基于容器化 Kubernetes 平台规范对接指南

版本	时间	作者
Beta	2024/11/28	SL-SA Kerbos
Beta v2	2024/11/29	SL-SA Kerbos

以下内容如有不足请及时指出

概述

本规范旨在为外部工作室提供一套关于 Kubernetes 容器化平台的对接指南与规范要求，确保外部工作室能够高效、统一地与我方运维团队协作。规范涵盖了容器化镜像构建、CI/CD 流程、应用编排、以及集群资源配置、部署流程等方面，确保外部工作室在对接过程中遵循统一标准，减少对接难度，提升协作效率。

规范目的

- **提升对接效率：**通过规范化的操作和配置，提高外部工作室与我方的对接效率，减少重复工作和人为错误。
- **统一接口与标准：**确保外部工作室与我方之间的接口、配置和流程统一，避免因标准不一致导致的资源冲突或服务不稳定。
- **确保系统稳定性与安全性：**通过健康检查、资源限制、安全控制等措施，确保外部工作室提供的服务稳定、安全地运行在 Kubernetes 平台上。
- **促进协作与沟通：**明确外部工作室与我方之间的接口和责任，确保高效、无缝的跨团队协作。

参考标准

本规范参考以下标准与最佳实践：

- **Kubernetes 官方文档：**遵循 Kubernetes 官方推荐的最佳实践和操作指南。
- **DevOps 标准：**包括 CI/CD 流程、自动化部署、版本控制等，确保外部工作室的应用能够与我方系统无缝集成。
- **容器化标准：**如 Dockerfile 编写规范、镜像管理等，确保外部工作室提供的容器镜像符合平台标准。
- **安全与合规要求：**根据我方企业的安全标准与合规性要求，保障外部工作室提供的应用和数据的安全。

适用范围

本规范适用于：

- 外部工作室与我方的协作，特别是在 Kubernetes 平台上的服务部署、管理和维护。
- 外部工作室提供的所有容器化应用的部署与管理，涵盖但不限于后端服务（Java、Go、Python 等）、前端服务和中间件。
- 外部工作室所涉及的所有资源配置、应用发布、健康检查、监控与日志管理等操作。

容器化应用规范

容器化应用是 Kubernetes 平台部署和管理的核心。为了确保容器镜像的一致性、可维护性和可移植性，以下是外部工作室在对接时需要遵守的 Dockerfile 要求、镜像命名规范、TAG 规范以及构建流程的具体要求。

Dockerfile 要求

1. **基础镜像选择：**
   - 推荐使用官方基础镜像，如 alpine、ubuntu、node 等。确保基础镜像来自可信来源，并且是经过安全审计的。
   - 对于 Java 应用，优选基于 OpenJDK 的镜像；对于 Go 应用，使用官方的 Go 镜像。
2. **镜像层优化：**
   - 尽量减少镜像层数，每一条 RUN 指令后会创建一个新的层，应合并相同操作，减少无用的层。
   - 使用多阶段构建（Multi-stage build）减少最终镜像的体积，避免将构建工具和中间文件包含在生产镜像中。
3. **清理缓存和临时文件：**
   - 在 Dockerfile 中安装软件包或依赖时，应清理临时文件和缓存，防止冗余文件占用空间。例如，在使用 apt-get 时使用 apt-get clean 和 rm -rf /var/lib/apt/lists/*。
4. **容器安全配置：**
   - 非 root 用户：尽量避免在 Dockerfile 中使用 root 用户运行容器，应该创建一个非 root 用户并使用该用户运行应用。
   - 最小权限原则：仅安装应用所需的最低依赖，不要安装任何未使用的工具或库。
   - 设置环境变量：使用 ENV 指令设置环境变量，如 PATH、JAVA_HOME 等。
5. **文件复制与路径规范：**
   - 使用 COPY 指令替代 ADD，因为 COPY 更加明确，不会发生意外的解压或下载行为。
   - 应确保应用代码、配置文件、脚本等文件的目录结构清晰，并避免在 Docker 镜像中出现敏感信息。
6. **健康检查与端口暴露：**
   - 使用 HEALTHCHECK 指令定义容器健康检查，以确保容器在 Kubernetes 中的健康状态。
   - 使用 EXPOSE 指令指定容器内暴露的端口，便于 Kubernetes 配置服务端口。

