日志服务

Agent 可观测

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

Agent 可观测
Agent 可观测应用详情

最近更新时间：2026-07-14 11:27:02

说明：

- 此功能需要白名单权限使用，如有需要，可 提交工单 与我们联系。
- 如果您正在使用 AI 工具，可安装 CLS Agent 可观测分析 Skill，让 AI 基于已上报的 Trace 数据自动执行多维度分析（错误诊断、性能瓶颈、Token 成本、用户画像等），直接输出结构化报告。安装后在 AI 对话中描述分析需求即可（如"帮我看看 Agent 最近的运行情况"）。

概述

应用详情页是 Agent 可观测的核心功能页面，为已接入的 AI 应用提供全方位的监控与分析视图。进入 CLS 控制台 > 应用中心 > 您的应用，即可查看应用详情。
页面包含三大功能模块：**仪表盘**、**调用链**和**会话**，帮助您从宏观指标到微观链路全面掌握 AI 应用的运行状况。

场景	能力	核心价值
整体运行状态如何?	仪表盘 – 总览	请求数、错误数、模型调用次数、Token 消耗等核心指标一览，支持按 Agent/模型维度 Top10排行
性能是否有瓶颈?	仪表盘 – 性能	请求耗时分布、P50/P90/P99分位趋势、模型平均耗时排行，定位慢模型和长尾延迟
Token 成本花在哪?	仪表盘 – 成本&Token	输入/输出 Token 总量、分位趋势，掌握成本走向
应用内部组件表现如何?	仪表盘 – 应用观测	按操作类型（generation/tool/agent/retriever 等）分析调用分布和耗时
单次请求发生了什么?	调用链	完整的 Trace 列表和详情，支持按状态/错误类型/耗时等过滤，展示调用树和每个节点的 Input/Output
用户多轮对话全貌?	会话	以 Session 维度聚合 Trace，还原多轮对话上下文，统计会话级 Token 和成本

