<集群性能监控系统>

软件需求规约

版本 <3.0>

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** |
| <日/月/年> | <x.x> | <详细信息> | <姓名> |
| <11/11/2024> | <1.0> | <设计，整体说明> | <吴伯涛> |
| <15/12/2024> | <2.0> | <细节补充> | <尚子森> |
| <10/01/2025> | <3.0> | <整体完善> | <尚子森> |

目录

1. 简介 4

1.1 目的 4

1.2 定义、首字母缩写词和缩略语 4

1.3 参考资料 4

2. 整体说明 4

3. 功能需求 4

3.1 <Use case 图> 5

3.2 <Use case1 规约> 5

3.3 <Use case2 规约> 5

4. 非功能需求 5

4.1 易用性 5

4.2 可靠性 5

4.3 性能 5

4.4 可支持性 6

4.5 设计约束 6

5. 其它产品需求 6

5.1 联机用户文档和联机帮助的需求 6

5.2 接口需求 6

5.2.1 用户界面 6

5.2.2 硬件接口 6

5.2.3 软件接口 6

5.2.4 通信接口 7

5.3 适用的标准 7

软件需求规约 (简化版)

# 简介

## 目的

本项目旨在提出“大模型训练监控平台的自动化搭建”，本项目的主要目的是：简化大规模分布式集群监控平台的搭建过程： 通过自动化工具和预设配置，降低搭建难度，帮助用户快速搭建一个稳定、高效的监控平台，减少学习成本，使其能够专注于数据分析而非平台构建。提升大模型训练的效率和稳定性： 实时监控GPU、CPU等资源的使用情况，帮助开发者动态调整和优化资源分配，确保训练任务顺利完成，避免资源浪费和训练中断。实现数据的持久化与可视化： 将监控数据持久化存储，方便后续分析调优，并通过可视化界面展示数据，帮助用户直观了解训练过程中的资源使用状况。

## 定义、首字母缩写词和缩略语

## 参考资料

本项目使用的技术工具和框架的开源社区和论坛：

Prometheus 社区: https://prometheus.io/community/

Grafana 社区: https://grafana.com/community/

NVIDIA 开发者论坛: https://forums.developer.nvidia.com

InfluxDB: 时序数据库的官方文档，例如 https://docs.influxdata.com/influxdb/latest/

Grafana: 监控数据可视化工具的官方文档，例如 https://grafana.com/docs/grafana/latest/

# 整体说明

1. 产品总体效果：

描述： 本部分旨在阐述该GPU和节点资源监控系统在整体上期望达成的效果。该系统主要用于对GPU和节点资源进行全面的监控、分析和管理，以确保系统的高效运行和性能优化。 - 整体上，它将为用户提供一个集中式的平台，能够实时和历史地了解GPU和节点资源的运行状态，包括但不限于GPU的温度、功耗、显存和核心利用率，以及节点的CPU、内存、磁盘和网络资源的使用情况。通过对这些资源的监控和分析，帮助用户发现潜在的性能问题、资源瓶颈和故障隐患，提高系统的可靠性和性能。 - 例如，对于一个深度学习计算集群，该系统可以帮助管理员确保GPU在训练模型时处于最佳性能状态，避免因GPU过热或资源争用而导致训练效率低下或硬件损坏。对于普通用户而言，通过直观的可视化界面，他们可以实时掌握系统的运行状态，确保自己的计算任务顺利进行。

1. 产品功能

概述： - 该系统包含多个功能模块，这些功能相互协作，形成一个完整的监控和分析系统。 数据采集功能： - 从GPU和节点收集各种关键指标数据，为后续的存储、分析和可视化提供数据基础。 - 对于GPU监控数据采集，它可以精确地获取GPU的性能指标，如温度、功耗、显存利用率和核心利用率，保证数据的准确性和实时性。 - 节点资源监控数据采集可以全面掌握节点的资源使用情况，包括CPU、内存、磁盘和网络等系统资源，让用户了解整个系统的负载状况。 - 其他数据采集功能可以根据系统的需求，可选地获取一些辅助信息，如集群规模、节点数量和资源分配，为系统管理提供更多信息。 数据存储功能： - 时序数据库存储功能将采集到的数据存储到时序数据库（如InfluxDB）中，保证数据的安全性和可靠性。 - 数据索引和查询功能允许用户方便地检索和分析数据，支持对存储的数据进行高效的索引和查询操作，方便用户进行性能分析和趋势预测。 数据可视化功能： - 仪表盘展示功能使用可视化工具（如Grafana）将监控数据以图表、仪表盘等形式展示，方便用户直观地了解系统的运行状态。 - 自定义仪表盘功能允许用户根据自身需求定制化展示关键指标和监控数据，提供个性化的查看体验。 - 实时监控功能支持对GPU和节点资源的实时监测，及时发现并处理异常情况。 告警和通知功能： - 告警规则设置功能允许用户自定义告警规则，如设置阈值告警和趋势告警，以满足不同的监控需求。 - 告警通知方式支持多种通知渠道，如邮件、短信和Slack，确保用户能够及时收到告警信息。 - 故障预警功能通过对历史数据的分析，预测潜在故障，提前发出预警，避免系统故障。 性能分析功能： - GPU和节点资源性能分析功能可找出GPU和节点资源的性能瓶颈，帮助用户优化系统性能。 - 资源使用情况分析功能分析GPU和节点资源的利用率，为资源分配和优化提供决策依据。 - 训练任务性能分析功能针对模型训练任务，分析其运行时间、训练速度和模型性能等指标，帮助用户优化训练任务。

