

# 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。

## 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          // 是否拦截  
3     confidence: FLOAT         // 置信度 [0.0, 1.0]  
4     model_version: UINT64      // 使用的模型版本  
5     reason: STRING            // 决策原因  
6     processing_time_ms: INT    // 处理耗时 (毫秒)  
7 END STRUCT
```

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
8:   return fast_result
9: end if
10:
11: deep_result  $\leftarrow$  DeepModelCheck(text)
12: if deep_result.confidence  $<$  LOW_CONFIDENCE_THRESHOLD then
13:   deep_result.blocked  $\leftarrow$  TRUE
14:   deep_result.reason  $\leftarrow$  "Low confidence, safety-first policy"
15: end if
16: deep_result.processing_time_ms  $\leftarrow$  GetCurrentTime()  $-$  start_time
17: return deep_result
```

---

优化效果:

- 90% 的请求在快速模型层完成, 平均延迟  $< 20\text{ms}$
- 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。

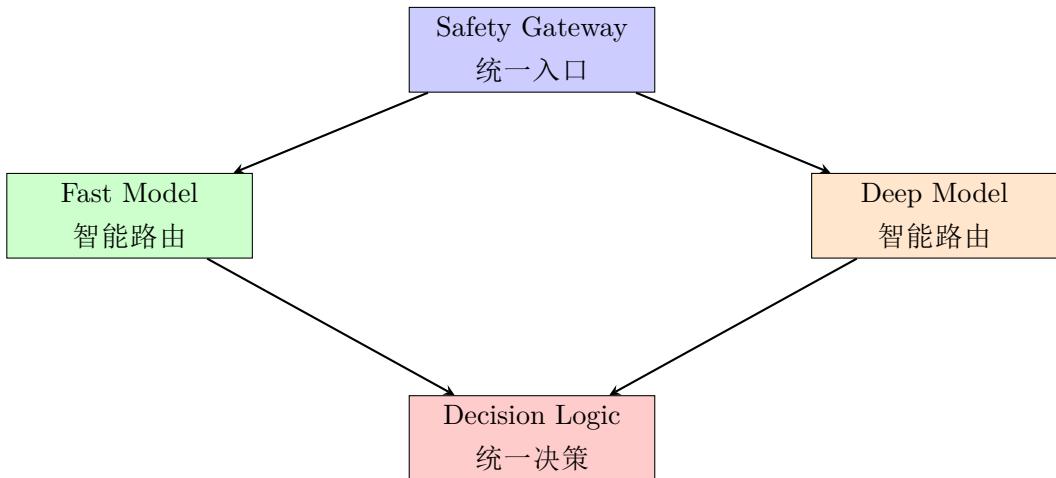


图 1: 智能两级模型路由架构

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
9:   return fast_result
10: else if fast_result.confidence  $\leq \text{LOW\_CONFIDENCE}$  then
11:   deep_result  $\leftarrow \text{DeepModelInference}(\text{text})$ 
12:   if deep_result.confidence  $< \text{LOW\_CONFIDENCE}$  then
13:     deep_result.blocked  $\leftarrow \text{TRUE}$ 
14:   end if
15:   return deep_result
16: else
17:   deep_result  $\leftarrow \text{DeepModelInference}(\text{text})$ 
18:   return FuseResults(fast_result, deep_result)
19: 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
13: return 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  $\leftarrow \text{GetCurrentTime}()$ 
5: if current_time  $< \text{exp.start\_time}$  OR current_time  $> \text{exp.end\_time}$  then
6:   return FALSE
7: end if
8:
9: hash_seed  $\leftarrow \text{user\_id} \oplus \text{exp.experiment\_id}$ 
10: hash_value  $\leftarrow \text{MurmurHash3}(\text{hash\_seed})$ 
11: bucket  $\leftarrow \text{hash\_value} \bmod 10000$ 
12: threshold  $\leftarrow \text{exp.ratio} \times 10000$ 
13: return bucket  $< \text{threshold}$ 
```

---

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)
14: return 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
10:   return TRUE
11: end if
12: return FALSE
```