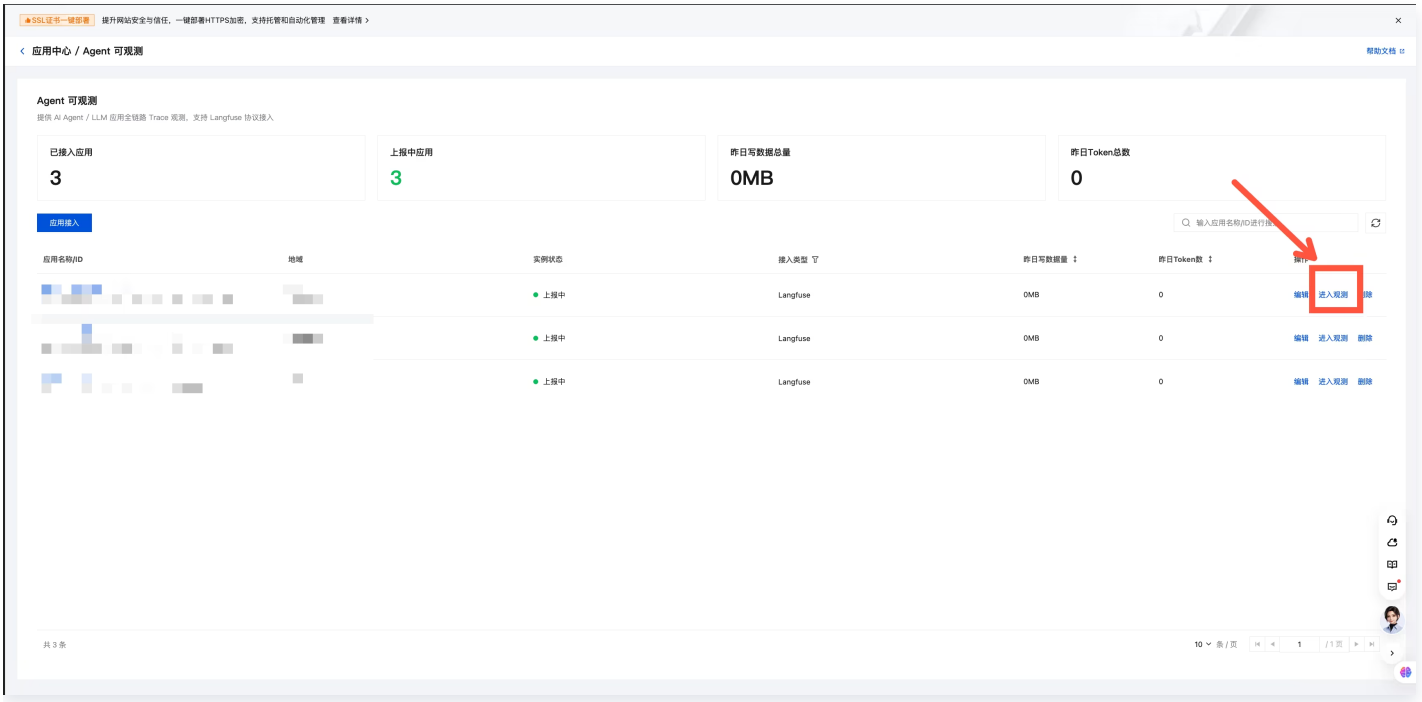
应用列表

进入 CLS 控制台 > 应用中心 > **Agent 可观测**页面，可查看所有已接入的 AI 应用。页面顶部展示以下全局指标：

指标	说明

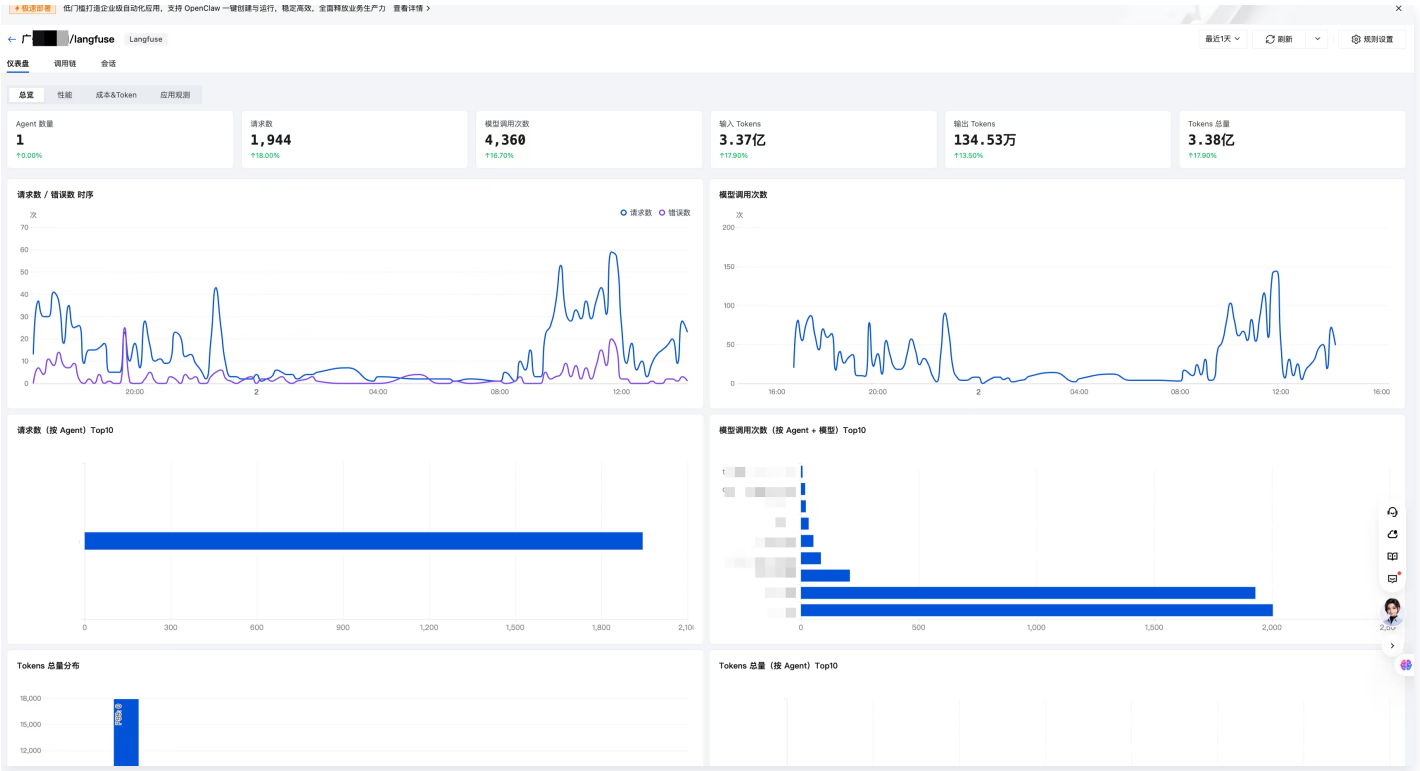
已接入应用	当前已成功接入 Agent 可观测的应用总数。
上报中应用	正在活跃上报数据的应用数量。
昨日写数据总量	前一日所有应用写入 CLS 的数据总量。
昨日 Token 总数	前一日所有应用消耗的 Token 总数。

应用列表展示每个应用的名称/ID、地域、实例状态、接入类型、昨日写数据量和昨日 Token 数等信息。单击操作列中的进入观测，即可进入该应用的详情页。



仪表盘

仪表盘页面提供应用运行的可视化监控面板，通过指标卡片和时序图直观展示应用各项核心指标。仪表盘包含四个子 Tab：**总览、性能、成本&Token** 和**应用观测**。



总览

总览面板提供应用运行状态的全局视角，适合日常巡检和快速定位异常。

功能	说明
核心指标卡片	展示 Agent 数量、请求数、模型调用次数、输入/输出 Tokens、Tokens 总量等核心 KPI，并显示环比变化百分比。
请求数 / 错误数时序	请求量和错误数随时间的变化趋势图，快速发现突增突降。
模型调用次数时序	底层大模型的调用量时序变化。
请求数 Top10（按 Agent）	按 Agent 维度统计请求量排行，定位高频调用的 Agent。
模型调用次数 Top10（按 Agent + 模型）	按 Agent 和模型组合维度统计调用量排行。
Tokens 总量分布	单次请求 Token 消耗的直方图分布，了解 Token 使用集中区间。
Tokens 总量按 Agent/模型 Top10	识别高成本 Agent 和高消耗模型。
模型调用错误数时序	观察模型错误是否存在规律性波动。
每秒输出 Tokens（按模型）	评估各模型的输出吞吐率。

模型首 Token 平均耗时（TTFT）	反映用户感知的响应速度。
请求平均消耗模型次数/Tokens（按 Agent）	了解各 Agent 的模型调用复杂度和单次成本。

性能

性能面板聚焦于延迟和吞吐分析，帮助发现和优化性能瓶颈。

功能	说明
核心指标时序	请求数、错误数、平均耗时、模型调用次数、模型错误数、模型平均耗时的趋势变化。
请求耗时分布	请求耗时的直方图，标注 P50 分位线，直观展示耗时集中区间。
请求耗时分位	P50、P90、P99 耗时的时序趋势，监控长尾延迟变化。
模型调用耗时分布	模型层面的延迟直方图分布。
模型平均耗时时序	各模型平均耗时随时间变化，发现模型性能劣化。
模型调用次数/平均耗时 Top10	按模型统计调用量和耗时排行，识别慢模型。
按 Agent + 模型组合 Top10	请求数和模型调用次数按组合维度排行。

成本&Token

成本&Token 面板帮助您监控和管理 Token 消耗趋势。

功能	说明
输入/输出 Tokens	选定时间范围内的输入和输出 Token 总量。
Token 总量时序	输入和输出 Token 数量随时间的变化趋势。
Token 分位时序	Token 消耗的 P50/P90/P99分位变化趋势，识别异常高消耗请求。

