OpenAIOps

2025 CCF 国际 AIOps 挑战赛

队伍方案报告

赛道一：基于大模型智能体的微服务根因定位

队伍名称：男团 910

队员名字：汤攀、唐世祥、浦桓齐

队员单位：上海大学

日期：2025 年 8 月 1 日

队伍基本信息表

答辩题目	基于大模型智能体的微服务故障根因定位系统		
队伍名称	男团 910	单位	上海大学
团队成员	汤攀、唐世祥、浦桓齐		
方案简介	本方案针对 2025 CCF 国际 AIOps 挑战赛赛道一"基于大模型智能体的微服务根因定位"问题，提出了一套多模态数据融合的智能故障根因定位系统，通过深度整合日志、链路追踪和系统指标三种监控数据，结合大语言模型推理能力，实现微服务环境下的根因定位。 系统采用模块化解耦架构，由数据预处理、日志故障抽取、链路故障检测、指标故障总结和多模态根因分析五大模块组成。数据预处理模块负责从原始 input.json 中解析关键信息并统一各模态数据时间戳；各故障数据模块根据各模态数据特点独立设计异常检测策略，便于系统模块独立优化升级；最终通过多模态根因分析模块实现智能根因定位。 技术创新主要体现在三个方面：第一，日志故障抽取模块结合预训练的 Drain[1]日志解析算法与多重筛选机制，有效将海量原始日志压缩为高质量故障特征；第二，链路故障检测模块采用 IsolationForest[2]无监督学习算法对 span 之间的调用 duration 进行异常检测，同时结合状态码直接检查实现双重异常识别；第三，指标故障总结模块设计统计对称比率过滤机制和双层级 LLM 分析策略，在大幅降低 token 消耗的同时，实现 node-service-pod 三层次全栈现象总结。 多模态根因分析模块通过精心设计的多模态提示词，将经过精细筛选和智能总结的多模态异常信息进行深度融合，利用大语言模型强大的跨模态理解和逻辑推理能力，输出包含故障组件、根因描述和推理过程的结构化分析结果。消融实验验证了各模态数据的互补价值和系统架构的有效性。 方案在复杂微服务故障场景中表现优异，最终获得 45.01 分的成绩，为微服务环境下的智能运维提供了创新性解决方案，有效验证了大模型在 AIOps 领域的应用价值和实用性。		

一、 总体设计

本方案采用模块化架构设计，整体系统分为五个核心模块：数据预处理模块、日志故障抽取模块、追踪故障检测模块、指标故障总结模块和多模态根因分析模块。各模块间采用松耦合设计，通过函数封装进行数据交互，既保证了系统的整体性，又确保了各模块的独立性和可扩展性。

数据预处理模块承担着系统数据标准化的关键职责。核心功能包括：从 input.json 中解析故障时间段信息；对 log、trace、metric 三种模态的原始数据进行时间戳格式统一，确保后续跨模态数据关联的准确性。

Log 故障抽取模块基于“模板化+规则化”的处理架构。首先构建分层次故障模板体系，设计了多条覆盖不同故障类型的正则表达式规则，包括完整错误消息与错误代码组合、错误类型与请求路径组合、异常类型区分等多个层级。利用预训练的 Drain 模型（error_log-drain.pkl）进行日志模板自动提取，通过模板识别与去重机制，将相同语义的日志条目合并为模板及其出现次数的形式，并可通过 drain3.ini 配置文件精确控制模板匹配参数。该模块实现多重筛选机制：时间范围过滤→error 关键词过滤→注入错误排除→列筛选→模板提取→去重合并→频次排序，有效将海量原始日志压缩为高质量故障特征。

Trace 故障检测模块采用“无监督学习+规则增强”的混合检测策略。核心基于 IsolationForest 算法构建异常检测模型，利用正常时期 parent_pod→chile_pod 调用链的 duration 数据进行模型训练。使用 50 个随机样本提取正常数据，采用 30 秒滑动窗口提取时序特征，训练生成 trace_detectors.pkl 模型文件。检测阶段对故障期间 parent_pod→chile_pod 调用链的 duration 进行异常检测，筛选出明显异常的调用链和其 duration。同时实现 status 组合分析功能，提取 trace 中的 status.code 和 status.message 信息，识别调用失败模式。最终分别输出出现次数最多的前 20 个异常调用组合的统计分析和前 20 个详细的 status 异常信息。

Metric 故障总结模块设计了统计对称比率过滤结合层次化两阶段的 LLM 分析策略。首先对于 Metric 指标进行数学统计分析，并借助对称比率判断正常区间和故障区间数据统计指标是否高度相似，以此过滤正常数据，大幅度降低大模型上下文长度，使其更关注可能的异常数据，过滤后进行两阶段 LLM 分析，第一阶段基于应用性能监控数据（APM）和数据库组件（TiDB）数据，分别进行了 service 级别和 pod 级别的现象总结，

包括微服务的请求相应、异常比例等业务指标，以及数据库附属组件 TiDB、TiKV 和 PD 等相关指标。第二阶段将第一阶段的业务指标现象总结与 Node 级别的基础设施数据相结合，进行 service，pod，node 三层次的现象综合分析。在保证分析深度的同时有效控制计算成本。

多模态根因分析模块核心设计了专门的多模态提示词，支持 log、trace 和 metric 三种模态数据的灵活组合。该模块采用多进程策略提升系统处理效率，集成了完整的容错机制和重试策略。最终输出包含 component、reason、reasoning_trace 的结构化根因分析结果，实现从现象观察到根因推理的完整闭环。

二、 创新性与实用性

本方案在创新性上体现在多层次特征提取与推理架构设计上。首先，日志、调用链与指标模块分别采用针对性较强的处理与筛选策略：Drain 算法结合多级过滤来压缩日志冗余；IsolationForest 与状态码检查互补识别 trace 异常；统计显著性筛选高价值 metric 特征。这种分模态“精准抽取—信息压缩”机制，在保证关键信息完整性的同时，显著降低了后续分析的计算与上下文开销。其次，指标分析部分设计了统计对称比率法过滤机制和双层级现象总结框架，在过滤出可能异常数据后，将服务/实例和容器/节点两类视角有机结合，并在现象层面保留业务与基础设施的关联脉络，为后续根因定位提供直接可用的语义化证据。

在实用性方面，本方案突出了模块化和可扩展性设计，能够灵活适配不同业务与系统环境。各功能模块（日志抽取、调用链检测、指标总结、根因分析）相互解耦，既可独立部署，也可按需组合，方便在不同规模与复杂度的微服务架构中快速落地。在处理效率上，方案通过分层筛选等策略，大幅减少无关数据进入分析环节，降低计算与存储开销，适合高并发、大数据量的生产环境。

三、 详细设计

本方案围绕微服务系统的故障根因定位需求，构建了由数据预处理、log 故障抽取、trace 故障检测、metric 故障总结和多模态根因分析五个模块组成的完整系统结构（如图 1 所示）。系统首先通过数据预处理模块实现输入故障信息的结构化解析和多模态时间戳统一；随后 log 模块从大规模日志多层级筛选，并利用 drain 提取日志模板去重，trace

模块采用 duration 性能与状态双策略识别调用异常，metric 模块借助大模型实现多层级监控指标的现象总结；最终，所有模态数据汇聚至多模态根因分析模块，通过大语言模型的跨模态推理能力，实现对故障组件的定位、原因的解释与证据链条的输出。