1. 用户特征

系统管理员： - 具备系统管理和维护的专业知识，熟悉服务器、GPU硬件和网络等方面的知识。 - 他们将负责系统的配置和管理，包括设置数据采集的参数、配置数据存储和可视化工具、管理告警规则、进行性能分析等操作。 - 期望通过该系统对GPU和节点资源进行深入的监控和管理，能够根据系统的性能数据调整系统的配置和资源分配，解决系统出现的各种问题。 - 例如，系统管理员可以根据GPU的温度和功耗数据，调整GPU的工作负载或优化散热系统；根据性能分析结果，优化系统的资源分配和任务调度。 普通用户： - 可能是使用系统进行计算任务的用户，如深度学习研究者或开发人员，主要关注自己的计算任务能否顺利进行。 - 他们使用该系统的目的是查看系统的运行状态，确保自己的任务不受资源瓶颈和系统故障的影响。 - 会通过可视化界面查看监控数据，如仪表盘和自定义仪表盘，接收告警通知，以便在出现问题时能及时调整自己的任务或联系管理员。

1. 约束

技术约束： 硬件约束： - 系统依赖于GPU硬件和主机节点的性能，例如GPU的型号和性能会影响数据采集的准确性和频率。不同的GPU可能需要不同的监控工具或库进行数据采集，如NVIDIA GPU可能使用CUDA Toolkit，而AMD GPU可能需要使用AMD的监控工具。 对于节点资源监控，主机的性能和配置也会影响数据采集的效果，例如，CPU性能较低的主机可能无法保证高频率的数据采集。 软件约束： - 系统的实现依赖于多个软件工具和库，如用于GPU监控的`pynvml`库（对于NVIDIA GPU）、用于节点资源监控的`psutil`库、用于数据存储的InfluxDB和相应的客户端库、用于可视化的Grafana等。 - 这些软件工具和库的版本兼容性需要考虑，如InfluxDB的不同版本可能具有不同的特性和API，需要确保与其他组件的兼容性。 - 系统需要运行在特定的操作系统上，不同操作系统可能需要不同的安装和配置步骤，例如在Windows和Linux系统上，系统的部署和运行可能有所不同。 时间约束： - 数据采集和存储需要在一定的时间间隔内完成，以保证数据的实时性和有效性。例如，对于实时监控，数据采集的频率可能需要达到每秒一次或更高，而数据存储和处理需要在较短的时间内完成，避免数据积压和延迟。 法规约束： - 如果系统处理的数据涉及用户隐私或企业敏感信息，需要遵守相关的数据保护法规，如确保数据的存储和传输安全，防止数据泄露。

1. 假设与依赖关系

假设： - 假设GPU和主机节点在正常运行状态下能够提供准确的性能数据，即GPU和主机节点的硬件和操作系统的性能监测接口是稳定可靠的。 - 假设网络连接是稳定的，以保证数据采集、存储、可视化和通知的正常进行，例如在数据存储到时序数据库和发送告警通知时，需要稳定的网络连接。 - 假设系统的用户具备一定的操作能力和知识，如系统管理员熟悉系统管理操作，普通用户具备基本的计算机操作知识和使用可视化工具的能力。 - 依赖关系： - 该系统依赖于GPU供应商提供的监控工具和库，如NVIDIA的CUDA Toolkit及其API，以获取GPU的性能数据。 - 依赖于时序数据库（如InfluxDB）的正常运行和性能，以存储和查询监控数据。 - 依赖于可视化工具（如Grafana）的功能和性能，以实现数据的可视化展示。

1. 需求子集

数据采集需求子集： - 主要关注如何高效、准确地从GPU和节点采集数据，包括对不同类型的GPU和不同系统资源的监控数据采集，同时考虑数据采集的频率、精度和范围。 - 数据存储需求子集： - 侧重于将采集的数据安全、可靠地存储到时序数据库中，包括数据库的存储策略、数据备份和恢复机制，以及数据的索引和查询功能。 - 数据可视化需求子集： - 围绕如何通过可视化工具将监控数据以各种形式展示给用户，包括仪表盘的布局、自定义仪表盘的灵活性和实时监控的效果。 - 告警和通知需求子集： - 主要涉及告警规则的设置、多种告警通知方式的实现以及故障预警的准确性和及时性。 性能分析需求子集： - 聚焦于对GPU和节点资源的性能分析，包括找出性能瓶颈、资源利用率分析和训练任务性能分析，旨在为用户提供优化系统性能的决策依据。 通过以上对影响产品及其需求的一般因素的阐述，为后续第3节中具体需求的详述提供了清晰的背景和基础，使读者能够更好地理解各项需求的来源和目的，以及它们如何相互关联和配合，以实现整个GPU和节点资源监控系统的功能和目标。 这些内容可以根据系统的实际情况进行进一步细化和扩展，确保在开发过程中充分考虑各种因素，满足不同用户的需求和期望，同时确保系统的可靠性、性能和安全性。