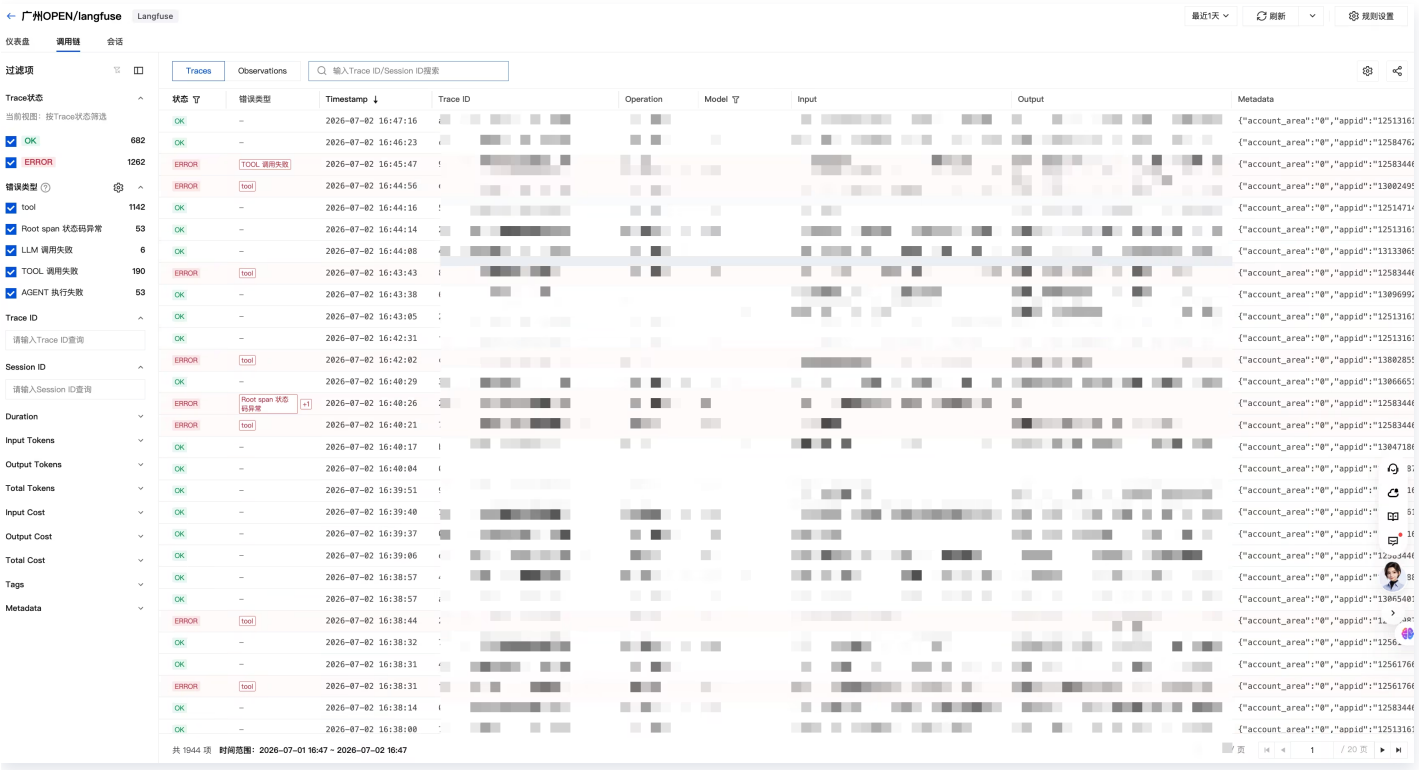
应用观测

应用观测面板提供应用内部组件级别的运行分析，按操作类型（Observation Type）维度展示各组件的行为和性能。

功能	说明
操作类型分布	展示 generation、guardrail、tool、chain、agent、retriever、event 等各操作类型的调用占比。
Observation 平均耗时	各操作类型的平均执行耗时对比，识别慢组件。
LLM 调用次数时序	LLM 调用量随时间的变化趋势。
LLM 耗时分位时序	LLM 调用耗时的 P50/P90/P99分位趋势。
LLM 调用模型排行	各 LLM 模型的调用次数排行。
LLM 平均耗时 Top10	各 LLM 模型的平均耗时排行，定位慢模型。

调用链

调用链页面提供请求级别的细粒度追踪能力，帮助您深入分析单次请求的完整执行路径。支持 Traces 和 Observations 两个视图切换。



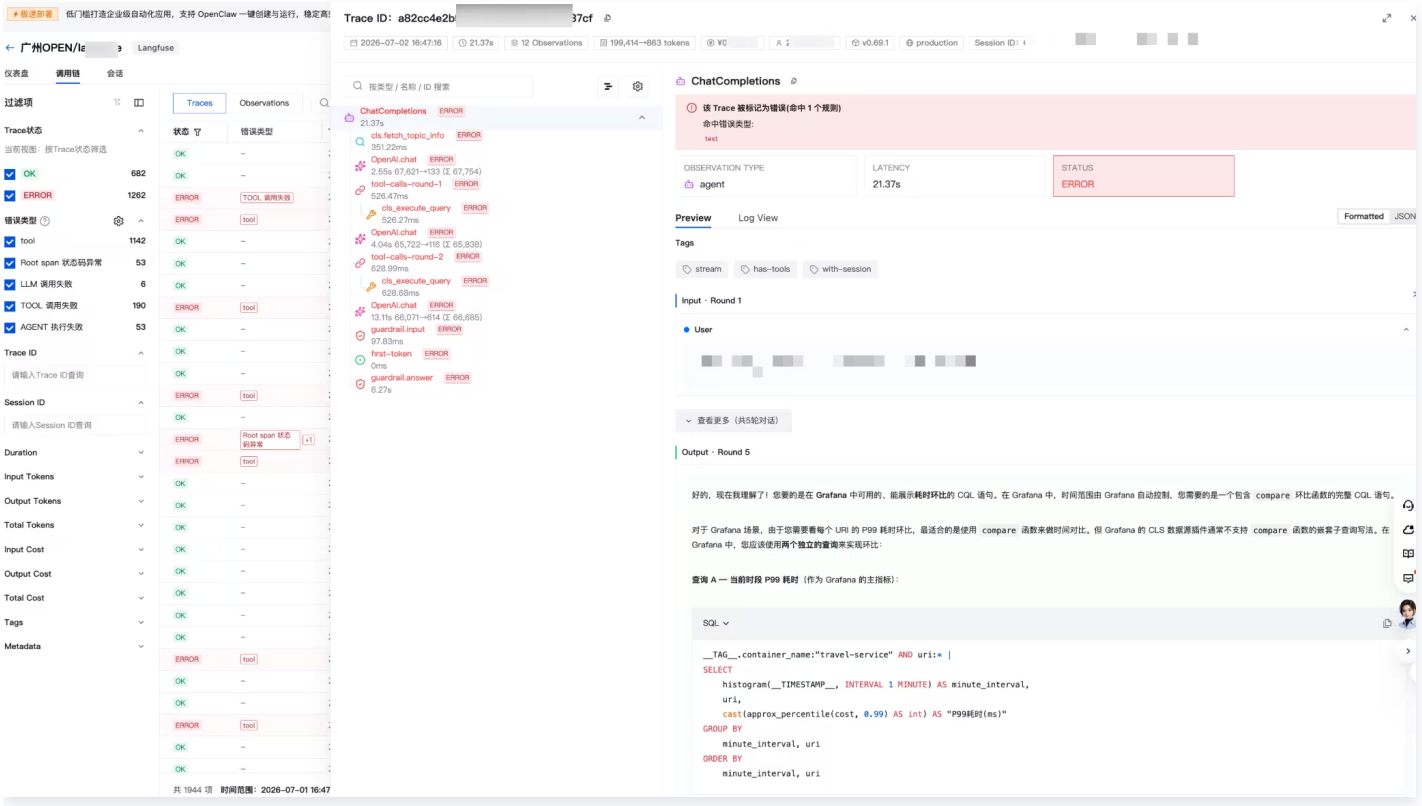
Trace 列表与筛选

功能	说明
Trace 状态筛选	按成功（OK）或错误（ERROR）状态过滤，快速聚焦异常请求。

错误类型筛选	按具体错误类型（tool 调用失败、LLM 调用失败、Root span 状态码异常、AGENT 执行失败等）分类过滤。
Trace ID / Session ID 查询	精确查找特定 Trace 或某个会话下的所有 Trace。
Duration 范围筛选	按耗时范围筛选，快速定位慢请求。
列表展示	每条 Trace 显示状态、错误类型、时间戳、Trace ID、Operation 名称、调用模型等信息。

