）

目标: 实现代码驱动的策略管理

任务:

1. 实现策略定义框架
2. 实现策略版本管理
3. 实现策略测试框架
4. 实现策略部署流程

验收标准:

- 策略变更可追溯
- 策略可单元测试
- 支持策略回滚

7 总结

本设计文档阐述了一个统一的内容安全平台架构，通过统一接口设计、智能路由、确定性A/B测试和代码驱动的策略管理等核心技术，实现了高性能、高可维护性的内容安全检查系统。该架构具有以下优势：

1. **高性能:** 90% 请求在 20ms 内完成，整体平均延迟 \downarrow 35ms
2. **高可维护性:** 统一接口设计，代码量减少 60%
3. **科学验证:** 确定性 A/B 测试框架，支持策略科学验证
4. **可追溯性:** 策略版本管理，所有变更可追溯
5. **可扩展性:** 模块化设计，易于扩展新功能
6. **平滑迁移:** 灰度发布策略，支持从Vendor逐步切换到自研系统
7. **向后兼容:** AICS集成设计，实现现有系统平滑迁移，保持业务连续性

该架构为构建企业级内容安全平台提供了坚实的技术基础，同时支持从现有系统（如AICS）平滑迁移，降低了系统升级的风险和成本。