# 功能需求

## <Use case 图>

1. 数据采集：

GPU 监控数据采集： 从 GPU 获取温度、功耗、显存利用率、核心利用率等关键指标数据。

节点资源监控数据采集： 从主机节点获取 CPU、内存、磁盘、网络等系统资源使用情况数据。

其他数据采集： 可选地采集其他数据，例如集群规模、节点数量、资源分配等信息。

2. 数据存储：

时序数据库存储： 将采集到的监控数据存储到时序数据库中，例如 InfluxDB，保证数据的安全性和可靠性。

数据索引和查询： 支持对存储的数据进行索引和查询，方便用户进行数据分析和趋势预测。

3. 数据可视化：

仪表盘展示： 利用可视化工具，例如 Grafana，将监控数据以图表、仪表盘等形式展示，方便用户直观了解 GPU 和节点资源的运行状态。

自定义仪表盘： 用户可以根据自己的需求，自定义创建仪表盘，展示关键指标和监控数据。

实时监控： 支持实时监控 GPU 和节点资源的运行状态，及时发现并处理异常情况。

4. 告警和通知：

告警规则设置： 用户可以自定义设置告警规则，例如阈值告警、趋势告警等。

告警通知方式： 支持多种告警通知方式，例如邮件、短信、Slack 等。

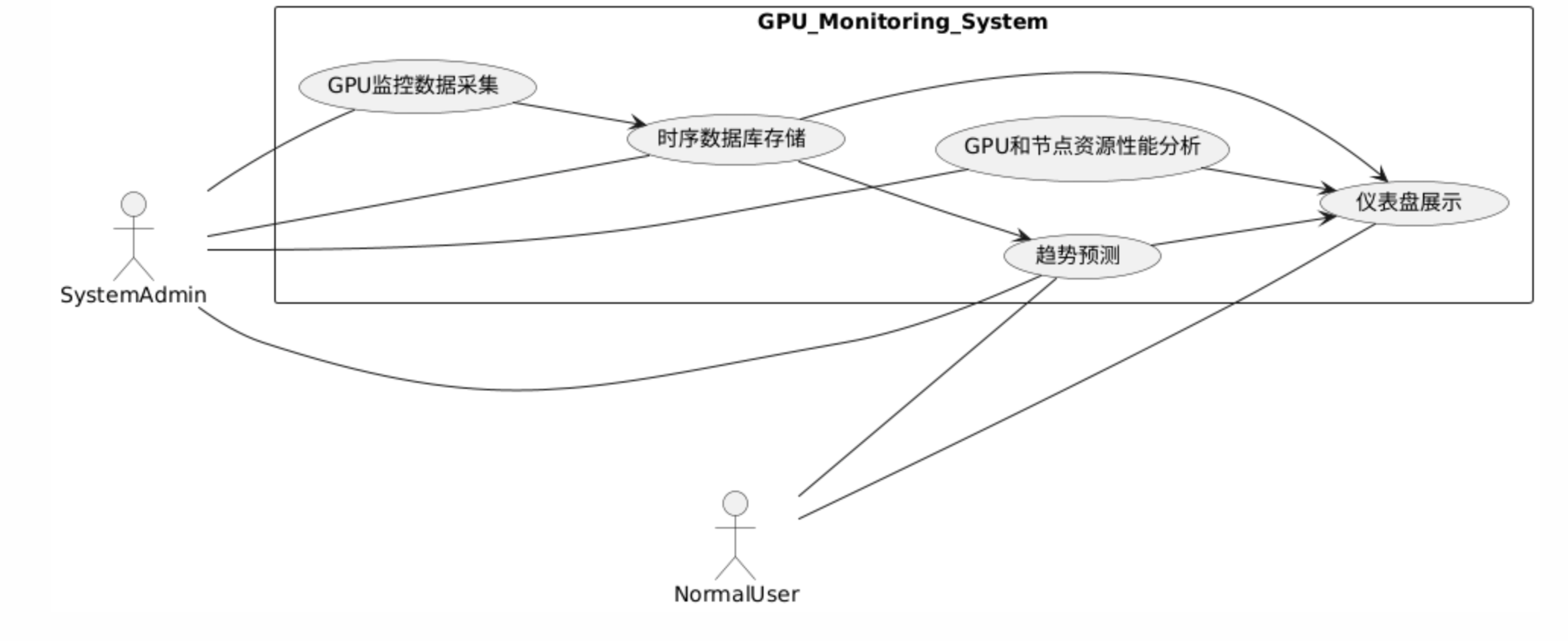
故障预警： 通过分析历史数据，预测潜在故障，提前发出预警，避免系统故障。

5. 性能分析：

GPU 和节点资源性能分析： 分析 GPU 和节点资源的性能瓶颈，例如 CPU 瓶颈、内存瓶颈等。

资源使用情况分析： 分析 GPU 和节点资源的利用率，例如 CPU 利用率、内存利用率等。

训练任务性能分析： 分析模型训练任务的运行时间、训练速度、模型性能等指标。

说明：

系统边界：矩形框GPU\_and\_Node\_Resource\_Monitoring\_System表示该 GPU 和节点资源监控系统的整体范围，包含了从数据采集、存储、可视化到性能分析和告警的各个功能模块。

参与者关联：

系统管理员（SA）参与了系统的核心操作，包括数据采集的配置、存储和查询功能的管理、性能分析、告警规则的设置等多个方面，以确保系统的正常运行和性能优化。

普通用户（NU）主要通过可视化界面查看数据、接收告警通知，并可以自定义仪表盘以满足自身需求。

用例关系：

数据采集的几个用例（GDC、NDC、ODC）都将数据存储到时序数据库存储（TDS）中。

时序数据库存储（TDS）为数据索引和查询（DIQ）提供数据，数据索引和查询（DIQ）又为各种分析（GPA、RUA、TPA）和展示（DS、CDS）提供支持。

实时监控（RM）依赖于仪表盘展示（DS）来呈现数据，而故障预警（FW）会触发告警通知方式（ANM）向用户发送告警信息。

此 Use Case 图展示了 GPU 和节点资源监控系统的主要功能及其参与者之间的关系，通过这种结构化的表示，可以清晰地看到系统的功能划分和不同角色的操作范围，为系统的开发、维护和使用提供了清晰的指导。

## <Use case1 规约>

· GPU 监控数据采集

· 前置条件：

GPU 硬件正常运行且可访问。

系统中已安装相应的 GPU 监控工具或库，如 NVIDIA 的 CUDA Toolkit 及其相关 API 库（如 pynvml）。

基本事件流：

系统管理员启动 GPU 监控数据采集功能。

系统调用 GPU 监控工具或库（如 pynvml）的 API 函数。

通过 API 函数（如 pynvml.nvmlDeviceGetHandleByIndex）获取 GPU 的设备句柄。

使用其他 API 函数（如 pynvml.nvmlDeviceGetTemperature、pynvml.nvmlDeviceGetPowerUsage、pynvml.nvmlDeviceGetMemoryInfo、pynvml.nvmlDeviceGetUtilizationRates）分别采集 GPU 的温度、功耗、显存利用率和核心利用率等关键指标数据。