Trace 详情

单击任意一条 Trace 可展开详情视图，完整还原单次请求的调用链路。



功能	说明
基本信息	Trace ID、起始时间、总耗时、Observations 数量、Token 消耗总量（输入→输出）、费用、版本号、Session ID。
调用树	左侧以树形结构展示本次请求的完整调用链路，包括各节点的操作类型（generation/tool/guardrail 等）、耗时和错误状态标识。
节点详情	选中调用树中的任意节点，右侧展示该节点的 Tags、Input/Output 内容、Observation 类型、延迟和执行状态。

错误诊断	ERROR 状态的 Trace 会标注错误信号来源（如"由以下上浮到 Trace 的 Observation 错误触发"），帮助快速定位根因。

会话

会话页面以 Session 维度聚合展示用户与 AI 应用的交互过程，帮助您了解用户的使用模式和对话质量。

功能	说明
Session ID 搜索	支持精确搜索特定会话。
Session Duration 筛选	按会话持续时长范围过滤。
Traces Count 筛选	按会话包含的 Trace 数量过滤。
Input / Output / Total Tokens 筛选	按会话级别的 Token 消耗量过滤。
Input / Output Cost 筛选	按会话级别的成本过滤。
会话列表	展示 Session ID、创建时间、持续时长、模型、用户输入摘要等信息。
会话详情	选中会话后展示该 Session 下的所有 Trace 调用树，右侧展示 Trace 详细信息（Tags、Input/Output 内容、Token 消耗、费用）。

使用 Langfuse SDK 上报 Trace 数据到 CLS

最近更新时间：2026-07-28 14:15:31

操作场景

Langfuse SDK 可用于追踪 LLM 应用中的请求链路、模型调用、输入输出、用户信息、会话信息、耗时和异常状态。您可以将 Langfuse SDK 生成的 OpenTelemetry Trace 数据上报到日志服务（Cloud Log Service，CLS），并在 CLS 中进行检索分析和链路排查。

本文介绍如何将 Langfuse SDK 生成的 Trace 数据上报到 CLS。本文提供以下接入方式：

- 通过 Skill 快速接入与分析：如果您使用支持 Skill 的 AI 工具，完成 CLS 日志主题准备后，可由 AI 自动识别项目语言并生成接入代码。
- 手动配置接入：按步骤完成环境变量配置、代码接入和上报验证。

接入方式	数据路径	推荐场景
OTLP/HTTP 标准直传	应用 > 标准 OTLP HTTP Exporter > CLS	手动接入推荐首选。适用于新项目、快速试用、希望减少自定义代码、希望按 OpenTelemetry 标准语义上报 Trace 的场景。
自定义 SpanExporter 直传	应用 > 自定义 `SpanExporter` > CLS LogItem > CLS	生产增强方案。适用于需要使用 CLS SDK 的压缩、批处理能力，或需要完全控制写入字段的高吞吐场景。Skill 可生成该方式的接入代码。

> **说明：**
> 本方案不需要部署 Langfuse Server，也不需要将数据发送到 Langfuse Cloud。业务代码仍使用 Langfuse SDK 生成 Trace，数据出口改为 CLS。

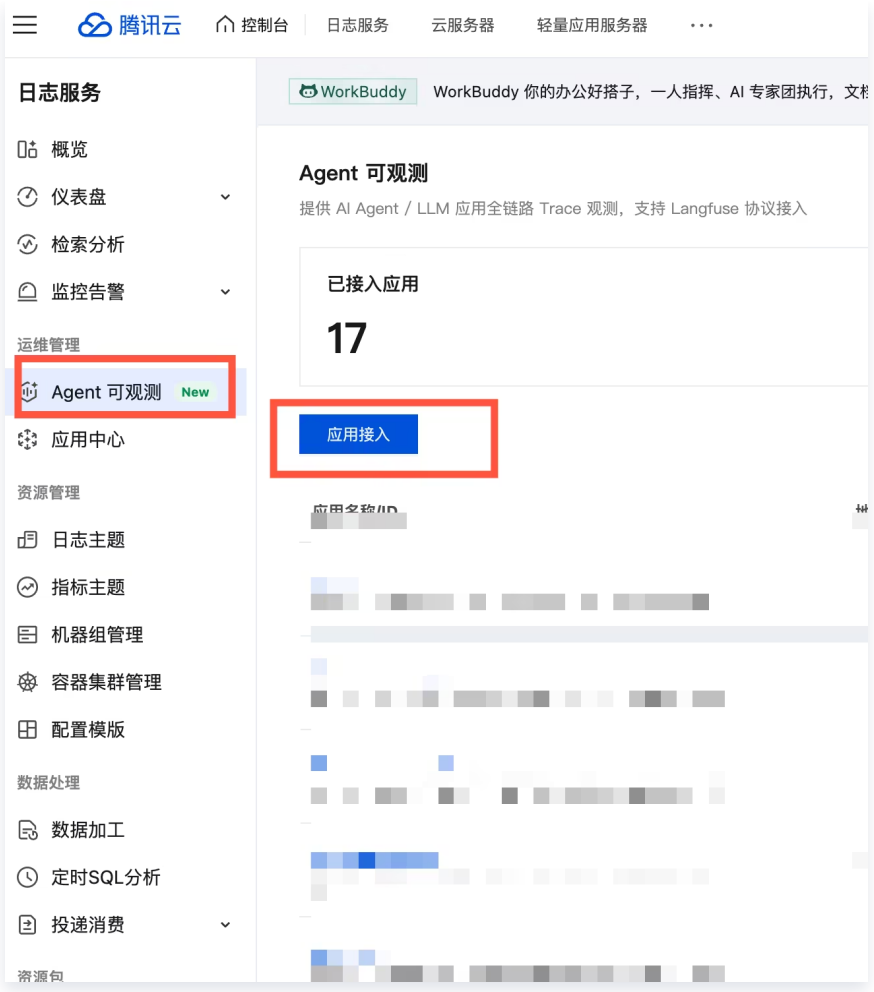
前提条件

开始接入前，请确保已完成以下准备工作：

- 已开通 日志服务 CLS。
- 已准备具备 CLS 写入权限的访问凭证，例如 CAM 子账号、CAM Role 或临时密钥。云 API 密钥信息请前往 API 密钥管理 获取。
- 应用使用 TypeScript/Node.js 或 Python，并已接入或计划接入 Langfuse SDK。