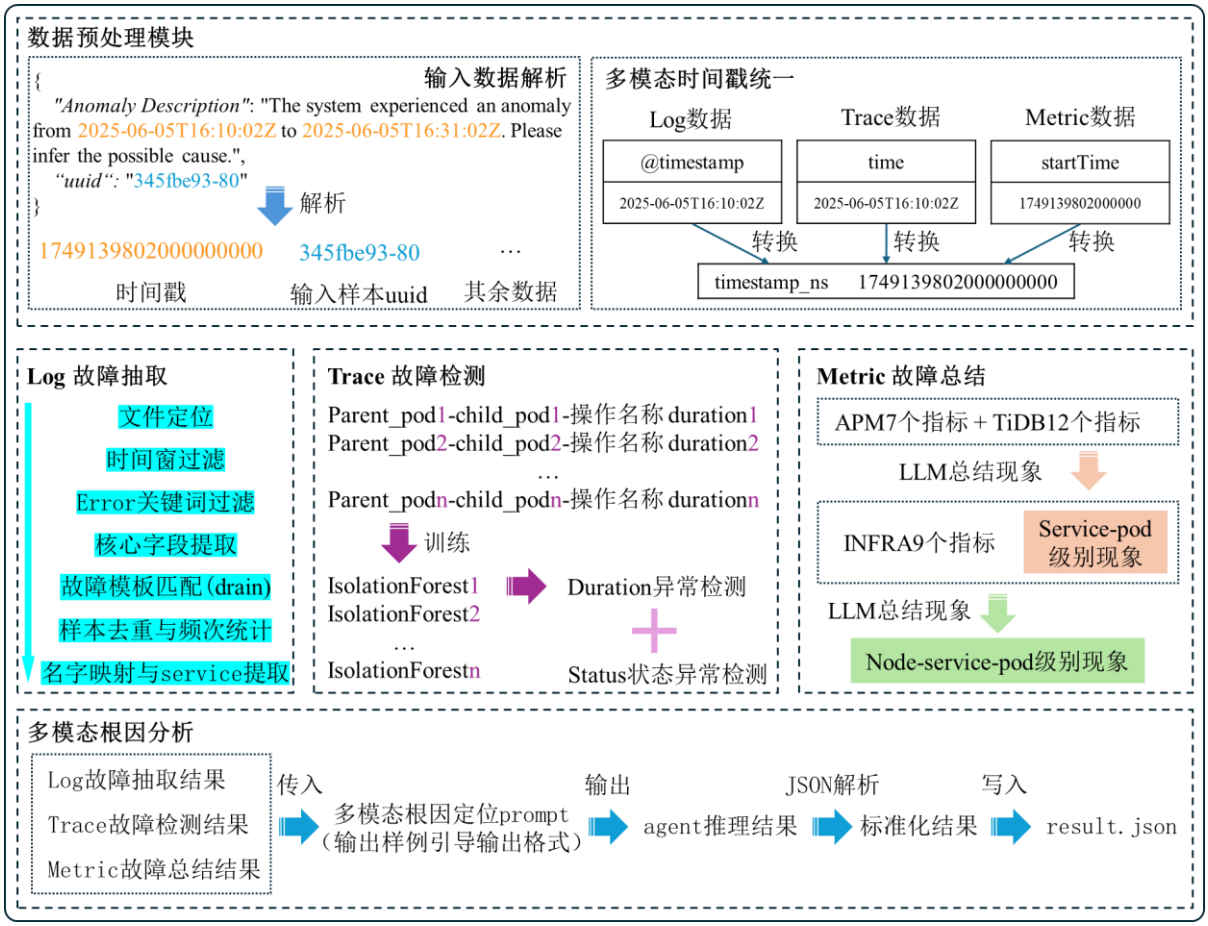

图 1 完整系统结构

1. 数据预处理

数据预处理模块聚焦于输入数据解析和多模态时间戳统一处理两个核心功能，确保后续各模态数据处理的一致性和准确性，为整个故障根因定位系统奠定数据基础。

1.1 输入数据解析设计

输入数据采用基于正则表达式的提取策略，对原始故障描述文件进行结构化处理。系统接收包含故障描述（Anomaly Description）和唯一标识符（uuid）的 JSON 格式输入数据，其中最关键的信息是故障描述中的故障起止时间和 uuid。

为便于后续多模态数据的精准匹配和故障时间段数据筛选，系统首先对输入文件进行关键信息解析提取，核心包括故障唯一标识和故障起止时间，具体如图 2 所示。

时间戳提取机制：系统采用 ISO 8601 时间格式标准，通过正则表达式模式(\d{4}-

\d{2}-\d{2}T\d{2}:\d{2}:\d{2}Z)实现自动化时间戳识别。提取规则遵循“首个匹配为故障起始时间，次个匹配为故障结束时间”的策略。

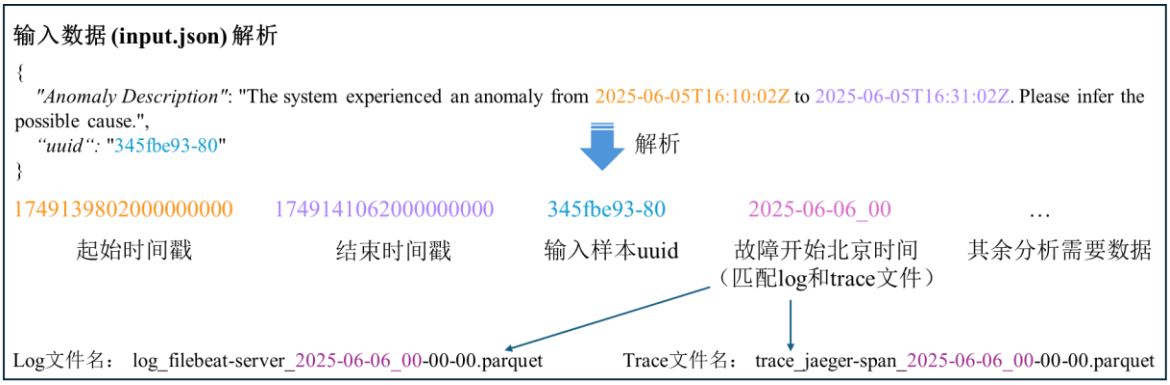

图 2 输入数据解析

时间索引生成：为提升后续数据关联效率，系统构建了多层级时间索引机制。生成“年-月-日_时”格式的时间标识（如 2025-06-06_00），用于快速定位对应的数据文件。同时将故障时间转换为 19 位纳秒级时间戳，为精确的时间范围筛选提供高精度基准，预处理后的 input 数据具体如图 3 所示。

```
uuid,       start_time_beijing, end_time_beijing,   start_timestamp,    end_timestamp,      start_time_hour
345fbe93-80,2025-06-06_00-10-02,2025-06-06_00-31-02,1749139802000000000,1749141062000000000,2025-06-06_00
```

图 3 预处理后的 input 数据

1.2 多模态数据时间戳统一

针对 log、 trace 和 metric 三种数据的不同格式特点，系统地设计了差异化的时间戳统一处理策略，实现跨模态时间基准的标准化，具体如图 4 所示。

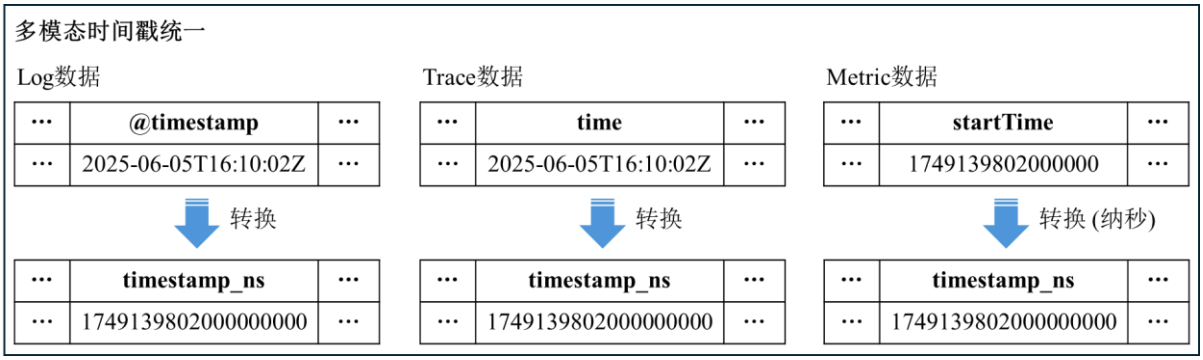

图 4 多模态数据时间戳统一

Log 数据时间戳标准化：log 数据采用 ISO 8601 格式的@timestamp 字段作为时间基准。系统通过时间格式解析，将原始时间戳转换为统一的 19 位纳秒级时间戳。处理完成后按时间戳升序排列，确保数据的时序一致性和后续分析的准确性。

Trace 数据时间戳转换：trace 数据具有独特的时间存储格式，使用 startTime 字段

存储微秒级时间戳。本文通过精度转换，将微秒级时间戳扩展为纳秒级（乘以 1000 倍数），实现与其他模态数据的时间精度对齐。

Metric 数据时间戳处理：metric 数据使用 time 字段存储时间信息，同样遵循 ISO 8601 格式标准。考虑到 metric 数据在存储架构中的多层级分布特点，系统采用递归搜索策略，遍历所有子目录结构，确保完整覆盖分散存储的指标文件。时间戳转换逻辑与日志数据保持一致。

2. Log 故障抽取

日志故障抽取模块是故障根因定位系统的核心组件，采用多层级筛选策略，可将海量原始日志高效转换为结构化故障特征信息。该模块的故障信息压缩功能基于预训练的 Drain 算法模型，通过提取 error 字段日志的模板，并对同一模板的日志进行去重处理，实现对故障日志的有效压缩，从而为后续的多模态故障分析提供高质量的结构化 log 数据输入。

2.1 日志解析原理与模型构建

2.1.1 非结构化日志处理挑战

系统日志作为非结构化信息载体，详细记录了系统的操作行为，但同时包含大量影响分析效率的无关变量（如时间戳、IP 地址、端口号等）。这些变量在日志中虽有重要意义，但在故障模式识别过程中，容易导致大量语义相同但表现形式不同的重复记录。这些重复记录占用了宝贵的大语言模型上下文空间，却无法提供额外的有效信息。

2.1.2 Drain 算法选择与应用

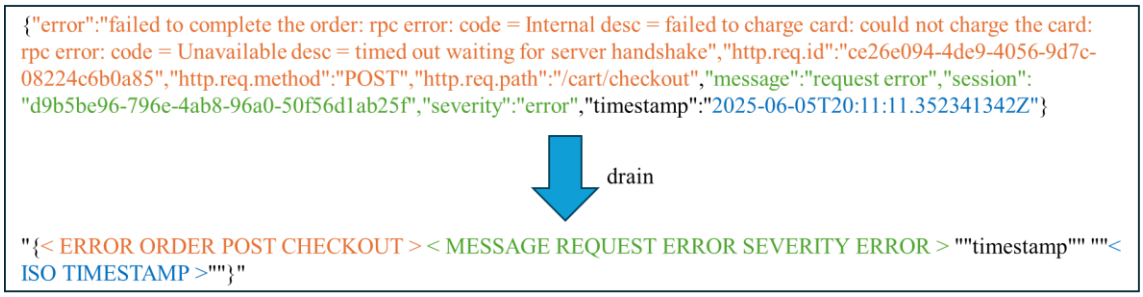

图 5 Drain 提取日志模板

经过对现有日志解析技术的综合评估，本研究采用广泛认可的 Drain 算法作为核心日志解析器。Drain 算法通过构建解析树的方式，能够有效识别日志中的固定部分和可变部分，将具有相同语义模式的日志归类为同一模板。这一过程实现了从非结构化日志到结构化特征的转换，同时压缩了大量冗余的日志。具体而言，Drain 算法提取日志的