将采集到的数据进行必要的格式转换和整理，使其符合后续存储和处理的要求。

将采集到的数据传递给数据存储模块进行存储。

异常事件流：

如果 GPU 硬件出现故障或无法访问，系统将记录错误信息，并向系统管理员发送通知，提示 GPU 硬件故障。

如果监控工具或库的 API 调用失败，将记录错误信息，并尝试重新调用或通知系统管理员进行维护。

后置条件：

成功采集到 GPU 的关键指标数据并传递给数据存储模块。

· 节点资源监控数据采集

· 前置条件：

主机节点正常运行，操作系统和网络连接正常。

已安装系统监控工具，如 psutil 库（对于 Python 实现）。

基本事件流：

系统管理员启动节点资源监控数据采集功能。

系统调用系统监控工具（如 psutil）的相应函数。

使用 psutil.cpu\_percent 函数获取 CPU 使用率，使用 psutil.virtual\_memory 函数获取内存信息，使用 psutil.disk\_usage 函数获取磁盘使用情况，使用 psutil.net\_io\_counters 函数获取网络 I/O 数据。

对采集到的数据进行整理和封装，以便存储和后续处理。

将采集到的数据传递给数据存储模块进行存储。

异常事件流：

如果主机节点出现故障，如 CPU 过载、内存溢出或网络中断，系统将记录错误信息，并向系统管理员发送通知。

如果监控工具调用失败，系统将记录错误信息并尝试重新调用或通知系统管理员进行维护。

后置条件：

成功采集到主机节点的系统资源使用情况数据并传递给数据存储模块。

· 其他数据采集

· 前置条件：

系统处于运行状态，并且存在可获取所需信息的数据源或配置文件。

基本事件流：

系统管理员启动其他数据采集功能。

系统根据预先设定的采集逻辑，从相应的数据源（如配置文件、系统命令、集群管理工具）获取数据。

对于集群规模，可能使用集群管理工具（如 kubectl 或自定义的集群管理工具）获取集群中的节点数量等信息。

对于资源分配信息，从配置文件或容器编排系统的 API 中读取资源分配情况。

将采集到的数据进行整理和封装，以便存储和后续处理。

将采集到的数据传递给数据存储模块进行存储。

异常事件流：

如果数据源不可用或读取失败，系统将记录错误信息，并通知系统管理员。

如果配置文件丢失或损坏，系统将记录错误信息并通知系统管理员进行修复。

后置条件：

成功采集到所需的其他数据并传递给数据存储模块。

## <Use case2 规约>

数据存储

时序数据库存储

前置条件：

时序数据库（如 InfluxDB）已安装、配置并正常运行。

系统与时序数据库建立了网络连接。

基本事件流：

系统接收来自数据采集模块的数据。

使用数据库客户端（如 InfluxDB 的 influxdb-client）连接到时序数据库。

将数据按照预先设定的格式和存储策略存储到时序数据库中。例如，使用 InfluxDBClient 的 write\_api 函数将数据写入 InfluxDB 的指定 bucket 中。

存储时，为数据添加时间戳和标签（如数据源类型、数据类别等），以便后续查询和分析。

异常事件流：

如果数据库连接失败，系统将尝试重新连接，并记录错误信息，同时通知系统管理员。

如果数据存储操作失败，将记录错误信息并尝试重新存储，若多次失败将通知系统管理员。

后置条件：

数据成功存储到时序数据库中，确保数据的安全性和可靠性。

数据索引和查询

前置条件：

时序数据库中已存储有足够的数据。

已建立正确的数据存储结构和索引机制。

基本事件流：

用户或系统管理员发起数据查询请求，指定查询条件（如时间范围、数据指标等）。

系统根据请求构建查询语句，使用相应的数据库查询语言（如 InfluxQL 或 Flux）。

系统将查询语句发送到时序数据库。

时序数据库执行查询并返回结果。

系统对返回的数据进行处理和解析，将结果呈现给用户或传递给分析模块。

异常事件流：

如果查询语句构建错误，系统将记录错误信息并通知用户或系统管理员。

如果数据库查询失败，将尝试重新查询，若多次失败将通知用户或系统管理员。

后置条件：

成功查询并返回满足条件的数据。

# 非功能需求

## 易用性

* 自动化搭建： 通过自动化工具和预设配置，降低搭建难度，用户无需深入了解复杂的监控技术，即可快速搭建完整的监控平台。
* 可视化界面： 利用 Grafana 等可视化工具，将监控数据以图表、仪表盘等形式展示，方便用户直观了解 GPU 和节点资源的运行状态。
* 自定义仪表盘： 用户可以根据自己的需求，自定义创建仪表盘，展示关键指标和监控数据。
* 告警通知： 支持多种告警通知方式，例如邮件、短信、Slack 等，方便用户及时了解系统状态。
* 用户友好的操作界面： 项目将提供用户友好的操作界面，方便用户进行系统配置、数据查询、告警设置等操作。

## 可靠性

* 开源工具的稳定性： 项目依赖于 Prometheus、Grafana、InfluxDB 等成熟的开源工具，这些工具经过了广泛的测试和验证，具有较高的稳定性。
* 数据备份和恢复： 项目支持数据备份和恢复功能，保证监控数据的安全性。
* 高可用性设计： 项目将采用高可用性设计，例如使用负载均衡、故障转移等技术，确保系统在出现故障时能够快速恢复。
* 故障预警： 通过分析历史数据，预测潜在故障，提前发出预警，避免系统故障。
* 系统监控： 项目将监控自身的运行状态，例如系统负载、内存使用情况等，及时发现并解决潜在问题。

## 性能

* 高效的数据采集： 项目采用高效的采集方式，例如 DCGM 和 Node Exporter，能够快速采集 GPU 和节点资源监控数据。
* 时序数据库： 项目采用 InfluxDB 等时序数据库存储监控数据，支持高效的写入和查询操作，保证数据的实时性和准确性。
* 分布式架构： 项目支持分布式部署，能够处理大规模 GPU 集群的监控数据。
* 性能优化： 项目将进行性能优化，例如数据压缩、缓存机制等，提高系统的运行效率。