示例 Dockerfile

```
# 基础镜像

FROM openjdk:17-alpine as build


# 安装构建工具（如果需要）

RUN apk add --no-cache maven


# 设置工作目录

WORKDIR /app


# 复制代码

COPY . /app


# 构建应用

RUN mvn clean package


# 第二阶段，生成最小生产镜像

FROM openjdk:17-alpine


# 创建非 root 用户

RUN adduser -D appuser

USER appuser
```

```
# 设置工作目录

WORKDIR /app



# 复制 JAR 文件

COPY --from=build /app/target/myapp.jar /app/myapp.jar



# 暴露端口

EXPOSE 8080



# 健康检查

HEALTHCHECK CMD curl --fail http://localhost:8080/health || exit 1



# 启动应用

CMD ["java", "-jar", "/app/myapp.jar"]
```

镜像命名规范

1. 镜像命名规则：

- 基本格式：**<registry>/<repository>/<image-name>:<tag>**
- **<registry>**：镜像仓库地址（如 DockerHub、Harbor、AWS ECR 等）。如果是 DockerHub，可以省略。
- **<repository>**：镜像所属的组织或项目名称。
- **<image-name>**：镜像名称，应具有描述性，简短且符合业务。

- **<tag>**：版本标签，遵循语义化版本规范。

2. **命名示例：**
   - **docker.io/mycompany/backend:1.0.0**
   - **harbor.mycompany.com/project1/frontend:v2.3.1**
   - **mycompany/java-app:latest**

3. **镜像命名注意事项：**
   - 避免使用 latest 作为标签，推荐使用具体的版本号。
   - 使用简洁且能描述镜像内容的名称，避免过长的命名。
   - 可以使用组织或团队名作为命名空间，确保镜像名称的唯一性。

TAG 规范

1. **版本控制：**
   - 使用语义化版本控制（Semantic Versioning）：**<MAJOR>.<MINOR>.<PATCH>**
   - **MAJOR**：大版本号，向后不兼容的变更。
   - **MINOR**：小版本号，向后兼容的功能新增。
   - **PATCH**：修复版本，向后兼容的 bug 修复。

例子：

- **v1.0.0**（首次发布）
- **v1.1.0**（添加新功能）
- **v1.0.1**（修复 bug）

2. **标签策略：**

- **稳定版本**：发布稳定版本时，使用 v<version>（如：v1.0.0）。
- **预发布版本**：对于测试或预发布版本，使用如 dev、alpha、beta、rc（如：v1.0.0-beta）。
- **最新版本**：latest 标签仅用于指向最新发布的稳定版本，但在生产环境中应避免使用 latest，以确保一致性和可追溯性。

3. **其他自定义标签：**
   - 根据需要，可以使用一些自定义标签，如环境标签（dev、uat、prod）、构建号（build-12345）等。

构建流程规范

构建流程是确保容器镜像质量与安全的关键环节。为了确保外部工作室提供的镜像符合质量标准、版本管理规范和安全要求，以下是外部工作室在构建镜像过程中应遵循的具体要求。

1. **自动化构建流程**
   - 推荐使用 CI/CD 工具（如 **Jenkins**、**GitLab CI**、GitHub Actions 等）来自动化构建 Docker 镜像，确保镜像的构建、测试和部署流程自动化、可追溯。
   - 在构建流程中，必须自动化运行单元测试和集成测试，以确保镜像构建的质量。如果构建失败，应阻止镜像推送并通知相关人员进行修复。
2. **镜像推送**
   - 镜像推送：在镜像构建成功后，必须自动推送到镜像仓库（如 **Harbor**、AWS ECR 等）。
   - 推送时，镜像标签应遵循规范（如使用版本号、构建号等），确保镜像具有明确的版本标识。例如：**v1.0.0**、**dev-12345**、**beta-1.0.0** 等。
   - 对于不同的环境（如 开发环境、测试环境、生产环境），应使用不同的标签或仓库。例如：
     - **mycompany/backend:dev-1.0.0**（开发环境）
     - **mycompany/backend:test-1.0.0**（测试环境）
     - **mycompany/backend:v1.0.0**（生产环境）

3. **构建缓存和增量构建**

- 在镜像构建过程中，尽量利用 Docker 的 **缓存机制**，以加速镜像构建过程。合理组织 Dockerfile，确保在源代码或依赖没有变化时，尽量避免重新构建不变的层。
- 使用 CI/CD 工具中的 **增量构建功能**，避免每次都重新构建整个镜像。只构建有变更的部分，从而节省时间和计算资源。