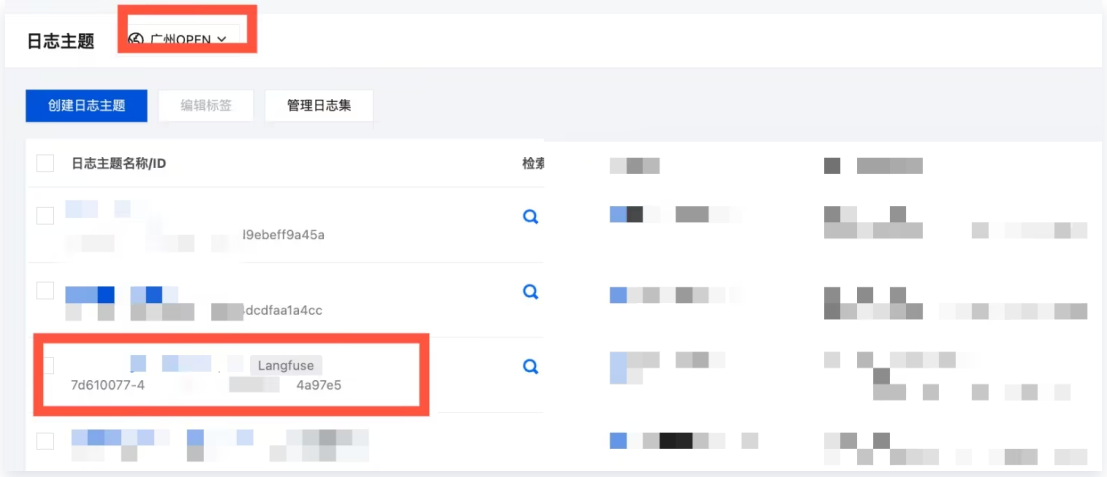
准备 CLS 日志主题

1. 登录 日志服务控制台，在左侧导航栏中，选择 **Agent 可观测**，然后单击**应用接入**。



2. 填写应用名称，选择地域和日志集，单击**确定**。

3. 在左侧导航栏中，选择日志主题，切换到您在上一步中选择的地域，找到名称为 {应用名称}-trace-topic（带有应用观测 tag 标签）的日志主题，复制其日志主题 ID。



4. 根据日志主题所在地域，获取 API 上传日志域名，详情请参见 地域和访问域名 。

通过 Skill 快速接入与分析

如果您使用支持 Skill 的 AI 工具，可使用 `cls-agent-obs-setup` Skill 自动完成接入。该 Skill 可生成**自定义 SpanExporter 直传方案**，并处理字段映射、时间单位、批量发送、退出 flush、`resource` 与 `attribute` 拆分和验证说明。
使用前请准备以下信息：

参数	说明
`CLS_DEFAULT_REGION`	日志主题所在地域，例如 `ap-guangzhou` 。
`CLS_TOPIC_ID`	准备 CLS 日志主题时复制的日志主题 ID。
`TENCENTCLOUD_SECRET_ID`	腾讯云访问凭证 SecretId。
`TENCENTCLOUD_SECRET_KEY`	腾讯云访问凭证 SecretKey。

在 AI 工具中输入以下提示词：

```
请使用腾讯云 CLS Agent 可观测接入 Skill：
https://skillhub.cn/skills/tencent-cls-agent-obs-setup

帮我把当前 AI 应用接入腾讯云 CLS Agent 可观测。

观测 SDK：Langfuse
CLS_DEFAULT_REGION={替换为你的地域，如 ap-guangzhou}
TENCENTCLOUD_SECRET_ID={替换为你的 SecretId}
TENCENTCLOUD_SECRET_KEY={替换为你的 SecretKey}
CLS_TOPIC_ID={替换为你的日志主题 ID}

请识别当前项目语言和框架，补齐依赖、初始化代码、环境变量模板，并接入 CLS SpanExporter。若需要 CLS_ENDPOINT，请根据地域和项目语言自动推导。完成后请告诉我如何验证 Trace 是否成功上报。
```

手动配置接入

步骤1：配置环境变量

请在应用项目中创建或更新 `.env` 文件。请确保 `.env` 已加入 `.gitignore` ，避免密钥泄露。

- **OTLP/HTTP 标准直传：**

```
CLS_DEFAULT_REGION=ap-guangzhou
CLS_TOPIC_ID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

```
TENCENTCLOUD_SECRET_ID=AKIDxxxxxxxx
TENCENTCLOUD_SECRET_KEY=xxxxxxxx
SERVICE_NAME=my-llm-app

# Python 额外需要。Langfuse Python SDK 启动时会校验密钥是否存在，此处可填写任意非空占位值。
LANGFUSE_PUBLIC_KEY=pk-placeholder
LANGFUSE_SECRET_KEY=sk-placeholder
```

- **自定义 SpanExporter 直传：**

TypeScript/Node.js

```
CLS_ENDPOINT=ap-guangzhou.cls.tencentcs.com
CLS_TOPIC_ID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
TENCENTCLOUD_SECRET_ID=AKIDxxxxxxxx
TENCENTCLOUD_SECRET_KEY=xxxxxxxx
TENCENTCLOUD_TOKEN=
CLS_DEFAULT_REGION=ap-guangzhou
SERVICE_NAME=my-llm-app
```

Python

```
CLS_ENDPOINT=https://ap-guangzhou.cls.tencentcs.com
CLS_TOPIC_ID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
TENCENTCLOUD_SECRET_ID=AKIDxxxxxxxx
TENCENTCLOUD_SECRET_KEY=xxxxxxxx
CLS_DEFAULT_REGION=ap-guangzhou
SERVICE_NAME=my-llm-app

# Python 额外需要。Langfuse Python SDK 启动时会校验密钥是否存在，此处可填写任意非空占位值。
LANGFUSE_PUBLIC_KEY=pk-placeholder
LANGFUSE_SECRET_KEY=sk-placeholder
```

说明：

> OTLP/HTTP 标准直传不需要 `CLS_ENDPOINT`，Endpoint 由 `CLS_DEFAULT_REGION` 拼接而成。自定义 SpanExporter 使用 CLS SDK 写入日志，需要配置 `CLS_ENDPOINT`。不同语言 SDK 对 Endpoint 的格式要求不同，请按对应语言示例填写。

常见地域 Endpoint 如下：

地域	公网 Endpoint	内网 Endpoint
广州	`ap-guangzhou.cls.tencentcs.com`	`ap-guangzhou.cls.tencentyun.com`
上海	`ap-shanghai.cls.tencentcs.com`	`ap-shanghai.cls.tencentyun.com`
北京	`ap-beijing.cls.tencentcs.com`	`ap-beijing.cls.tencentyun.com`
新加坡	`ap-singapore.cls.tencentcs.com`	`ap-singapore.cls.tencentyun.com`

> **说明：**
> 公网 Endpoint 适用于公网访问场景。内网 Endpoint 适用于同地域 VPC/CVM 环境，可降低网络延迟并避免公网流量。

步骤2：接入应用

手动接入时，建议先使用 OTLP/HTTP 标准直传 完成快速接入；如您需要使用 CLS SDK 的压缩、批处理或自定义字段能力，再选择 自定义 SpanExporter 直传。如果您通过 Skill 自动接入，可生成自定义 SpanExporter 方案。

方式一：OTLP/HTTP 标准直传

> **说明：**
> 不同语言的 OpenTelemetry SDK 对 Endpoint 参数的要求不同。本文 TypeScript/Node.js 和 Python 示例使用完整 OTLP Trace URL，即 `https://{CLS_DEFAULT_REGION}.cls.tencentcs.com/v1/traces`。

TypeScript/Node.js

1. 安装依赖。