## 可支持性

## 编码标准

## 编程语言选择与版本： -明确规定系统开发所使用的编程语言及相应的版本。例如，如果系统使用 Python 开发，应指定 Python 的具体版本，如 Python 3.8 及以上。这有助于确保开发环境的一致性，避免因语言版本差异导致的兼容性问题。 - 对于不同模块或组件，如果涉及多种语言，分别说明其使用的语言和版本，如部分性能关键的模块使用 C++ 开发，指定 C++ 版本为 C++17 或更高。 代码风格规范： - 遵循统一的代码风格，如使用 PEP 8 标准（对于 Python 开发）或 Google 的 C++ 代码风格指南（对于 C++ 开发）。这包括缩进、变量命名、函数和类的命名、注释等方面的规范。 - 要求代码具有良好的可读性，如代码行长度不超过一定限制（例如 Python 中建议不超过 79 个字符），函数长度适中，避免过长函数，提高代码的可理解性。 - 对于代码中的注释，要求在关键部分添加清晰、详细的注释，解释代码的功能、输入、输出和重要逻辑，尤其是复杂的算法或数据结构部分。例如，在函数开头使用文档字符串说明函数的功能和参数，对于重要的逻辑块添加行内注释解释代码的实现细节。

## 命名约定

## 变量和函数命名： - 采用有意义且描述性的命名方式。变量名应清晰地表示其存储的数据，例如 `gpu\_temperature` 表示 GPU 的温度，`cpu\_usage\_percentage` 表示 CPU 的使用百分比。 - 函数名应使用动词开头，清晰地描述其功能，如 `get\_gpu\_power\_usage()` 表示获取 GPU 的功耗，`store\_data\_in\_influxdb()` 表示将数据存储到时序数据库。 - 对于常量，使用大写字母加下划线的方式命名，如 `MAX\_GPU\_TEMPERATURE\_THRESHOLD` 表示 GPU 温度的最大阈值。 类和模块命名： - 类名使用首字母大写的驼峰命名法，如 `GPUDataCollector` 表示 GPU 数据采集类，`NodeResourceMonitor` 表示节点资源监控类。 - 模块名使用小写字母加下划线的方式命名，如 `gpu\_monitoring.py` 表示 GPU 监控的 Python 模块，`node\_resource\_monitor.cpp` 表示节点资源监控的 C++ 模块。

## 类库

## 使用标准库优先： - 在满足系统需求的前提下，优先使用编程语言的标准库，以减少外部依赖。例如，对于 Python 开发，优先使用 `os`、`sys`、`time` 等标准库进行系统级操作和时间处理；对于 C++，优先使用 STL 提供的容器、算法和工具类。 - 只有当标准库无法满足需求时，才考虑使用第三方类库，并在引入时进行严格评估。 第三方类库管理： - 对于使用的第三方类库，明确列出其名称、版本和来源。如使用 `pynvml` 库进行 NVIDIA GPU 监控，指定版本为 11.4.1，并从 PyPI 下载；使用 `psutil` 进行节点资源监控，指定版本为 5.8.0，也从 PyPI 下载。 - 对第三方类库的更新和维护要有明确的策略，定期检查其更新，评估更新带来的影响，确保更新不会破坏系统的功能和性能。例如，在更新第三方库时，先在测试环境中测试，确保与系统的兼容性。

## 维护访问权

## 系统维护人员权限： 系统管理员应具有对系统的完全访问权，包括对数据采集模块、存储模块、可视化工具和性能分析工具的配置和修改权限。例如，能够修改数据采集的频率、更改告警规则、调整可视化界面的布局等。 对于系统的代码库，只有经过授权的开发人员和维护人员可以访问和修改。使用版本控制系统（如 Git）管理代码库，设置不同的权限级别，如管理员可以合并代码、创建分支和修改主分支，普通开发人员仅能提交代码到开发分支。 数据访问权限： 对存储的数据，区分不同用户的访问权限。系统管理员可以访问和修改所有数据，普通用户只能查看和部分修改自己的数据（如自定义仪表盘）。 对于存储在时序数据库中的数据，通过数据库的用户权限管理机制实现权限控制，确保数据的安全性和隐私性。例如，在 InfluxDB 中设置不同用户角色和权限，仅允许管理员执行数据写入和删除操作，普通用户仅可查询数据。

## 维护实用程序

## 日志记录： - 系统应具备日志记录功能，记录系统的运行状态、异常情况和重要操作。使用日志记录库，如 Python 中的 `logging` 模块或 C++ 中的 `spdlog` 库。 - 日志应包括不同的日志级别（如 INFO、WARNING、ERROR），根据操作的重要性和异常程度进行记录。例如，记录数据采集的开始和结束时间、存储操作的成功或失败信息、告警的触发和通知情况。 系统备份和恢复： - 对于存储在时序数据库中的数据，定期进行备份。可以使用 InfluxDB 的备份工具，设定备份的时间间隔（如每天或每周备份）和备份的存储位置（如外部存储设备或云存储）。 - 开发系统恢复程序，在系统故障或数据丢失时，能够根据备份数据快速恢复系统的运行状态。例如，使用脚本将备份数据从存储位置恢复到时序数据库中。 性能监控和优化工具： - 集成性能监控工具，如使用 `cProfile`（对于 Python）或 `perf`（对于 C++）监控系统性能，帮助找出性能瓶颈。 - 提供性能优化工具或脚本来优化系统性能，例如在 Python 中使用 `memory\_profiler` 检查内存使用情况，在 C++ 中使用 `valgrind` 检查内存泄漏和性能问题。

## 版本控制和文档管理

## 版本控制： - 使用版本控制系统（如 Git）管理系统的代码，为每个版本创建相应的标签，如 `v1.0`、`v1.1` 等，方便回滚和追踪代码的变更历史。 - 建立分支管理策略，如 `main` 分支用于稳定版本，`develop` 分支用于开发，不同的功能开发可以创建单独的功能分支，如 `feature/gpu\_monitoring\_optimization`。 文档更新： - 随着系统的开发和维护，及时更新文档，包括软件需求规约、设计文档、用户手册和 API 文档。 - 确保文档与系统的最新版本同步，如在更新系统功能后，及时修改软件需求规约中相应的部分，更新用户手册中的操作步骤，更新 API 文档中的接口信息。 通过明确这些可支持性和可维护性需求，可以确保系统在开发和后续的维护过程中保持良好的结构和性能，便于开发人员和维护人员进行操作和管理，提高系统的生命周期和用户满意度。 在实际开发和维护过程中，这些需求可以根据系统的具体情况进行细化和调整，以满足不同阶段和不同环境的需求。同时，应建立相应的检查和评估机制，确保这些需求得到有效执行和遵循。