4. **镜像扫描与安全检查**

- 在镜像构建后，必须运行 **自动化的镜像安全扫描**，检查镜像是否包含已知漏洞或不符合安全标准。常用的安全扫描工具包括 **Clair、Trivy、Anchore** 等。
- 确保镜像符合企业的安全要求，包括但不限于：
  - 无硬编码密码
  - 无未授权访问权限
  - 确保镜像中的操作系统和依赖项是最新的，避免使用已知存在漏洞的软件包。
  - 镜像不包含敏感信息，如 API 密钥、数据库密码等。

Kubernetes 资源发布与规范化说明

在 Kubernetes 环境中，应用的发布和配置规范化对于提高应用的稳定性、可扩展性和可维护性至关重要。以下是外部工作室在对接时需要遵守的 Kubernetes 资源配置规范。

基本信息规范

1. **命名空间（Namespace）：**

- 所有应用必须在独立的命名空间中运行，以保证环境的隔离和资源管理的清晰性。
- 命名空间应根据应用类别、环境、项目等进行命名。例如：
  - **g01-dev**：开发环境
  - **g01-uat**：UAT 环境

- **g01-prod**：生产环境

2. **Kubernetes 应用名（AppName）：**
   - 应用名应简洁、明确，具有描述性。
   - 避免使用过于简短的命名，应能够准确反映应用的功能。例如，使用 **api-backend**，而不是 **api**。
   - 命名示例：
     - 前端应用：**myapp-frontend**
     - 后端应用：**myapi-backend**
     - 示例：**openapi-backend**，**user-center-backend**，**game-client-frontend**
3. **Service 名称（SvcName）：**
   - 服务名称应与应用名相匹配，便于识别和管理。
   - 服务命名应与应用的功能和目标保持一致。
   - 命名示例：
     - **前端服务**：myapp-frontend-svc
     - **后端服务**：myapi-backend-svc
     - **示例**：openapi-backend-svc，user-center-backend-svc，game-client-frontend-svc
4. **容器端口与服务端口：**

容器的端口和服务端口应保持一致，暴露必要的端口以供应用之间通讯。推荐采用标准端口号。

命名示例：

- **Web 应用**：80 端口
- **HTTPS 服务**：443 端口
- **Java 服务**：http：8080、8081、9000 等，https：8443 或 443。
- **Prometheus Metrics 端口**：8080、9100、8080 等
- **gRPC 服务端口**：50051 端口
- **其他服务端口**：Redis：6379、MySQL：3306、Elasticsearch：9200 端口等。

HPA 配置（Horizontal Pod Autoscaler）

1. **命名规范：**
   - HPA 的命名应与其对应的应用或服务名称一致，以便于管理和辨识。
   - 推荐命名格式：**<应用名>-hpa**。
   - 命名示例：
     - **myapi-backend-hpa**
     - **game-client-frontend-hpa**
2. **最小与最大 Pod 数：**
   - **最小 Pod 数：**应根据应用的基本负载需求，确保应用能处理基本流量。
   - **最大 Pod 数：**应根据集群资源和应用负载设置，避免过度扩容导致资源瓶颈。

配置示例：

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-backend-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app-backend
  minReplicas: 2
```

```
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 80   # 目标 CPU 使用率
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: 80   # 目标内存使用率
```

容器配置

1. 资源请求（requests）与限制（limits）：

- 容器启动时所需最低的资源量（请求）与最大资源量（限制）应合理配置，避免资源浪费或过度分配。
- **配置示例：**

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: example-pod
spec:
  containers:
    - name: example-container
      image: nginx:latest
      resources:
        requests:
          cpu: "128m"
          memory: "256Mi"
        limits:
          cpu: "2"
          memory: "2048Mi"
```

应用健康与故障恢复

健康检查与探针配置

外部工作室团队必须为每个应用配置 **livenessProbe** 和 **readinessProbe**，以便 **Kubernetes** 能够检测应用的健康状态，并根据健康状况自动进行容器重启或流量转发。

- **Readiness Probe**：用于判断容器是否准备好接收流量，容器准备好后才会接收流量。
- **Liveness Probe**：用于检测容器是否仍然存活，若失败则重启容器。