模板，并去除无关变量（时间、id 等），将相同模板的日志合并为一个条目，从而有效减少数据量。图 5 展示了日志模板的提取操作，其输出中所有的变量都由 Drain 解析成不变的内容，能减少无关变量的干扰。

2.1.3 预训练模型构建

系统基于 phaseone 中的全量日志数据，筛选出所有包含 error 字段的日志记录作为训练样本，构建 Drain 预训练模型。通过对微服务环境中海量错误日志的学习分析，模型成功提取并构建了 156 个具有代表性的故障模板，较为全面的覆盖了微服务系统中的主要故障模式类型。

2.2 多层级数据筛选处理流程

多层级数据筛选处理流程具体如图 6 所示，从文件定位开始，经时间窗过滤、error 关键词过滤、核心字段提取与重构、故障模板匹配、样本去重与频次统计，到名字映射与 service 提取。最终将海量日志转化为结构化故障特征，同时实现有效的内容压缩。

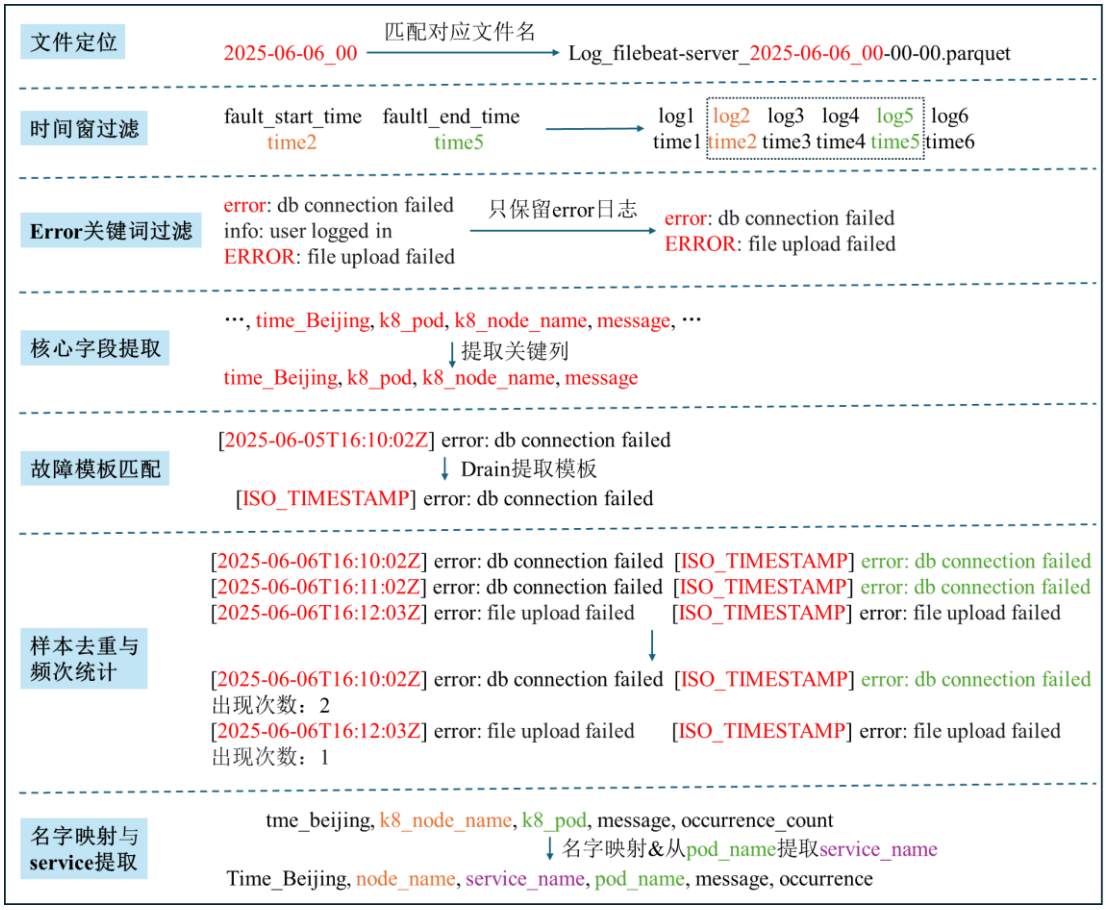

图 6 多层级数据筛选处理流程

2.2.1 文件定位

系统根据预处理部分输出的每项 input 时间信息，采用“年-月-日_时”格式的时间标

识进行文件匹配，如图 7 所示。通过在项目数据目录中搜索对应时间段的日志文件，确保精准定位到故障时间窗口内的数据源。

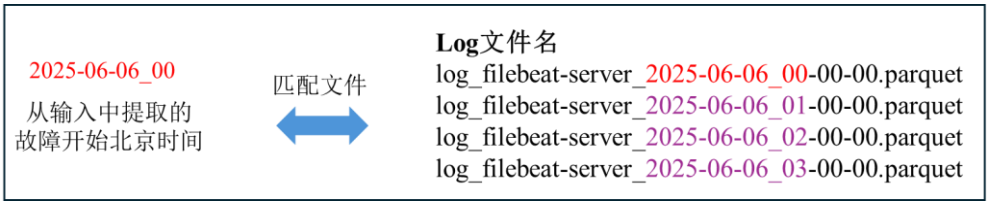

图 7 input 中的时间与 log 文件名匹配

2.2.2 时间窗过滤

基于故障起止时间的纳秒级精度时间戳，对日志数据执行严格的时间边界筛选。系统通过时间戳字段进行范围查询，确保仅保留故障时间段内的日志记录，有效排除时间维度上的无关信息干扰。

2.2.3 Error 关键词过滤

针对日志消息字段实施基于“error”关键词的过滤机制。在进行大量日志数据分析后，观察到故障可能伴随 error 字段的形式大量出现，因此，自动识别和提取包含 error 信息的日志条目，过滤掉正常业务操作日志，从而显著提升后续分析的针对性和效率。

2.2.4 核心字段提取与重构

从原始日志的多维度字段中精确提取分析必需的核心信息，包括时间信息(time_beijing)、容器标识(k8_pod)、节点信息 (k8_node_name) 和错误消息内容 (message)等关键维度。通过字段精简策略减少数据冗余，同时保证信息完整性。

2.2.5 故障模板匹配与标准化

利用预训练的 Drain 模型对每条错误日志进行模板匹配。系统将原始日志消息转换为标准化的模板表示形式，去除无关变量，仅保留核心故障语义内容，实现从个体日志到模式类别的提取。

2.2.6 样本去重与频次统计

针对相同容器和模板组合的重复日志记录（比如：[[2025-06-06 08:00, **frontend-0**, **template0**, message0], [2025-06-06 09:00, **frontend-0**, **template0**, message1]]），采用去重策略。系统保留每种组合的首次出现记录，同时统计该组合的出现频次[2025-06-06 08:00, frontend-0, message0, 出现次数: 2]，通过量化标注为故障严重程度评估提供客观依据。

2.2.7 名字映射与 service 提取

Log 中的 k8_node_name 和 k8_pod 进行映射，转换为 node_name 和 pod_name，并通过解析 pod_name 提取对应的 service 信息（例如：frontend-0→frontend）。最终，系统

将数据重构为标准化的多层次格式，包含 node、service、pod、错误消息和频次统计等维度，为后续的多模态故障分析提供结构化的数据基础。

```
node_name,      service_name,   pod_name,   message,                                          occurrence_count
aiops-k8s-04,   frontend,       frontend-2, "{""error"":""could not retrieve product: ...,  出现次数:9
aiops-k8s-03,   frontend,       frontend-0, "{""error"":""could not retrieve product: ...,  出现次数:81
...
aiops-k8s-04,   frontend,       frontend-2, "{""error"":""failed to get product ...,        出现次数:1
```

图 8 log 故障抽取输出

通过上述系统化的处理流程，log 故障抽取模块能够将原始的海量日志数据高效压缩为高质量的结构化故障特征集合，最终抽取出来的故障示例具体如图 8 所示。该模块不仅显著降低了数据处理复杂度，还保持了故障信息的完整性和准确性，为后续基于智能体的多模态故障根因定位提供 log 层次的数据基础。

3. Trace 故障检测

调用链故障检测模块是微服务系统故障根因定位的关键组件，采用双重异常检测策略，分别从性能维度和状态维度识别微服务调用链中的异常模式。该模块基于 IsolationForest 机器学习算法和状态码直接检查方法，实现对 duration 异常和 status 异常的有效识别，两种识别结果同时传入多模态分析，为后续的多模态故障分析提供高质量的结构化 trace 数据输入。

3.1 双重异常检测原理与模型构建

微服务架构下的分布式调用链包含丰富的性能和状态信息，但异常模式呈现多样化特征。性能异常主要表现为调用链 duration 的明显偏离正常范围，需要通过机器学习方法建立正常行为基线；状态异常则体现为明显的错误码出现，可通过直接检查的方式识别。两种异常检测机制的结合能够提供更丰富的故障特征信息。

3.1.1 IsolationForest 性能异常检测原理

针对 duration 性能异常检测，采用 IsolationForest 无监督学习算法。该算法基于“异常点在特征空间中更容易被孤立”的核心假设，通过构建多个随机分割树来量化数据点的异常程度。IsolationForest 特别适用于连续数值型数据的异常检测，能够在不需要标注异常样本的情况下，自动识别 duration 数据中的明显偏离模式。