## 设计约束

# 软件语言

# 编程语言选择： 系统的主要开发语言为 Python，部分性能关键的模块可以考虑使用 C 或 C++ 语言进行优化。 - Python 用于实现系统的核心功能，因为其具有丰富的库和简洁的语法，方便开发和维护。例如，使用 Python 中的 `pynvml` 库来实现 GPU 监控数据采集，使用 `psutil` 库来实现节点资源监控数据采集，使用 `influxdb-client` 进行时序数据库操作，使用 `flask` 或 `django` 开发系统的 Web 界面（如果有）等。 - C 或 C++ 可用于实现对性能要求极高的部分，例如某些算法的核心计算部分，以提高系统的运行效率。

# 软件流程需求

# 开发流程： - 采用敏捷开发流程，将系统开发分为多个迭代周期。每个迭代周期包括需求分析、设计、编码、测试和部署阶段。 - 定期召开敏捷会议，如每日站会、迭代回顾会议和迭代计划会议，以确保项目的进度和质量。例如，每日站会让团队成员分享进展和问题，迭代回顾会议总结经验教训，迭代计划会议规划下一个迭代的任务。 - 进行持续集成和持续部署（CI/CD），使用工具如 Jenkins、GitLab CI 或 GitHub Actions 对代码进行自动化构建、测试和部署。例如，开发人员提交代码后，CI 工具自动运行测试用例，确保代码质量，通过测试的代码可自动部署到测试环境或生产环境。 测试流程： - 遵循单元测试、集成测试和系统测试的流程。 - 对于单元测试，使用 Python 的 `unittest` 或 `pytest` 框架，对每个函数和类进行单元测试，确保其功能的正确性。例如，对 `get\_gpu\_data()` 函数进行单元测试，检查其是否能正确获取 GPU 的各项指标。 - 集成测试将多个组件组合在一起进行测试，检查它们之间的协作是否正常。例如，测试数据采集模块与数据存储模块的集成，确保采集到的数据能正确存储到时序数据库。 - 系统测试对整个系统进行测试，模拟真实环境进行功能和性能测试，确保系统满足用户的需求。

# 开发工具的指定用途

# 代码编辑器和 IDE： - 推荐使用 Visual Studio Code 或 PyCharm 作为主要的代码编辑器和集成开发环境（IDE）。 - Visual Studio Code 可通过安装相关插件（如 Python 插件、C/C++ 插件）进行不同语言的开发和调试。PyCharm 则专门针对 Python 开发，提供强大的代码自动补全、调试和测试功能。 - 开发人员可以根据自己的喜好选择合适的工具，但要确保代码的一致性和质量。 版本控制工具： - 使用 Git 作为版本控制工具，管理代码的版本和分支。 - 开发人员使用 Git 客户端（如 Git Bash、SourceTree 或 GitKraken）进行代码的提交、拉取和合并操作。 - 利用 GitHub 或 GitLab 作为远程代码仓库，存储代码并进行团队协作，例如通过 Pull Request 机制进行代码审查和合并。

# 构架及设计约束

# 系统架构： - 采用分层架构，将系统分为数据采集层、数据存储层、数据处理层、数据可视化层和用户界面层。 - 数据采集层负责从 GPU 和节点收集数据；数据存储层将数据存储到时序数据库；数据处理层对数据进行分析和处理；数据可视化层将数据以可视化形式展示；用户界面层提供用户操作的界面。 - 各层之间通过定义清晰的接口进行通信，例如数据采集层将采集到的数据传递给数据存储层，通过调用数据存储层的接口将数据存储到时序数据库。 分布式架构（如果适用）： - 如果系统是分布式的，如在集群环境中，考虑使用分布式架构模式，如微服务架构或主从架构。 - 对于微服务架构，将不同的功能模块作为独立的服务部署，通过服务间的通信机制（如 RESTful API 或消息队列）进行交互。例如，将 GPU 监控数据采集作为一个微服务，将节点资源监控数据采集作为另一个微服务。

# 购买的构件

# 商业软件和服务： - 购买 InfluxDB 作为时序数据库，使用其企业版或社区版，根据系统的规模和性能要求选择合适的版本。 - 如果使用 Grafana 进行数据可视化，可购买 Grafana 的专业版或使用其开源版，根据需要购买 Grafana 的插件和扩展功能。

# 类库

# 必须使用的类库： - 对于 Python 开发，必须使用 `pynvml` 库进行 GPU 监控，使用 `psutil` 库进行节点资源监控，使用 `influxdb-client` 进行时序数据库操作，使用 `requests` 库进行 HTTP 请求（如果需要）等。 - 在使用类库时，确保使用最新的稳定版本，并定期更新类库，同时考虑兼容性问题。例如，使用 `pynvml` 库时，遵循其官方文档进行 API 调用，注意版本更新带来的 API 变化。 类库使用限制： - 避免使用过多的第三方类库，除非必要，以减少系统的复杂性和潜在的兼容性问题。 - 在引入新的类库之前，进行充分的评估和测试，确保其不会对系统的性能、稳定性和安全性产生负面影响。

# 数据库约束

# 数据库选择： 选定 InfluxDB 作为时序数据库，因为其适合存储时间序列数据，并且具有良好的性能和扩展性。 数据库的部署可以在本地服务器或云端，根据系统的性能和成本要求进行选择。例如，对于小规模系统，可以部署在本地服务器；对于大规模系统，可以使用云服务提供商的 InfluxDB 服务。 数据库性能和存储要求： 数据库应能满足系统对数据存储和查询的性能要求，如支持高并发的数据写入和快速的数据查询。 设定数据库的存储容量，根据系统的数据增长预测，考虑存储容量的扩展，例如使用分布式存储机制或扩展存储硬件。