```
npm install @langfuse/tracing @langfuse/otel @langfuse/openai \
  @opentelemetry/sdk-node \
  @opentelemetry/exporter-trace-otlp-http \
```

```
  dotenv
```

2. 创建 `instrumentation.ts` 。

```
import "dotenv/config";

import { NodeSDK } from "@opentelemetry/sdk-node";
import { OTLPTraceExporter } from "@opentelemetry/exporter-trace-
otlp-http";
import { LangfuseSpanProcessor } from "@langfuse/otel";

const secretId = process.env.TENCENTCLOUD_SECRET_ID!;
const secretKey = process.env.TENCENTCLOUD_SECRET_KEY!;
const region = process.env.CLS_DEFAULT_REGION!;
const topicId = process.env.CLS_TOPIC_ID!;

const auth =
Buffer.from(`${secretId}:${secretKey}`).toString("base64");

export const sdk = new NodeSDK({
  spanProcessors: [
    new LangfuseSpanProcessor({
      exporter: new OTLPTraceExporter({
        url: `https://${region}.cls.tencentcs.com/v1/traces`,
        headers: {
          Authorization: `Basic ${auth}`,
          topic_id: topicId,
        },
      }),
      flushAt: 512,
      flushInterval: 5000,
    }),
  ],
});

sdk.start();

process.on("SIGTERM", async () => {
  await sdk.shutdown();
```

```
});
```

3. 在业务入口文件顶部引入 `instrumentation.ts` 。请确保该引入早于 OpenAI、LangChain 等会被 instrumentation patch 的模块。

```
import { sdk } from "./instrumentation";

import OpenAI from "openai";
import { observeOpenAI } from "@langfuse/openai";
import { startActiveObservation, propagateAttributes } from
"@langfuse/tracing";

const openai = observeOpenAI(new OpenAI());

async function chat(userMessage: string) {
  return startActiveObservation("chat", async (span) => {
    span.update({ input: [{ role: "user", content: userMessage }]
});

    return propagateAttributes(
      { userId: "u-123", sessionId: "s-456", tags: ["model:gpt-
4o"] },
      async () => {
        const res = await openai.chat.completions.create({
          model: "gpt-4o",
          messages: [{ role: "user", content: userMessage }],
        });
        span.update({ output: res.choices[0].message });
        return res;
      },
    );
  });
}

await chat("你好");
await sdk.shutdown();
```

Python

1. 安装依赖。

```
pip install langfuse opentelemetry-sdk opentelemetry-exporter-otlp
python-dotenv
```

2. 创建 `instrumentation.py` 。该文件必须早于任何 `langfuse` 模块导入。

```
import os
import base64
from dotenv import load_dotenv

from opentelemetry import trace
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.http.trace_exporter import
OTLPSpanExporter

load_dotenv()

secret_id = os.environ["TENCENTCLOUD_SECRET_ID"]
secret_key = os.environ["TENCENTCLOUD_SECRET_KEY"]
region = os.environ["CLS_DEFAULT_REGION"]
topic_id = os.environ["CLS_TOPIC_ID"]

auth = base64.b64encode(f"{secret_id}:
{secret_key}".encode()).decode()

provider = TracerProvider(
    resource=Resource.create({
        "service.name": os.environ.get("SERVICE_NAME", "my-llm-
app")
    })
)
provider.add_span_processor(
    BatchSpanProcessor(
        OTLPSpanExporter(
```

```
endpoint=f"https://{region}.cls.tencentcs.com/v1/traces",
            headers={
                "Authorization": f"Basic {auth}",
                "topic_id": topic_id,
            },
        ),
        max_export_batch_size=512,
        schedule_delay_millis=5000,
    )
)
trace.set_tracer_provider(provider)
```

3. 在业务入口中引入 `instrumentation.py` 。

```
import instrumentation

from langfuse import observe, propagate_attributes, get_client
from langfuse.openai import openai


@observe(name="chat", as_type="generation")
def chat(user_message: str):
    with propagate_attributes(user_id="u-123", session_id="s-456",
tags=["model:gpt-4o"]):
        return openai.chat.completions.create(
            model="gpt-4o",
            messages=[{"role": "user", "content": user_message}],
        )


if __name__ == "__main__":
    chat("你好")
    get_client().flush()
```

进阶配置

- **采样配置：**您可以在 `TracerProvider` 上配置采样策略，控制上报到 CLS 的 Trace 数据量。

采样器	说明	适用场景
`ALWAYS_ON`	全量采样	测试环境或低流量环境。
`ALWAYS_OFF`	全部不采样	临时关闭 Trace。
`TraceIdRatioBased(0.1)`	按比例采样 10%	生产环境控制写入量。
`ParentBased(root)`	根据父 Span 决定是否采样	分布式链路，保证整条链路采样一致。

TypeScript/Node.js

1. 安装额外依赖。

```
npm install @opentelemetry/sdk-trace-base
```

2. instrumentation.ts 中配置采样器。

```
import {
  ParentBasedSampler,
  TraceIdRatioBasedSampler,
} from "@opentelemetry/sdk-trace-base";

export const sdk = new NodeSDK({
  sampler: new ParentBasedSampler({
    root: new TraceIdRatioBasedSampler(0.1),
  }),
  // 其他配置保持不变
});
```

Python

在 instrumentation.py 中配置采样器。

```
from opentelemetry.sdk.trace.sampling import ParentBased,
TraceIdRatioBased
```

```
provider = TracerProvider(
    sampler=ParentBased(TraceIdRatioBased(0.1)),
    resource=Resource.create({
        "service.name": os.environ.get("SERVICE_NAME", "my-llm-app")
    })
)
```

- **跨地域上报**：如果上报链路需要跨地域，可在 OTLP 请求 Header 中添加源地域标识。

TypeScript/Node.js