3.1.2 状态码直接检查机制

并行构建基于状态码的直接检查机制，无需训练过程。系统直接分析 trace 数据中 tags 中的 status.code 和 status.message 字段，通过简单的条件判断（status.code≠0）筛选出所有异常状态调用。这种方法能够直接捕获系统错误、超时、连接失败等明确的异常

状态和具体信息，为故障根因定位提供确定性的异常证据。

3.1.3 IsolationForest 训练数据构建与模型训练

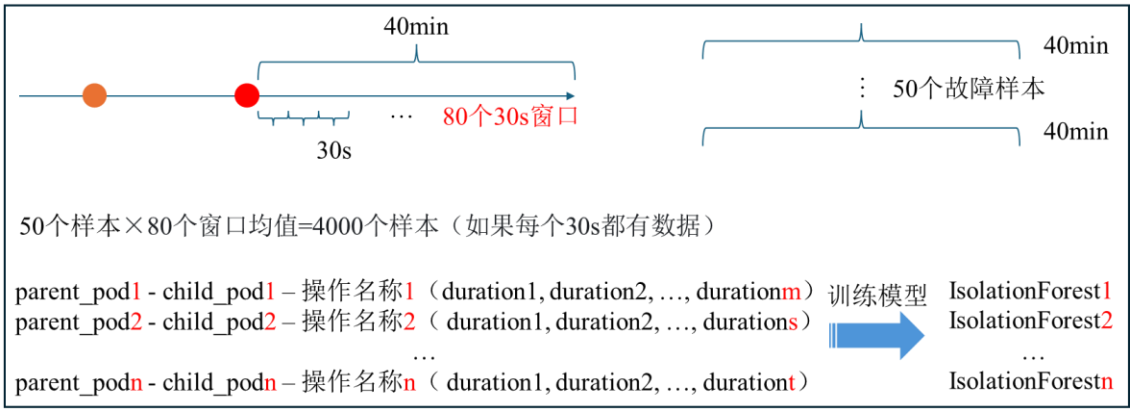

图 9 IsolationForest 训练数据构建与模型训练

IsolationForest 训练数据构建与模型训练如图 9 所示，主要包括下面几部分内容：

训练数据来源：本方案利用 phaseone 阶段的数据构建训练集，通过随机抽样 50 个故障样本，提取每个故障结束后 40 分钟的时间窗口作为“正常时期”数据。这种基于故障恢复后系统行为趋于正常的假设，确保了大部分训练数据都是正常数据。

分组训练策略：按照“parent_pod - child_pod - 操作名称”的服务调用组合对训练数据进行分组，为每个唯一的服务调用模式训练独立的 IsolationForest 检测器。这种细粒度的分组策略考虑了不同服务调用的性能基线差异，提升了异常检测的精准度。

滑动窗口平均化处理：在训练阶段，采用 30 秒滑动窗口对 duration 数据进行事件序列处理。由于一个 30 秒滑动窗口内可能出现多种“parent_pod - child_pod - 操作名称”的调用组合，且每种组合可能出现一次或多次，因此针对窗口内同一种组合，若出现一次及以上，则计算器 duration 的平均值；若未出现，则记为 None。这种处理方式能有效消除单个调用样本的随机波动，让 duration 特征更稳定，从而更准确地反映服务调用的真实性能水平。

异常检测阈值设置：IsolationForest 模型采用 contamination 参数设置为 0.01（1%），表示预期异常样本占总体的比例。同时使用 n_estimators=100 构建 100 个决策树。模型输出异常分数，当分数为-1 时判定为异常，为 1 时判定为正常。

3.2 多维度 trace 数据处理流程

3.2.1 文件定位与时间过滤

系统根据预处理部分解析出的每项 input 时间信息，采用“年-月-日_时”格式进行 trace 文件精确匹配。基于故障起止时间的纳秒级时间戳，对调用链数据执行时间窗口

筛选，确保分析范围覆盖完整的故障时间段。

3.2.2 调用关系映射与结构化提取

从复杂的 trace 结构中提取关键维度信息，包括通过解析 process 字段获取 pod_name、service_name、node_name，利用 spanID 和 references 建立完整的调用链父子关系映射。系统将非结构化 trace 数据转换为包含“parent_pod - child_pod - 操作名称”调用表示。

3.2.3 Duration 异常检测处理流程

预测数据处理：对故障时间段内的所有 trace 数据进行异常检测预测，采用与训练阶段相同的 30 秒滑动窗口策略，计算每个时间窗口内的 duration 平均值。这种一致性处理确保了训练和预测阶段的特征分布一致性。

IsolationForest 异常预测：利用预训练的检测模型对故障期间的 duration 特征进行异常识别。每个服务调用组合使用对应的专用检测器进行预测（若故障期间存在之前未训练的调用组合则忽略不进行预测），输出异常标签（-1 表示异常，1 表示正常）。

Duration 异常结果输出：系统输出前 20 个最频繁的 duration 异常组合，包含以下具体内容：

- node_name：异常发生的节点标识；
- service_name：异常服务名称；
- parent_pod：调用方容器标识；
- child_pod：被调用方容器标识；
- operation_name：具体操作名称；
- normal_avg_duration：正常时期的平均调用时延（来自训练数据统计）；
- anomaly_avg_duration：异常时期的平均调用时延（来自故障阶段实际统计）；
- anomaly_count：异常出现频次（格式：出现次数：N）；

利用 IsolationForest 检测得到的异常 trace 的具体输出实例如图 10 所示：

```
node_name,    service_name,parent_pod,child_pod, operation_name,                                  normal_avg_duration,anomaly_avg_duration,anomaly_count
aiops-k8s-07,frontend,     frontend-1,frontend-1,hipstershop.ProductCatalogService/GetProduct,11203.635050935625, 56375295.28801888,   出现次数:32
aiops-k8s-04,frontend,     frontend-2,frontend-2,hipstershop.ProductCatalogService/GetProduct,11480.30423854856,  57508444.499718376,   出现次数:31
...
aiops-k8s-03,frontend,     frontend-0,frontend-0,hipstershop.ProductCatalogService/GetProduct,11371.846083869175, 56044805.23074916,   出现次数:30
```

图 10 IsolationForest 检测出来的异常 trace 示例输出

3.2.4 Status 状态异常检测处理流程

直接状态检查：对故障时间段内的所有 trace 数据进行状态码检查，无需训练过程。从 trace 的 tags 字段中解析 status.code 和 status.message 信息，通过条件过滤（status.code≠0）直接识别所有异常状态调用。

状态异常模式统计：对识别出的状态异常按照服务调用组合进行分组统计，计算每种异常模式的出现频次。

Status 状态结果输出：系统输出前 20 个最频繁的状态异常组合，包含以下具体内容：

- Node_name：异常发生的节点标识；
- Service_name：异常服务名称（redis 自动映射为 redis-cart）；
- Parent_pod：调用方容器标识；
- Child_pod：被调用方容器标识；
- Operation_name：具体操作名称；
- Status_code：具体的错误状态码；
- Status_message：错误状态的详细描述信息；
- Occurrence_count：状态异常出现频次（格式：出现次数：N）；

利用 status 状态检测得到的异常 trace 具体输出实例如图 11 所示：

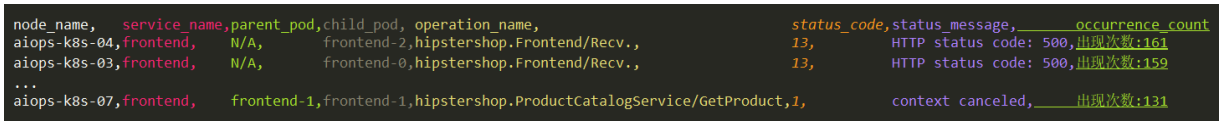

图 11 status 状态检测出来的异常 trace 示例输出

通过上述双重异常检测流程，trace 故障检测模块实现了基于机器学习的异常识别和基于规则的直接异常检查的有机结合。Duration 异常检测通过 IsolationForest 算法提供量化的性能异常分析，status 异常检测通过直接检查提供确定性的错误状态识别。两种检测策略分别输出的前 20 个最重要的异常模式，为后续故障根因定位提供 trace 数据基础。

4. Metric 故障总结

指标故障总结模块是分布式微服务系统故障的全栈监控分析组件，采用基于大语言模型的双层级现象总结策略(见图 12)，通过规则筛选和统计对比方法识别显著异常指标，利用大模型的推理能力对复杂的多为监控数据进行现象归纳和模式识别，为后续的多模态故障根因分析提供高质量的现象描述输入。

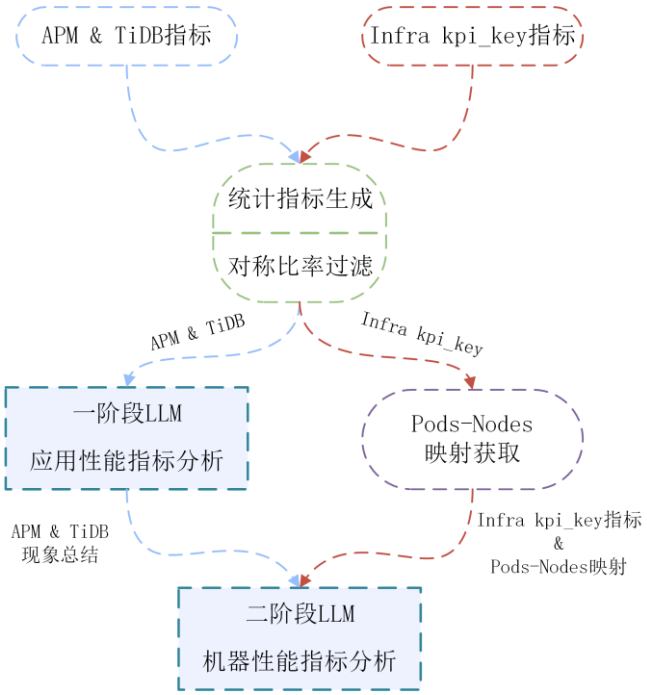

图 12 双层级大模型现象总结示意图

4.1 大模型驱动的现象总结原理与方法构建

4.1.1 传统异常检测方法的局限性

分布式微服务架构下的监控指标具有高维度、多层次和强关联的特征，传统的基于阈值或机器学习的异常检测方法往往只能识别单一指标的数值异常，难以理解指标间的关联关系和业务语义。这些方法缺乏对异常现象的语义化描述能力，无法为后续的多模态故障根因定位提供可解释的现象总结。