# 用户界面设计约束

# 界面设计原则： 遵循用户界面设计的最佳实践，确保界面简洁、直观和易用。例如，在数据可视化界面，使用户能够方便地查看不同指标的仪表盘和图表，方便用户操作和自定义。 界面设计应符合用户的使用习惯，对于不同类型的用户（系统管理员和普通用户），提供不同的操作界面和权限。 响应时间要求： 对于用户界面的操作，如仪表盘的刷新、数据查询和自定义操作，响应时间应在可接受范围内，如在普通网络条件下，页面响应时间不超过 3 秒。 通过明确这些设计约束，开发团队可以确保在开发过程中遵循既定的决策，避免不必要的变更和风险，提高系统的开发效率和质量，保证系统最终满足用户的需求和期望。在实际开发中，可根据具体情况对这些约束进行细化和调整，以更好地指导系统的开发和设计。

# 其它产品需求

## 联机用户文档和联机帮助的需求

Prometheus 社区: https://prometheus.io/community/

Grafana 社区: https://grafana.com/community/

NVIDIA 开发者论坛: https://forums.developer.nvidia.com

InfluxDB: 时序数据库的官方文档，例如 https://docs.influxdata.com/influxdb/latest/

Grafana: 监控数据可视化工具的官方文档，例如 https://grafana.com/docs/grafana/latest/

## 接口需求

### 用户界面

**概述**：

该系统的用户界面主要为用户提供操作和查看系统信息的入口，使用户能够方便地监控 GPU 和节点资源的状态，设置告警规则，查看分析结果等。

**登录界面**：

**协议**：使用 HTTP/HTTPS 协议进行通信。

* **端口**：通常在开发阶段使用 8080 端口（HTTP），部署阶段使用 443 端口（HTTPS）。
* **逻辑地址**：/login
* **功能描述**：
  + 用户输入用户名和密码，系统将通过后端服务对输入信息进行验证。
  + 界面应包含输入框、密码框和提交按钮，支持用户输入用户名和密码，并在用户点击提交按钮时向服务器发送请求。
  + 服务器端将根据用户输入进行身份验证，验证成功后将用户重定向到相应的操作界面（如仪表盘界面），验证失败将显示错误信息。

**仪表盘界面**：

* **协议**：HTTP/HTTPS
* **端口**：同登录界面。
* **逻辑地址**：/dashboard
* **功能描述**：
  + 展示 GPU 和节点资源的实时监控数据，包括 GPU 的温度、功耗、显存利用率、核心利用率，以及节点的 CPU 使用率、内存使用情况、磁盘使用量、网络 I/O 数据等。
  + 可使用多种可视化元素，如折线图、柱状图、表格等，以直观的方式呈现数据。
  + 支持用户进行操作，如切换不同时间范围的数据查看，刷新数据，以及点击进入详细信息查看或分析界面。

**告警设置界面**：

* **协议**：HTTP/HTTPS
* **端口**：同登录界面。
* **逻辑地址**：/alarm-settings
* **功能描述**：
  + 为系统管理员提供设置告警规则的界面。
  + 包含各种输入框和下拉菜单，用于设置不同类型的告警，如阈值告警、趋势告警等。
  + 提供选项让用户选择告警通知方式（如邮件、短信、Slack），并可输入相应的接收信息（如邮件地址、手机号码、Slack 频道等）。

### 硬件接口

**GPU 接口**：

* **逻辑结构**：
  + 对于 NVIDIA GPU，通过 NVIDIA 的 NVML（NVIDIA Management Library）与 GPU 硬件进行通信。
  + 对于 AMD GPU，使用 AMD 的相应管理库（如 ADL 或 Atiadlxx）进行通信。
* **物理地址**：
  + 硬件设备的物理地址由操作系统和 PCIe 总线分配，系统通过操作系统提供的设备管理功能来识别和定位 GPU。
* **预期行为**：
  + 能够准确读取 GPU 的各种性能指标，如温度、功耗、显存使用情况、核心频率等。
  + 可通过调用相应库的 API 实现对 GPU 状态的监控和部分性能参数的调整

**节点硬件接口**：

* **逻辑结构**：
  + 主要通过操作系统的系统调用和系统监控工具库（如 psutil）来访问节点的硬件资源。
* **物理地址**：
  + 系统将自动识别和访问 CPU、内存、磁盘和网络接口卡等硬件，这些硬件的物理地址由主板和操作系统分配。
* **预期行为**：
  + 对于 CPU，能够获取使用率、核心数、频率等信息。
  + 对于内存，可获取总内存、可用内存、已使用内存等信息。
  + 对于磁盘，可获取磁盘的总容量、已使用容量、I/O 速度等。
  + 对于网络接口卡，可获取网络流量、带宽使用情况等。

### 软件接口

**数据采集与存储接口**：

* **协议**：
  + 对于数据采集模块和存储模块之间的数据传递，使用自定义的内部数据结构和方法调用，或消息队列（如 RabbitMQ、Apache Kafka）进行解耦。
* **逻辑地址**：
  + 对于内部方法调用，数据采集模块将调用存储模块的相应函数（如 store\_data(data)）将采集到的数据传递给存储模块。
  + 对于消息队列，可使用队列名称，如 monitoring\_data\_queue 作为逻辑地址。
* **功能描述**：
  + 数据采集模块将采集到的 GPU 和节点资源数据封装成特定的数据结构（如 JSON 格式），然后传递给存储模块。
  + 当使用消息队列时，数据采集模块将数据作为消息发送到队列，存储模块从队列中消费消息。