```
new OTLPTraceExporter({
  url: `https://${region}.cls.tencentcs.com/v1/traces`,
  headers: {
    Authorization: `Basic ${auth}`,
    topic_id: topicId,
    "x-cross-region": "ap-guangzhou",  // 源地域标识
  },
})
```

Python

```
OTLPSpanExporter(
    endpoint=f"https://{region}.cls.tencentcs.com/v1/traces",
    headers={
        "Authorization": f"Basic {auth}",
        "topic_id": topic_id,
        "x-cross-region": "ap-guangzhou",  # 源地域标识
    },
)
```

- **关闭 HTTPS（仅测试环境）**。

TypeScript/Node.js

可传入 http:// 协议的 URL 来使用明文 HTTP。

```
url: `http://${region}.cls.tencentcs.com/v1/traces`,
```

Python

可使用 insecure=True 参数。

```
OTLPSpanExporter(
    endpoint=f"http://{region}.cls.tencentcs.com/v1/traces",
    insecure=True,    # 仅测试环境
    # ...
)
```

方式二：自定义 SpanExporter 直传

如您需要使用 CLS SDK 的压缩、批处理能力，或需要完全控制写入 CLS 的字段，可使用自定义 `SpanExporter` 。该方式会把 OTel Span 手动转换为 CLS LogItem 后写入 CLS。

> **注意：**
> 自定义 SpanExporter 不是 OTLP 标准协议上报。请将 `span.attributes` 和 `span.resource.attributes` 分开映射，分别写入 `attribute` 和 `resource` 字段，不要将 resource 合并到 `attribute` 中。

TypeScript/Node.js

1. 安装依赖。

```
npm install @langfuse/tracing @langfuse/otel @langfuse/openai \
  @opentelemetry/sdk-node @opentelemetry/sdk-trace-base \
  @opentelemetry/api @opentelemetry/core \
  tencentcloud-cls-sdk-nodejs dotenv
```

2. 创建 `cls-span-exporter.ts` 。

```
import {
```

```
  SpanExporter,
  ReadableSpan,
} from "@opentelemetry/sdk-trace-base";
import { ExportResult, ExportResultCode, hrTimeToNanoseconds }
from "@opentelemetry/core";
import {
  Producer,
  LogItem,
  Content,
  TencentCloudClsSDKException,
} from "tencentcloud-cls-sdk-nodejs";

export interface ClsExporterOptions {
  endpoint: string;
  topicId: string;
  secretId: string;
  secretKey: string;
  token?: string;
}

export class TencentCLSSpanExporter implements SpanExporter {
  private producer: Producer;

  constructor(opts: ClsExporterOptions) {
    this.producer = new Producer({
      endpoint: opts.endpoint,
      topic_id: opts.topicId,
      credential: {
        secretId: opts.secretId,
        secretKey: opts.secretKey,
        token: opts.token ?? "",
      },
      time: 2,
      count: 1000,
      maxMemLogCount: 10000,
      onSendLogsError: (err) => console.warn("[cls] send error",
err),
    });
  }
```

```
  export(spans: ReadableSpan[], cb: (r: ExportResult) => void):
void {
    Promise.all(spans.map((span) =>
this.producer.send(this.spanToLogItem(span))))
      .then(() => cb({ code: ExportResultCode.SUCCESS }))
      .catch((err: Error) => {
        if (err instanceof TencentCloudClsSDKException) {
          console.warn("[cls] sdk exception:", err.toString());
        }
        cb({ code: ExportResultCode.FAILED, error: err });
      });
  }

  private spanToLogItem(span: ReadableSpan): LogItem {
    const item = new LogItem();
    const endNs = hrTimeToNanoseconds(span.endTime);
    const startNs = hrTimeToNanoseconds(span.startTime);
    item.setTime(Math.floor(endNs / 1e6));

    const ctx = span.spanContext();
    item.pushBack(new Content("traceID", ctx.traceId));
    item.pushBack(new Content("spanID", ctx.spanId));
    item.pushBack(new Content("parentSpanID",
span.parentSpanContext?.spanId ?? ""));
    item.pushBack(new Content("name", span.name));
    item.pushBack(new Content("kind", String(span.kind)));
    item.pushBack(new Content("start", String(startNs)));
    item.pushBack(new Content("end", String(endNs)));
    item.pushBack(new Content("duration", String(endNs -
startNs)));
    item.pushBack(new Content("statusCode",
String(span.status.code)));
    item.pushBack(new Content("statusMessage", span.status.message
?? ""));
    item.pushBack(new Content("attribute",
JSON.stringify(span.attributes)));
    item.pushBack(new Content("resource",
JSON.stringify(span.resource.attributes)));
    return item;
  }
```

```
  shutdown(): Promise<void> {
    return Promise.resolve();
  }

  forceFlush(): Promise<void> {
    return Promise.resolve();
  }
}
```

3. 创建 `instrumentation.ts` 。

```
import "dotenv/config";

import { NodeSDK } from "@opentelemetry/sdk-node";
import { LangfuseSpanProcessor } from "@langfuse/otel";
import { TencentCLSSpanExporter } from "./cls-span-exporter";

export const sdk = new NodeSDK({
  spanProcessors: [
    new LangfuseSpanProcessor({
      exporter: new TencentCLSSpanExporter({
        endpoint: process.env.CLS_ENDPOINT!,
        topicId: process.env.CLS_TOPIC_ID!,
        secretId: process.env.TENCENTCLOUD_SECRET_ID!,
        secretKey: process.env.TENCENTCLOUD_SECRET_KEY!,
        token: process.env.TENCENTCLOUD_TOKEN,
      }),
      flushAt: 512,          // 攒批阈值
      flushInterval: 5,      // 秒
    }),
  ],
});

sdk.start();

process.on("SIGTERM", () => sdk.shutdown());
```

Python

1. 安装依赖。

```
pip install langfuse opentelemetry-sdk tencentcloud-cls-sdk-python
python-dotenv
```

2. 创建 `cls_span_exporter.py` 。