4.1.2 大模型现象总结方法

本方案在 metric 数据处理方法中，采用大语言模型作为核心的现象总结工具，充分利用其强大的语义理解和推理归纳能力。通过传递筛选出来的监控数据，让模型自动识别异常变化模式、关联关系和业务影响，输出语义丰富的现象描述。需要特别强调的是，本模块的大模型仅用于现象总结和模式描述，不进行故障判断或根因推断，为后续结合 log 和 trace 等多模态数据的综合根因分析奠定基础。

4.1.3 正常时间段定义

系统采用相对时间窗口的方式定义正常时间段，具体规则为：提取当前故障前一个故障结束后 10 分钟至当前故障开始前的时间段，以及当前故障结束后 10 分钟至下一个故障开始前的时间段作为正常运行时间段。这种定义方式不仅确保正常时间段数据反映微服务系统的稳定运行状态，避免了故障“余波”影响的干扰，还通过选择故障时间段前后的相邻时间窗口，有效减少了每日不同时间段业务波动的影响（如白天访问量高

峰与夜间访问量低估），为对比分析提供更加可靠的性能基线。

4.2 多层级监控指标体系与筛选

4.2.1 Pod-Service 统一分析架构

所提出系统将 pod 和 service 层级进行统一处理，通过解析 Pod 名称自动提取 Service 标识。例如 frontend-0、frontend-1、frontend-2 等 Pod 实例通过去除数字后缀统一归类为 frontend 服务。这种设计简化了微服务架构下的分析复杂度，同时保持了服务级别和实例级别的双重视角，能够有效识别服务级别的共性问题和个别实例的特异性异常。

4.2.2 应用性能监控指标筛选

从微服务 APM 监控数据的众多冗余指标中精选出 7 个核心关键指标：client_error_ratio（客户端错误率）、error_ratio（总体错误率）、request（请求数量）、response（响应数量）、rrt（平均响应时间）、server_error_ratio（服务端错误率）、timeout（超时次数）。这些指标涵盖了微服务性能的核心维度，能够全面反映服务的健康状况和异常特征，同时避免了冗余数据对现象分析效果的影响。

4.2.3 机器性能指标监控指标分层覆盖

构建涵盖容器层和节点层的基础设施监控体系。容器层包含 9 个 Pod 基础设施指标：pod_cpu_usage（pod CPU 使用率）、pod_memory_working_set_bytes（pod 工作集内存使用量）、pod_fs_reads_bytes（pod 文件系统读取字节数）、pod_fs_writes_bytes（pod 文件系统写入字节数）、pod_network_receive_bytes（pod 网络接收字节数）、pod_network_receive_packets（pod 网络接收数据包数）、pod_network_transmit_bytes（pod 网络发送字节数）、pod_network_transmit_packets（pod 网络发送数据包数）、pod_processes（pod 内运行进程数量）。节点层包含 16 个 Node 基础设施指标：node_cpu_usage_rate（节点 CPU 使用率）、node_memory_usage_rate（节点内存使用率）、node_disk_read_bytes_total（磁盘读取字节数）、node_disk_written_bytes_total（磁盘写入字节数）、node_disk_read_time_seconds_total（磁盘读取时间）、node_disk_write_time_seconds_total（磁盘写入时间）、node_filesystem_usage_rate（文件系统使用率）、node_network_receive_bytes_total（网络接收字节数）、node_network_transmit_bytes_total（网络发送字节数）、node_network_receive_packets_total（网络接收数据包总数）、node_network_transmit_packets_total（网络发送数据包总数）、node_sockstat_TCP_inuse（TCP 连接数）等。通过分层监控确保从微服务应用到具体部署的基础机器设施的完整

覆盖。

4.2.4 TiDB 数据库组件监控指标

针对 TiDB 分布式数据库构建专项监控指标体系，覆盖 tidb-tidb、tidb-tikv、tidb-pd 三个核心组件。包括 failed_query_ops（失败查询数）、duration_99th（99 分位请求延迟）、connection_count（连接数）、server_is_up（服务存活节点数）、cpu_usage（CPU 使用率）、memory_usage（内存使用量）、store_up_count（健康 Store 数量）、store_down_count（Down Store 数量）、store_unhealth_count（不健康 Store 数量）、storage_used_ratio（已用容量比）、available_size（可用存储容量）、raft_propose_wait（RaftPropose 等待延迟）、raft_apply_wait（RaftApply 等待延迟）、rocksdb_write_stall（RocksDB 写阻塞次数）等数据库特有指标，实现对数据存储层的专业化监控分析。

4.3 数据处理与异常筛选流程

4.3.1 分层级数据加载与组织

系统根据预处理部分输出的每项 input 日期信息，按照监控层级构建不同的数据路径进行文件定位。APM 业务监控数据位于“年-月-日/metric-parquet/apm/pod”目录，基础设施节点监控数据位于“年-月-日/metric-parquet/infra/infra_node”目录，容器监控数据位于“年-月-日/metric-parquet/infra/infra_pod”目录，TiDB 数据库监控数据位于“年-月-日/metric-parquet/infra/infra_tidb”等目录。通过分层级的文件组织方式安监控维度进行精准的数据定位和加载。

4.3.2 时间窗口精准过滤与数据合并

系统对各层级监控数据执行纳秒级时间窗口筛选，确保分析范围完整覆盖故障时间段。将故障时间段前后 2 个正常时间段的数据进行合并处理，跳过故障时间段前 10 分钟，以消除故障“余波”造成的影响，并通过移除最大最小各 2 个极值的方式消除随机波动影响，构建稳定的统计基线。这种处理方式确保正常期间数据的代表性和稳定性。

4.3.3 基于统计对称比率的显著性筛选

实施基于统计对称比率的显著性筛选机制，计算故障期间与正常期间的中位数和 99 分位数的对称比率。自动过滤变化幅度小于 5%的稳定指标，仅保留变化显著的可能关键异常指标进入大模型分析流程。这种筛选策略能够将海量原始数据压缩为高价值的异常特征集合，大幅度减少大模型的上下文，token 数量减少 50%左右，同时显著提升现象总结的效率和质量，以中位数的对称比率为例，其具体公式定义如下：

$$P50_{symmetric-ratio} = \frac{\left|P50_{fault} - P50_{noraml}\right|}{\overline{\left(P50_{fault} + P50_{noraml}\right)} + \varepsilon}$$

其中 P50 为中位数，$\overline{\left(P50_{fault} + P50_{normal}\right)}$为故障期间和正常期间中位数的平均值，$\varepsilon$ 为极小值。统计描述信息标准化提取

对筛选后的关键指标，利用 pandas 的 describe 方法提取完整的描述统计信息，包括中位数、四分位距、99 分位数等关键统计量。通过标准化的数据表示格式，将不同类型和量纲的监控指标转换为大模型可理解的结构化输入，确保后续现象总结的准确性和一致性。

4.4 双层级大模型现象总结与输出生成

4.4.1 第一层：应用性能监控现象识别与总结

```
请根据提供的APM（应用性能监控）指标数据和TiDB分布式数据库指标数据，描述所有服务在正常期间和故障期间的业务服务性能表现差异现象。

## 系统整体信息
- **微服务总数**: {microservice_count}
- **TiDB组件数**: {tidb_service_count}
- **服务总数**: {len(all_services)}
- **Pod总数**: {total_pods}
- **服务列表**: {all_services}

## APM关键指标说明（微服务）
### 请求响应类指标：{...}
### 异常类指标：{...}

## TiDB关键指标说明（数据库组件）
### TiDB组件指标：{...}
### TiKV组件指标：{...}
### PD组件指标：{...}

## 数据对比表格,特别注意**缺失数据和空数据,代表数据波动极小,通常情况默认正常**:
{json.dumps(combined_json, ensure_ascii=False, indent=2)}

## 现象描述要求
请从以下维度进行现象描述，**仅描述观察到的现象，不做异常判断或结论**：

### 微服务级别现象观察
- 集中在同一类型的微服务中出现的问题，比如emailservice-0, emailservice-1, emailservice-2, 都存在相似的异常数据变化
- 个别Pod的异常表现特征，比如cartservice-0中存在异常的数据变化，而cartservice-1, cartservice-2以及其他pod正常

### TiDB数据库组件现象观察
- TiDB组件（tidb-tidb）、TiKV组件（tidb-tikv）和 PD组件（tidb-pd）的异常数据变化

## 重要提示
**这是为后期综合决策分析提供的系统级现象总结，请控制总结内容在2000字左右，重点突出主要变化现象。**

**总结要求：**
- 如果整体表现正常稳定，请简要说明"系统各项指标表现稳定，未观察到显著变化现象"
- 如果出现明显变化，请重点描述变化显著的服务、指标和现象
- 采用概括性语言，重点关注对业务影响较大的指标变化
- 缺失或为空的数据表示波动极小，可视为正常，无需描述

请基于APM业务监控数据和TiDB数据库监控数据客观描述观察到的现象，控制在2000字以内，为后续综合分析提供简洁有效的现象总结。
```