**存储与可视化接口**：

* + **协议**：
    - 存储模块和可视化模块通过数据库查询语言（如 InfluxQL 或 Flux）进行交互，存储模块作为数据提供方，可视化模块作为数据使用方。
  + **逻辑地址**：
    - 在使用 InfluxDB 作为存储时，可视化模块使用 InfluxDB 的查询语句从存储中获取数据。例如，SELECT \* FROM monitoring\_data WHERE time > now() - 1h 从 monitoring\_data 表中查询过去一小时的数据。
  + **功能描述**：
    - 可视化模块根据用户的操作和需求，构建相应的查询语句从存储模块中查询数据。
    - 存储模块根据查询语句返回相应的数据，数据格式可以是 JSON 或其他数据格式，以便可视化模块进行处理和展示。

### 通信接口

**局域网接口**：

* + **协议**：
    - 使用 TCP/IP 协议族，包括 TCP 和 UDP 协议。
  + **端口**：
    - 对于系统内部服务之间的通信（如不同微服务之间），可使用自定义端口，如 5000 - 6000 范围的端口。
    - 对于外部服务访问，使用标准的 HTTP/HTTPS 端口（如 80/443）。
  + **逻辑地址**：
    - 内部服务的逻辑地址根据服务的功能命名，如 /data-collection-service、/data-storage-service 等。
  + **功能描述**：
    - 系统内的不同服务之间通过 TCP 或 UDP 协议进行数据交换，确保数据的可靠或快速传输。
    - 对于外部服务，通过 HTTP/HTTPS 提供 Web 服务，允许用户通过网络访问系统。

## 适用的标准

**一、法律标准**

* **数据保护和隐私法规**：
  + **适用法规**：
    - 遵循通用数据保护条例（GDPR）：适用于处理欧盟公民个人数据的系统，确保系统在数据存储、处理和传输过程中保护用户的隐私。
    - 加州消费者隐私法案（CCPA）：如果系统涉及处理加州居民的数据，需遵守该法案的相关要求，如向用户提供数据访问、删除和选择退出的权利。
  + **具体要求**：
    - 在收集和存储用户数据时，获得用户的明确同意，特别是涉及个人信息（如用户登录信息）的数据。
    - 对存储的数据进行加密，防止数据泄露和未经授权的访问。例如，使用 SSL/TLS 协议对传输的数据进行加密，使用 AES 或 RSA 等加密算法对存储的数据进行加密。
    - 当用户请求删除或访问其数据时，系统应提供相应的操作功能，并在规定时间内完成。

1. **质量标准**

**ISO 9001 质量管理体系**：

* + **适用部分**：
    - 涵盖系统开发、测试和维护的全过程，确保系统的质量满足用户需求。
  + **具体要求**：
    - 建立质量管理体系，对项目的各个阶段进行质量控制，包括需求分析、设计、开发、测试和部署。
    - 记录和跟踪质量问题，使用质量工具（如缺陷跟踪系统）对开发过程中发现的问题进行记录和解决。
    - 进行定期的质量审核，确保开发过程符合 ISO 9001 标准，持续改进系统的质量。

**三、业界标准**

**易用性标准**：

* + **Web 内容可访问性指南（WCAG）**：
    - **适用部分**：
      * 对于系统的用户界面，尤其是 Web 界面部分，遵循 WCAG 2.1 或更高版本的标准。
    - **具体要求**：
      * 确保界面元素具有足够的对比度，方便用户阅读和操作。例如，文本和背景颜色的对比度应符合 WCAG 的 AA 级或 AAA 级标准。
      * 为界面元素提供适当的文本描述和标签，方便屏幕阅读器等辅助技术使用，使系统可供残疾用户使用。
      * 确保界面操作可通过键盘完成，方便不使用鼠标的用户操作。

**互操作性标准**：

* + **开放系统互连基本参考模型（OSI）**：
    - **适用部分**：
      * 系统在网络通信方面遵循 OSI 参考模型，确保不同系统之间的网络通信顺畅。
    - **具体要求**：
      * 系统的网络协议栈遵循 TCP/IP 协议族，确保不同设备和系统之间的数据传输符合 OSI 模型的分层结构。
      * 对于系统间的接口，如与外部系统的通信接口，确保遵循标准的 HTTP/HTTPS 协议，便于与其他系统的集成和互操作。

**国际化标准**：

* + **Unicode 标准**：
    - **适用部分**：
      * 在处理文本数据时，使用 Unicode 编码（如 UTF-8），确保系统支持多种语言。
    - **具体要求**：
      * 系统的用户界面、数据存储和处理都使用 Unicode 编码，避免出现乱码问题。
      * 支持不同语言的显示和输入，满足不同国家和地区用户的需求，例如在输入和显示中文、英文、日文等多种语言时都能正确显示。

**操作系统相容性标准**：

**POSIX 标准（对于 Unix/Linux 系统）**：

* + - **适用部分**：
      * 对于在 Unix/Linux 平台上开发的系统组件，遵循 POSIX 标准。
    - **具体要求**：
      * 系统的系统调用、文件操作、进程管理等遵循 POSIX 标准，确保在不同 Unix/Linux 发行版（如 Ubuntu、CentOS）上的可移植性。
      * 使用 POSIX 标准的线程和进程管理函数，避免使用平台特定的函数，提高代码的可移植性。

**Windows API 标准（对于 Windows 系统）**：

* + - **适用部分**：
      * 在 Windows 平台上开发的系统组件，遵循 Windows API 标准。
    - **具体要求**：
      * 使用 Windows 标准的系统调用和库函数，如 CreateProcess 用于进程创建，ReadFile 和 WriteFile 用于文件操作，确保系统在 Windows 系统上的兼容性。

**四、行业特定标准**

* **深度学习框架标准（如果系统服务于深度学习领域）**：
  + **ONNX 标准（开放神经网络交换格式）**：
    - **适用部分**：
      * 当系统涉及深度学习模型的处理和交换时，遵循 ONNX 标准。
    - **具体要求**：
      * 支持将不同深度学习框架（如 TensorFlow、PyTorch）的模型转换为 ONNX 格式，以便在不同平台和框架之间进行模型的移植和共享。
      * 能够正确导入和导出 ONNX 格式的模型，对模型的输入和输出进行正确的解析和处理。