```
import json
from typing import Sequence

from opentelemetry.sdk.trace import ReadableSpan
from opentelemetry.sdk.trace.export import SpanExporter,
SpanExportResult
from tencentcloud.log.cls_pb2 import LogGroupList
from tencentcloud.log.logclient import LogClient
from tencentcloud.log.logexception import LogException


class TencentCLSSpanExporter(SpanExporter):
    def __init__(self, endpoint: str, topic_id: str, secret_id:
str, secret_key: str):
        self.client = LogClient(endpoint, secret_id, secret_key)
        self.topic_id = topic_id

    def export(self, spans: Sequence[ReadableSpan]) ->
SpanExportResult:
        group_list = LogGroupList()
        group = group_list.logGroupList.add()
        group.source = "langfuse-otel"

        for span in spans:
            ctx = span.get_span_context()
            log = group.logs.add()
            log.time = span.end_time // 1_000_000

            def add(key: str, value: str):
                content = log.contents.add()
```

```
                content.key = key
                content.value = value

            add("traceID", format(ctx.trace_id, "032x"))
            add("spanID", format(ctx.span_id, "016x"))
            add("parentSpanID", format(span.parent.span_id,
"016x") if span.parent else "")
            add("name", span.name)
            add("kind", span.kind.name)
            add("start", str(span.start_time))
            add("end", str(span.end_time))
            add("duration", str(span.end_time - span.start_time))
            add("statusCode", span.status.status_code.name)
            add("statusMessage", span.status.description or "")
            add("attribute", json.dumps(dict(span.attributes),
ensure_ascii=False))
            add("resource",
json.dumps(dict(span.resource.attributes), ensure_ascii=False))

        try:
            self.client.put_log_raw(self.topic_id, group_list)
            return SpanExportResult.SUCCESS
        except LogException as err:
            print("[cls] put_log_raw failed:", err)
            return SpanExportResult.FAILURE
        except Exception as err:
            print("[cls] unexpected error:", err)
            return SpanExportResult.FAILURE

    def shutdown(self):
        pass

    def force_flush(self, timeout_millis: int = 30000) -> bool:
        return True
```

3. 创建 `instrumentation.py` 。

```
import os
from dotenv import load_dotenv
```

```
from opentelemetry import trace
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

from cls_span_exporter import TencentCLSSpanExporter

load_dotenv()

provider = TracerProvider(
    resource=Resource.create({
        "service.name": os.environ.get("SERVICE_NAME", "my-llm-
app")
    })
)
provider.add_span_processor(
    BatchSpanProcessor(
        TencentCLSSpanExporter(
            endpoint=os.environ["CLS_ENDPOINT"],
            topic_id=os.environ["CLS_TOPIC_ID"],
            secret_id=os.environ["TENCENTCLOUD_SECRET_ID"],
            secret_key=os.environ["TENCENTCLOUD_SECRET_KEY"],
        ),
        max_export_batch_size=512,
        schedule_delay_millis=5000,
    )
)
trace.set_tracer_provider(provider)
```

4. 在业务入口文件顶部引入 `instrumentation.py`，并在脚本退出前执行 `flush()`。

```
import instrumentation

from langfuse import observe, get_client
from langfuse.openai import openai


@observe(name="chat", as_type="generation")
```

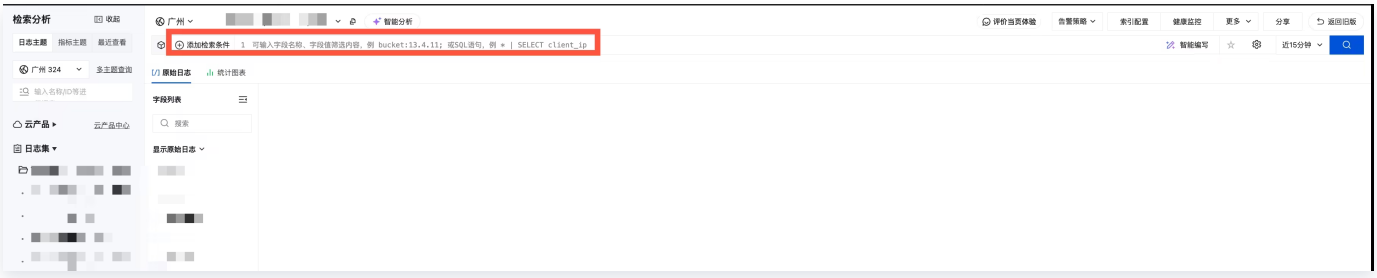
```
def chat(user_message: str):
    return openai.chat.completions.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": user_message}],
    )


if __name__ == "__main__":
    chat("你好")
    get_client().flush()
```

步骤3：验证上报结果

完成接入后，在应用中触发一次 LLM 调用，然后在 CLS 控制台中查询 Trace 数据。

1. 登录 日志服务控制台 ，在左侧导航栏中，选择**检索分析**。
2. 选择用于存储 Trace 数据的日志主题。
3. 在检索框中输入以下语句，查询最新上报的 Trace 数据。



```
* | SELECT traceID, spanID, name, duration, statusCode ORDER BY
__TIMESTAMP__ DESC LIMIT 10
```

也可以查询指定调用链：

```
traceID:"4bf92f3577b34da6a3ce929d0e0e4736"
```

如需按模型统计平均耗时，请确保模型字段已写入 `attribute` 并已配置索引或使用 JSON 提取函数：

```
* | SELECT json_extract_scalar(attribute,'$."gen_ai.request.model"')
AS model,
         AVG(duration) AS avg_duration_ns
```

```
    GROUP BY model
```

字段说明

使用 OTLP/HTTP 标准直传时，`resource`、`scope`、`span` 等信息会按 OpenTelemetry 标准协议结构上报。使用自定义 SpanExporter 时，需要手动映射字段。
自定义 SpanExporter 建议按以下字段写入 CLS LogItem：

OTel Span 字段	CLS LogItem Key	说明
traceId	traceID	Trace ID，hex 字符串。
spanId	spanID	Span ID，hex 字符串。
parentSpanId	parentSpanID	父 Span ID，根 Span 为空。
span.name	name	Span 名称。
SpanKind	kind	Span 类型。
startTime	start	Span 开始时间，纳秒。
endTime	end	Span 结束时间，纳秒。
endTime – startTime	duration	Span 耗时，纳秒。
status.code	statusCode	Span 状态码。
status.message	statusMessage	Span 状态说明。
attributes	attribute	Span attributes，JSON 字符串。
resource.attributes	resource	Resource attributes，JSON 字符串。
events / links	logs / links	可选。如需保留 Span Events 或 Links，可序列化为 JSON 字符串写入。

⚠ **注意：**
自定义 SpanExporter 中的 `attribute` 只用于保存 `span.attributes`。请勿将 `resource.attributes` 写入 `attribute.resourceAttribute`，应将其作为独立 `resource` 字段写入。

注意事项

- `start`、`end`、`duration` 字段需要保持纳秒单位。请勿将这些字段除以 `1000` 转成微秒，否则耗时会显示为实际值的千分之一。
- 自定义 SpanExporter 中，`log.time` 建议使用 Span 结束时间，例如 TypeScript 中使用 `span.endTime` 计算，Python 中使用 `span.end_time // 1_000_000`。
- 短生命周期脚本退出前需要执行 `flush()` 或 `shutdown()`，避免最后一批 Span 未完成上报。
- 访问凭证请通过环境变量或密钥管理工具注入，不要硬编码到代码中。