图 13 业务指标和 TiDB 数据库组件指标综合分析 prompt

系统首先对微服务应用性能监控指标和 TiDB 数据库组件指标进行综合分析。将筛选后的关键异常指标按服务类型组织、每个服务包含其下属多个 Pod 实例数据。例如 emailservice 服务包含 emailservice-0、emailservice-1、emailservice-2 等多个 pod 实例的 client_error_ratio（客户端错误率）、error_ratio（总体错误率）、request（请求数量）、response（响应数量）、rrt（平均响应时间）、server_error_ratio（服务端错误率）、timeout（超时次数）等 APM 指标对比数据。同时整合 TiDB 各组件的 failed_query_ops（失败查询数）、duration_99th（99 分位请求延迟）、connection_count（连接数）等数据库专项指标。构建包含正常期间和故障期间对比数据的结构化表格，通过 service-pod 的层次化组织方

式，调用大语言模型进行第一次现象总结，提示词设计图 13 所示，数据对比表格采用 json 形式，与 markdown 形式相比，json 格式在保持清晰度的同时，大幅度减少 token 使用量，具体形式如图 14 所示。在分析 pod 级别现象的同时，自然总结出服务级别的共性现象和差异化表现。此阶段大模型仅进行现象描述，不做故障判断。

```
{
服务1（如adservice）:{
  服务名称：服务1,
  服务类型：microservice 或 TiDB-component,
  pod数量：3,
  pod列表：[Pod-1, Pod-2, Pod-3],
  具体pods指标：{
    Pod-1（如adservice-1）：{
      具体指标（如error_ratio）:{正常期间中位数：xx.xx, 正常期间四分位距：xx.xx, 正常期间99分位数：xx.xx,
                                故障期间中位数：xx.xx, 故障期间四分位距：xx.xx, 故障期间99分位数：xx.xx },
      其他指标：{…}，……
          }
    Pod-2：{…},
    Pod-3：{…},
        },
服务2:{…},
…
}
```

图 14 业务指标和 TiDB 数据库组件对比数据表格示例

4.4.2 第二层：基础设施机器性能指标综合现象总结与关联分析

在应用性能监控数据的分析基础上，系统进一步整合 pod 容器层和 node 节点层的基础设施的机器性能监控数据。结合 pod 在各节点的部署拓扑信息和第一层的服务分析结果，构建涵盖应用、容器、节点三个层级的全栈现象视图。整合 pod_cpu_usage（Pod CPU 使用率）、pod_memory_working_set_bytes（Pod 工作集内存使用量）、pod_network_receive_bytes（Pod 网络接收字节数）等容器指标，以及 node_cpu_usage_rate（节点 CPU 使用率）、node_memory_usage_rate（节点内存使用率）、node_disk_read_bytes_total（磁盘读取字节数）、node_network_transmit_bytes_total（网络发送字节数）等节点指标。调用大语言模型进行第二次综合现象总结，数据对比表格同样采用 json 形式，在描述节点 kpi_key 统计指标后，描述在此节点上部署的 pods 相关 kpi_key 指标，以保持紧密的数据流，具体形式如图 15 所示，提示词设计如图 16 所示。重点分析跨层级的异常关联关系，输出从微服务应用到基础设施的完整故障现象描述。此阶段同样只进行现象总结，为后续多模态根因分析提供基础。

图 15 不同 Nodes 与在其上部署的 pods 的 kpi_key 指标统计数据对比表格示例

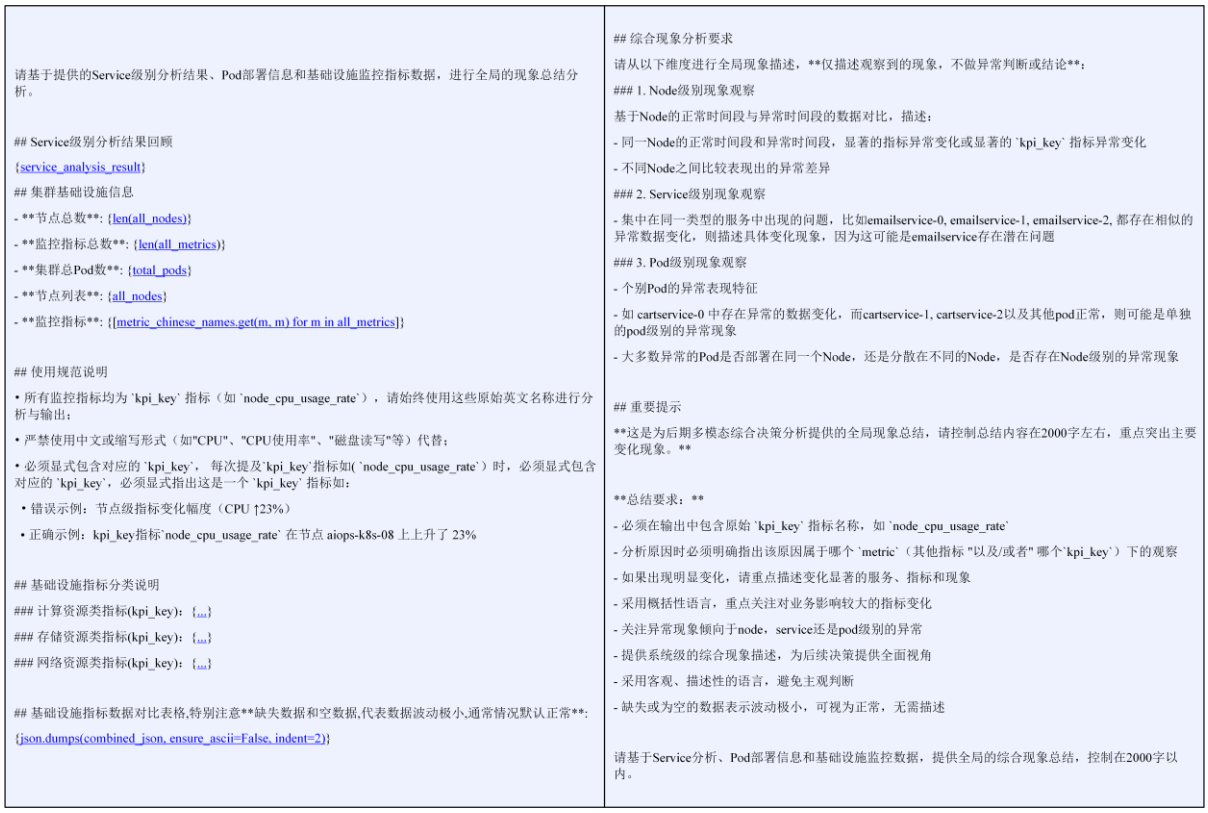

图 16 第二次综合现象总结 prompt

4.4.3 结构化现象输出与特征提取

经过双层级大模型分析处理，metric 故障总结模块输出包含以下内容的现象描述：

应用性能异常现象：通过 service-pod 的层次化信息，识别出现显著性能变化的微服务组件和数据库组件。包括 request（请求数量）、response（响应数量）、rrt（平均响应时间）等请求处理指标的变化，error_ratio（总体错误率）、client_error_ratio（客户端错误率）、server_error_ratio（服务端错误率）、timeout（超时次数）等异常指标的波动，以及 TiDB 组件 failed_query_ops（失败查询数）、duration_99th（99 分位请求延迟）、connection_count（连接数）等数据库性能指标的变化。能够同时体现服务级别的整体趋势和 Pod 级别的个体差异。

基础设施机器性能异常现象：描述容器和节点层面的资源状态变化，包括 node_cpu_usage_rate（节点 CPU 使用率）、pod_cpu_usage（Pod CPU 使用率）等计算资源指标，node_memory_usage_rate（节点内存使用率）、pod_memory_working_set_bytes（Pod 工作集内存使用量）等内存资源指标，node_disk_read_bytes_total（磁盘读取字节数）、node_disk_written_bytes_total（磁盘写入字节数）、pod_fs_reads_bytes（Pod 文件系统读取字节数）等存储资源指标，node_network_receive_bytes_total（网络接收字节数）、

node_network_transmit_bytes_total（网络发送字节数）、pod_network_receive_bytes（Pod 网络接收字节数）等网络资源指标的变化模式，以及异常分布在不同节点和容器间的空间特征。

跨层级关联现象模式：通过 Pod 部署拓扑和现象关联分析，识别服务异常与基础设施异常之间的关联模式，判断异常现象的分布特征和传播路径，为后续结合 log 和 trace 数据的多模态故障根因分析提供重要的现象线索。