---

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

**设计要点 1.4** (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 7** 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
6:     return FALSE {VIP 用户, 放宽拦截}
7:   end if
8: end if
9:
10: if user.registration_days < NEW_USER_DAYS then
11:   if result.confidence ≥ policy.block_threshold × 0.9 then
12:     return TRUE {新用户, 降低阈值}
13:   end if
14: end if
15:
16: if user.risk_score > HIGH_RISK_THRESHOLD then
17:   if result.confidence ≥ policy.block_threshold × 0.85 then
18:     return TRUE {高风险用户, 更严格}
19:   end if
20: end if
21:
22: if policy.strict_mode then
23:   return result.confidence ≥ policy.block_threshold × 0.95
24: end if
25:
26: return result.confidence ≥ policy.block_threshold
```

---

## 2 系统架构设计

### 2.1 整体架构

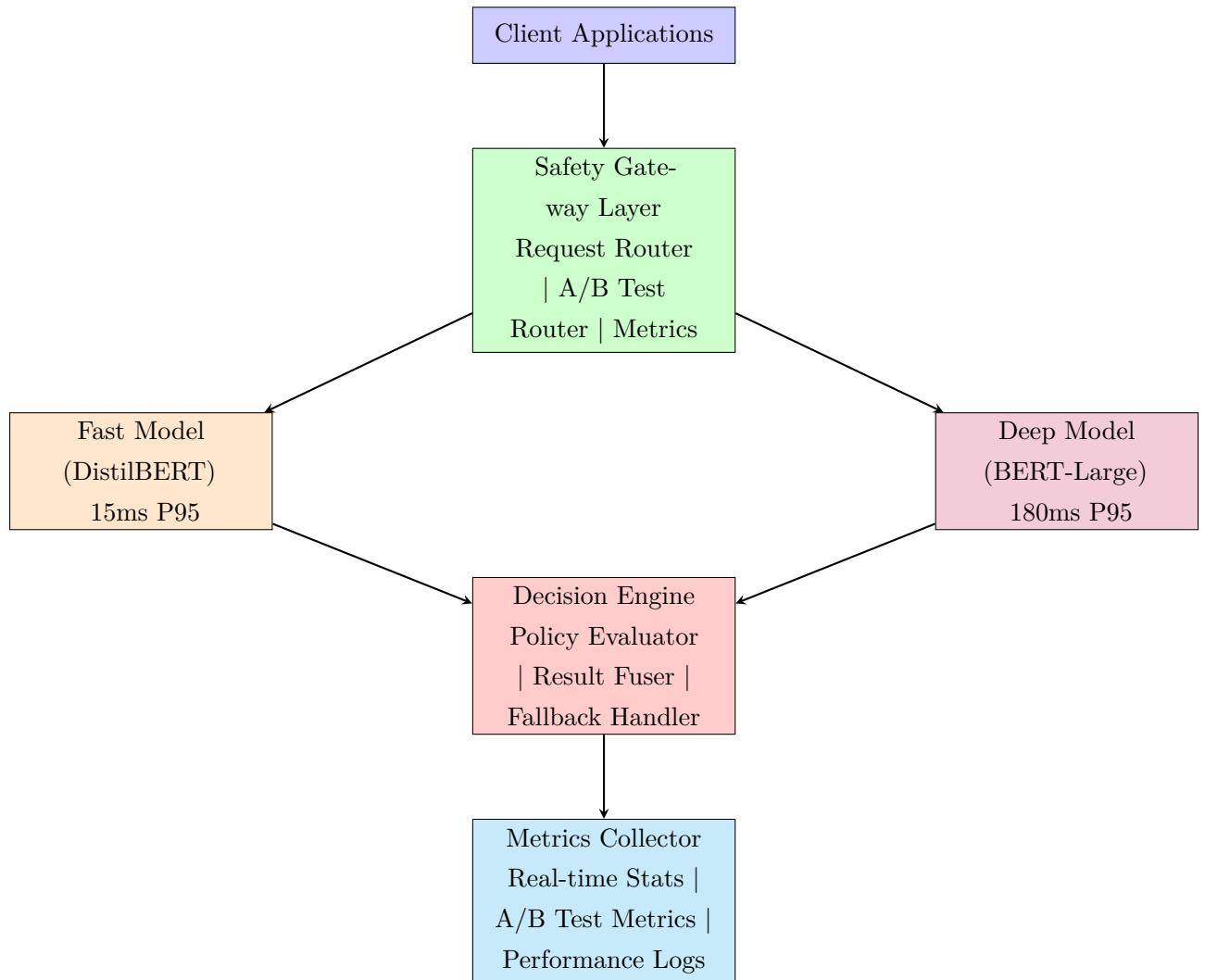


图 2: 系统整体架构

### 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 8** 请求处理流程

---

```
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
5:   return ErrorResponse("Invalidrequest")
6: end if
7:
8: if NOT CheckRateLimit(request.user_id) then
9:   return ErrorResponse("Ratelimiteceeded")
10: end if
11:
12: experiment ← GetActiveExperiment(request.user_id)
13: if experiment ≠ NULL then
14:   return ExecuteABTest(request, experiment)
15: end if
16:
17: result ← CheckContent(request.text, request.user_id)
18: RecordRequestMetrics(request, result)
19: return BuildResponse(result)
```

---

### 3 性能优化

#### 3.1 缓存策略

---

**Algorithm 9** 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
7:   return cached_result
8: end if
9:
10: result  $\leftarrow$  CheckContent(text, user_id)
11: if result.confidence  $>$  0.90 then
12:   Cache.Set(cache_key, result, TTL = 3600) {1 小时 TTL}
13: end if
14: return result
```

---

#### 3.2 批量处理优化

---

**Algorithm 10** 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)
18: return MERGE(fast_results, deep_results)
```

---

## 4 监控与可观测性

### 4.1 实时指标监控

---

#### Algorithm 11 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
```

---

## 4.2 告警机制

---

### Algorithm 12 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
```

---

## 5 实施路线图

### 5.1 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 < 50ms
- 接口可用性 > 99.9%

## 5.2 阶段二：A/B 测试框架（2-3 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

## 5.3 阶段三：策略管理系统（3-4 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

## 6 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 < 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

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。