通过上述系统化的处理流程，metric 故障总结模块实现了从海量多维监控数据到精炼现象描述的转换。该模块充分发挥了大语言模型的语义理解和推理能力，专注于现象总结而非故障判断，不仅显著提升了异常现象识别的准确性和完整性，还为后续结合多模态数据的故障根因分析提供了高质量的语义化 metric 数据基础。

metric 故障总结模块输出样例具体如图 17 所示:

```
### 应用性能异常现象和基础设施机器性能异常现象综合分析

#### 1. Node级别现象观察

**计算资源类指标：**
- kpi_key指标`node_cpu_usage_rate`在多个节点出现显著变化：
  - aiops-k8s-03：从正常中位数35.86%升至39.97%（+11.5%），99分位数从39.15%升至44.2%
  - aiops-k8s-05：从14.95%升至21.62%（+44.6%），99分位数从18.99%升至26.32%
  - aiops-k8s-07和aiops-k8s-08保持相对稳定（波动<±5%）

- kpi_key指标`node_memory_usage_rate`呈现差异化变化：
  - aiops-k8s-
01：从50.26%升至54.94%（+9.3%），伴随kpi_key指标`node_memory_MemAvailable_bytes`下降9.2%
  - aiops-k8s-08：从55.92%升至65.2%（+16.6%），可用内存下降22.6%
  - aiops-k8s-03和aiops-k8s-04出现内存使用率下降现象（分别-4.7%和-25.8%）

**存储资源类指标：**
- kpi_key指标`node_disk_written_bytes_total`在多个节点出现异常：
  - aiops-k8s-03：中位数从60,620.8升至145,681.07（+140%），99分位数增长32.5%
  - aiops-k8s-08：写入量保持高位（中位数>150MB），但波动范围缩小
  - aiops-k8s-02写入量异常下降50%

**网络资源类指标：**
- kpi_key指标`node_network_transmit_packets_total`在aiops-k8s-08出现基线偏移（中位数1.2→1.27）
- 其他网络指标（如`node_network_receive_bytes_total`）未显示显著变化

**节点间差异：**
- aiops-k8s-03和aiops-k8s-05同时出现CPU和磁盘写入压力
- aiops-k8s-01和aiops-k8s-08主要面临内存压力
- aiops-k8s-04和aiops-k8s-07资源指标相对稳定

#### 2. Service级别现象观察

**广告服务(adservice)：**
- adservice-0表现最显著：
  - RRT中位数增长6.5倍（3,230μs→21,092μs）
  - kpi_key指标`pod_cpu_usage`从0.01飙升至0.6（60倍增长）
  - kpi_key指标`pod_memory_working_set_bytes`增长27.1%
- adservice-1和adservice-2未显示相同程度的CPU/内存压力

**购物车服务(cartservice)：**
- cartservice-0和cartservice-2：
  - RRT 99分位数分别增长4倍和43%
  - 但kpi_key指标`pod_memory_working_set_bytes`仅增长2.2%-4.9%
  - 未观察到对应节点的CPU/内存瓶颈
- cartservice-1表现正常

**前端服务(frontend)：**
- 全实例RRT增长（12,000μs→15,800-17,200μs）
- frontend-0的kpi_key指标`pod_memory_working_set_bytes`波动增大（IQR从185,185→594,384）
- 部署在aiops-k8s-03/04/07的实例表现一致

**缓存服务(redis-cart-0)：**
- timeout次数中位数从1次增至4次
- kpi_key指标`pod_memory_working_set_bytes`异常激增（中位数124.33→42,411.29）
- kpi_key指标`pod_processes`的99分位数从1.52升至2.0

#### 3. Pod级别现象观察

**异常Pod分布特征：**
- 关键异常Pod集中在特定节点：
  - aiops-k8s-03：adservice-0、frontend-0
  - aiops-k8s-08：redis-cart-0、cartservice-2
  - aiops-k8s-07：emailservice-0（RRT增长2.5倍）

**典型异常Pod：**
- adservice-0（aiops-k8s-03）：
  - 唯一显示CPU使用率飙升的Pod（kpi_key指标`pod_cpu_usage` 0.01→0.6）
  - 内存使用与磁盘写入（kpi_key指标`pod_fs_writes_bytes`）同步增长
- redis-cart-0（aiops-k8s-08）：
  - 内存占用99分位数从3.3GB升至5.3GB
  - 进程数（kpi_key指标`pod_processes`）达到上限阈值
- emailservice-0（aiops-k8s-07）：
  - 网络吞吐量（kpi_key指标`pod_network_transmit_bytes`）增长35.3%
  - 但节点级网络指标无对应变化

**稳定服务特征：**
- paymentservice所有实例：
  - RRT波动<±10%
  - 内存使用（kpi_key指标`pod_memory_working_set_bytes`）无显著变化
- recommendationservice：
  - 所有实例指标波动在正常范围内
  - 网络流量（kpi_key指标`pod_network_receive_packets`）变化<±5%

#### 跨层级关联现象模式

1. **资源热点集中**：
   - CPU压力集中在aiops-k8s-03和aiops-k8s-05
   - 内存压力集中在aiops-k8s-01和aiops-k8s-08
   - 磁盘写入热点出现在aiops-k8s-03和aiops-k8s-08

2. **服务级联影响**：
   - 前端服务全实例延迟增长可能由下游adservice/cartservice异常放大
   - redis-cart-0的异常直接影响所有cartservice实例

3. **异常分布模式**：
   - 有状态服务（adservice/cartservice）问题更显著
   - 同一服务的不同实例表现差异大（如adservice-0 vs adservice-1）
   - 无状态服务（paymentservice等）整体稳定

4. **指标关联现象**：
   - adservice-0的CPU飙升与RRT增长直接相关
   - redis-cart-0内存增长与timeout次数增加同步出现
   - 节点级磁盘写入增长（如aiops-k8s-03）未对应到具体Pod的写入指标异常
```

图 17 metric 故障总结模块输出样例

5. 多模态根因分析

多模态根因分析模块是微服务系统故障根因定位的核心决策组件，采用基于大语言模型的综合推理策略，将经过预处理的 log 故障数据、trace 异常模式和 metric 现象

总结进行综合分析，实现从多维度异常信息到精准根因定位的自动化推理。该模块充分利用大语言模型的跨模态理解能力和逻辑推理能力，为复杂微服务故障提供最终的组件定位和原因解释。

5.1 输入数据整合

系统首先对三种模态的处理结果进行整合。Log 故障抽取模块基于 drain 日志解析算法结合多重筛选机制，将海量原始日志压缩为高质量故障特征；Trace 故障检测模块输出两部分异常信息，duration 异常检测结果包含前 20 个最频繁的 IsolationForest 识别异常，涵盖调用关系、性能对比和异常统计，status 异常检测结果包含前 20 个最频繁的状态码异常，涵盖错误类型和具体描述；Metric 故障总结模块输出经过双层级大模型分析的现象描述总结，包含约 2000 字的详细异常现象描述，涵盖 node、service、pod 多层级的监控异常模式。

5.2 多模态 prompt 生成

```
### Language Enforcement
-Input may contain Chinese, **but output MUST be entirely in English** (no Chinese characters).
请根据提供的{modalities_text}，进行综合故障分析，识别最有可能的单一故障原因。不要包含任何其他解释或文本。
特别注意**缺失数据和空数据,代表数据波动极小,通常情况默认正常,严禁分析和定位为根因**
### 微服务架构调用关系图谱
理解以下关键调用路径有助于识别故障传播和根因定位：

**主要调用路径:**
1. **用户请求入口**: User → frontend (所有用户请求的统一入口)
2. **购物核心流程**: frontend → checkoutservice → (paymentservice, emailservice, shippingservice, currencyservice)
3. **商品浏览相关**: frontend → (adservice, recommendationservice, productcatalogservice, cartservice)
4. **服务间依赖**: recommendationservice → productcatalogservice (推荐依赖商品目录)
5. **数据存储层**:
  - adservice/productcatalogservice → tidb (广告和商品数据存储)
  - cartservice → redis-cart (购物车缓存)
  - tidb 集群内部: tidb → (tidb-tidb, tidb-tikv, tidb-pd)

**故障传播分析要点:**
- **上游故障影响**: frontend故障会影响所有下游服务
- **checkout相关**: checkoutservice故障会影响支付、邮件、物流、货币服务
- **数据层故障**: tidb故障影响adservice和productcatalogservice; redis故障影响cartservice
- **横向依赖**: recommendationservice依赖productcatalogservice

要求：
1. 综合多种监控数据进行分析，优先考虑数据间的关联性，**特别关注调用路径上的故障传播模式**
2. 只返回一个最可能的故障分析结果
3. 故障级别判断标准：
  **Node级别故障**: 单个节点的监控指标(kpi_key)（node_cpu_usage_rate,node_filesystem_usage_rate等）对比正常期间,故障期间存在显著异常变化，且该节点上的多个不同服务的Pod均受影响
  **Service级别故障**: 同一服务的多个Pod实例（如emailservice-0, emailservice-1, emailservice-2）都出现相似的异常数据变化，表明服务本身存在问题
  **Pod级别故障**: 单个Pod（如cartservice-0）出现异常数据变化，而同服务的其他Pod（cartservice-1, cartservice-2）及其他Pod正常
  **重要说明**：所有监控指标均为 `kpi_key` 指标（例如 `node_cpu_usage_rate`），请在描述中直接使用这些原始 `kpi_key` 英文指标名，不得使用中文或其他名称。
```

```
4. 请确保:
  - component必须从提供的组件列表中选择，组件列表包含三种故障层级：
    * 节点名(aiops-k8s-01~08) - 表示节点级别的基础设施故障
    * 服务名(cartservice等) - 表示微服务级别的故障
    * Pod名(cartservice-0等) - 表示单个Pod级别的故障
  - reason必须说明在哪个具体的监控指标(kpi_key)发生故障,且要使用kpi_key原始英文（node_cpu_usage_rate,node_filesystem_usage_rate等）,要简明扼要，说明故障的根本原因
  - time要基于数据中的earliest_time
  - observation要基于多模态数据中的关键证据,要简明扼要,并明确指出来自哪个模态(如来自metric必须指出相应的监控指标(kpi_key),log只需要提及log(日志)检索和日志故障关键词,trace(调用链)需要按照*故障根因component*在*trace路径*出现*异常调用行为(caller/callee/self-loop)*的方式提及)
  - **重要**: 在分析trace数据时，请参考上述调用关系图谱，优先分析关键调用路径上的异常(如frontend→checkout→payment链路，或数据存储相关的服务调用)
  - **特别要求**严禁分析和定位缺失数据和空数据为根因,默认其正常
5. The JSON output must be fully in English. Any Chinese characters are strictly prohibited.
  **Strictly follow the JSON format below**:
{{
  "component": "Select from the following components: {components_list}",
  "reason": "Most likely root cause based on comprehensive multi-modal analysis; (must include kpi_key for metrics. (Do not infer from missing data.))",
  "time": "Recommended time format: 'YYYY-MM-DD HH:mm:ss', e.g. '2025-04-21 12:18:00'",
  "observation": "Key evidence from multiple data sources (log,metric,trace)",
  "reasoning_trace": [
    {{
      "step": 1,
      "action": "Such as: LoadMetrics(checkoutservice)",
      "observation": "Describe (≤20 words) the most critical anomaly in metric modality, must include exact kpi_key and change (e.g., ' node_cpu_usage_rate` increased 35% at 12:18 in metric')"
    }},
    {{
      "step": 2,
      "action": "Such as: TraceAnalysis('frontend -> checkoutservice -> paymentservice')",
      "observation": "Describe (≤20 words) the most critical abnormal behavior in trace modality, include trace path and anomaly type based on call graph (e.g., 'timeout in `checkoutservice -> paymentservice` call in trace')"
    }},
    {{
      "step": 3,
      "action": "Such as: LogSearch(checkoutservice)",
      "observation": "Describe (≤20 words) the most critical anomaly in log modality, mention error keyword and count/context (e.g., 'IOError found in 3 entries in log')"
    }}
  ]
}}

可用的监控数据:
{data_content}
```

图 18 多模态分析 prompt

基于整合后的多模态数据，系统构建专门的跨模态分析 prompt 模板。prompt 设计遵循结构化原则，明确标识各模态数据的来源、类型和分析重点，为大语言模型提供清晰的数据理解指导。同时在 prompt 中明确输出要求和格式规范，确保大模型能够基于多模态证据链进行系统化的根因推理，prompt 设计如图 18 所示。

5.3 结构化输出格式设计

系统主要通过给出输出样例提示词引导标准化的输出格式，确保根因分析结果的一致性和可解释性。输出结果包含三个核心字段：故障组件（component）、故障原因（reason）和推理过程（reasoning_trace）。故障组件字段明确标识出问题的微服务组件名称，故障原因字段提供简介明确的根因描述，推理过程字段详细记录模型的分析逻辑和证据链条，为结果验证和后续优化提供依据。

5.4 结果提取与验证机制

由于大模型输出结果可能具有不稳定性（比如最后缺少返括号，包含多余的解释等），系统采用正则表达式和结构化解析方法从大语言模型的自然语言输出中提取标准化结果。通过 JSON 格式验证确保输出结果的格式正确，通过内容完整性检查确保关键字段的有效性。对于解析失败或格式不符的结果，系统提供重试和异常处理流程，最大程度保障输出质量。

通过上述多模态根因分析流程，系统实现了从异构监控数据到故障定位的端到端自动化处理。该模块充分发挥了大语言模型在跨模态理解和逻辑推理方面的优势，为复杂微服务系统的故障根因定位提供了高效、准确、可解释的解决方案。

四、 结果分析

1. Good case 与 bad case 对比

如图 19 和图 20 所示，所提出的多模态数据融合的智能故障根因定位系统，通过深度整合日志、链路追踪和系统指标三种监控数据能，够精准识别三种模态数据的最可能的故障组件和关键根因的推导证据，这证明了设计的多模态提取方案均有效实行。此外，方案能够作出合理的顺序推理，并能有效识别故障层次类型，判断可能异常数据指标的实际层次来源（pods，service，nodes）。最终，多模态根因分析模块能够从 log 故障数据、trace 异常模式和 metric 现象三类数据的中做出合理的故障根因判断，将大预言模型的多模态分析和逻辑推理判断能力有效结合，实现合理故障根因定位的目标。

```
{
 "fault_category": "jvm fault",
 "fault_type": "jvm cpu",
 "instance_type": "pod",
 "service": "adservice",
 "instance": ["adservice-0"],
 "start_time": "2025-06-12T08:02:48Z",
 "end_time": "2025-06-12T08:25:48Z",
 "uuid": "8c1e8ce9-237",
 "key_observations": [
   {
    "type": "log",
    "keyword": ["adservice--stress"], ["deadlineexceeded"], ["invocationtargetexception"], ["transformlistener"]
   },
   {
    "type": "trace",
    "subtype": "latency_anomalies",
    "keyword": ["frontend"]
   },
   {
    "type": "metric",
    "keyword": ["pod_cpu_usage"],  ["rrt", "rrt_max"]
   }
 ],
 "key_metrics": ["pod_cpu_usage"], ["adservice--stress"],
 "fault_description": ["JVM CPU spike","JVM CPU overload","JVM high CPU usage","JVM CPU saturation"]
}
```


图 19 Good case 真实值范围


```
{
 "component": "adservice-0",
 "uuid": "8c1e8ce9-237",
 "reason": "kpi_key `pod_cpu_usage` surged and kpi_key `pod_memory_working_set_bytes` increased significantly in adservice-0",
 "reasoning_trace": [
   {
    "step": 1,
    "action": "LoadMetrics(adservice-0)",
    "observation": "kpi_key `pod_cpu_usage` surged and kpi_key `pod_memory_working_set_bytes` increased in metric"
   },
   {
    "step": 2,
    "action": "TraceAnalysis('frontend -> adservice')",
    "observation": "hipstershop.AdService/GetAds duration spiked in trace"
   },
   {
    "step": 3,
    "action": "LogSearch(adservice-0)",
    "observation": "DeadlineExceeded error found in log"
   }
 ]
}
```


图 20 Good case 实际输出值

```
{
 "fault_category": "pod fault",
 "fault_type": "pod failure",
 "instance_type": "service",
 "service": "cartservice",
 "instance": "cartservice",
 "start_time": "2025-06-05T18:10:05Z",
 "end_time": "2025-06-05T18:34:05Z",
 "uuid": "38ee3d45-82",
 "key_observations": [
  {
   "type": "log",
   "keyword": ["Connection", "refused"], ["Error", "while", "dialing"], ["failure"], ["unavailable"]
  },
  {
   "type": "metric",
    "keyword": ["client_error", "client_error_ratio", "error", "error_ratio"];["pod_cpu_usage"];
                ["pod_network_receive_bytes", "pod_network_receive_packets",
                 "pod_network_transmit_bytes", "pod_network_transmit_packets"]
  },
 ],
 "key_metrics": ["client_error", "client_error_ratio", "error", "error_ratio", "pod_processes","request","response"],
 "fault_description": ["pod crash", "pod terminated unexpectedly", "container exit", "pod failure"]
}
```


图 21 Bad case 真实值范围


```
{
 "component": "cartservice",
 "uuid": "38ee3d45-82",
 "reason": "All cartservice instances show `pod_network_receive_bytes` dropped to zero and high client error rates, indicating
complete network connectivity failure",
 "reasoning_trace": [
  {
   "step": 1,
   "action": "LoadMetrics(cartservice)",
   "observation": "`pod_network_receive_bytes` dropped to zero for all cartservice instances in metric"
  },
  {
   "step": 2,
   "action": "TraceAnalysis('frontend -> cartservice')",
   "observation": "Connection refused errors in all frontend to cartservice calls in trace"
  },
  {
   "step": 3,
   "action": "LogSearch(cartservice)",
   "observation": "Connection refused errors found in frontend logs when calling cartservice"
  }
 ]
}
```


图 22 Bad case 实际输出值

此外我们还探索了典型 bad case 以进一步研究方案的提升空间，如图 21 和图 22 所示，大模型在最终进行根因分析时出现了幻觉现象（图中蓝色所示），相关 trace 调用

链实际未传入大模型进行根因分析，从而导致错误的推导链。这可能导致在最终在故障根因判断时出现偏差，影响方案稳定性。后续或采用添加检索增强生成，优化提示工程等方式进行优化。此外 log 相关模块能够在比赛公布答案后，通过更加广泛和具有代表性的故障数据上进一步大模型语义判断等方式生成更有效的关键词过滤模板，以此增强 log 筛选能力。

五、 总结

本方案围绕微服务系统的故障根因定位构建了一个大模型智能体分析框架，整体由五大模块组成：数据预处理、日志故障抽取、调用链异常检测、指标现象总结与多模态根因分析。首先，数据预处理模块实现了对故障输入的结构化解析与时间戳标准化，为多源数据融合打下基础。日志模块基于 Drain 算法和多层级过滤机制，提取错误模板筛选有效的结构化日志特征；调用链模块结合 IsolationForest 模型和状态码检查，双重检测 duration 和 status 异常；指标模块通过筛选核心业务与基础设施指标，借助大语言模型进行两层次现象总结，生成可读性强的异常描述。最终，多模态根因分析模块将 log、trace 与 metric 输出融合，以精心设计的 prompt 驱动大语言模型进行综合推理，输出包含故障组件、原因与推理过程的结构化根因分析结果。整套系统实现了从数据处理到根因定位决策的闭环，兼顾高效性、准确性与可解释性。

六、 参考文献

[1] He P, Zhu J, Zheng Z, et al. Drain: An online log parsing approach with fixed depth tree[C]//2017 IEEE international conference on web services (ICWS). IEEE, 2017: 33-40.

[2] Liu F T, Ting K M, Zhou Z H. Isolation forest[C]//2008 eighth ieee international conference on data mining. IEEE, 2008: 413-422.