配置示例：

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
      image: nginx:latest
      readinessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
        timeoutSeconds: 2
        failureThreshold: 3
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 10
        periodSeconds: 10
        timeoutSeconds: 2
        failureThreshold: 3
```

错误处理和服务恢复：

- 外部工作室团队需要确保服务设计具有容错能力，能够在部分故障时保持高可用性。比如，确保应用支持 **幂等性** 操作，避免在请求失败时丢失数据或造成状态不一致。
- 研发团队应配合运维团队实现**自愈能力**，例如，服务实例失败时，能够自动重启或切换到备用实例。

自动化故障恢复机制：

- 研发团队应确保服务具备自动化故障恢复能力，特别是对于高可用要求的微服务，必须设计为支持横向扩展，避免单个节点故障导致全局不可用。
- 对于生产环境中关键应用，研发团队需要提供详细的故障恢复方案，包括回滚、重启、数据恢复等步骤。

中间件配置与环境管理

1. 中间件版本管理与配置明确

在部署中间件时，必须明确以下内容：

- 中间件版本管理：确保中间件的版本和配置在开发、测试、生产等环境中的一致性，避免版本不兼容带来的问题。
- 使用 Docker 镜像标签 来指定版本。例如，使用 **postgres:14.2** 来确保使用 PostgreSQL 14.2 版本。
- 推荐在 Helm Chart 或 Kubernetes 部署 YAML 文件 中明确指定版本，而不是使用 **latest** 标签。

版本管理示例：

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: my-database

spec:

  replicas: 1

  template:

    spec:

      containers:

        - name: database
```

```
        image: postgres:14.2   # 使用明确的版本标签
```

2. 配置与环境管理

中间件通常需要大量的配置，如连接信息、密码、日志级别等。为了避免将配置硬编码在代码中，推荐使用 **ConfigMap** 和 **Secret** 来存储配置信息。

- **ConfigMap** 用于存储非敏感配置信息，如数据库连接信息、日志级别等。
- **Secret** 用于存储敏感信息，如数据库密码、API 密钥等，并通过加密存储。

ConfigMap 配置示例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-app-config
  namespace: default
data:
  app.config: |
    database_url: "postgres://user:password@my-database:5432/mydb"
    cache_size: "1024MB"
    log_level: "info"
```

Secret 配置示例：

```
apiVersion: v1
kind: Secret
metadata:
  name: my-db-secret
```

```
  namespace: default
type: Opaque
data:
  db_password: cGFzc3dvcmQ=   # Base64 编码的密码
```

3. 多环境配置支持

- 使用 **Helm Chart** 部署中间件：Helm 能帮助自动化中间件的版本管理、配置管理和依赖管理，避免重复配置和手动部署的错误。
- 应用应支持不同的环境（如**开发、测试、生产**）配置，并通过 Helm 或 Kustomize 等工具管理环境差异。
- 环境分离：确保开发、测试和生产环境完全隔离，避免环境交叉导致的错误。
- 配置版本控制：所有环境配置（如 **values.yaml、ConfigMap**）应进行版本控制，确保一致性。

应用日志与追踪规范

日志格式

为了方便日志收集、解析和跨服务追踪，研发需要确保日志统一使用 **JSON 格式**，并包含以下字段：

- **timestamp**：日志时间戳（UTC）。
- **level**：日志级别（如 info、warning、error、debug）。
- **message**：日志内容。
- **service**：服务名称，用于标识日志来源。
- **traceid**：跨服务追踪 ID，确保同一请求流的日志关联。
- **requestid**：请求唯一标识。
- **clientip**：客户端 IP 地址。

研发团队应确保在应用中记录关键操作（如用户登录、支付请求、数据修改等）的日志，并保证日志数据在必要时能追溯到对应的请求链路。

JSON 格式示例：

```
{
   "timestamp": "2024-11-28T12:00:00Z",
   "level": "error",
   "message": "Failed to connect to the database",
   "service": "my-backend-service",
   "instance": "instance-001",
   "traceid": "b8eacb6e1235bc5678a7a1f9f8ed8755",
   "requestid": "9eec8b7a-2d7b-4eab-8c1c-c8a58b8d86d3",
   "clientip": "192.168.1.100",
   "error_code": "DB_CONN_TIMEOUT",
   "stack_trace": "org.example.DatabaseException: Timeout"
}
```

链路追踪（Tracing）：

- 研发团队需集成链路追踪工具（如 Zipkin、SkyWalking）到应用中，以便跟踪跨服务的请求链路，帮助快速定位性能瓶颈或故障。
- 每个请求在日志中应包含 traceid 和 requestid，确保日志和追踪数据能够关联，便于快速定位问题。

服务暴露与流量管理控制规范

概述

在现代云原生架构中，Istio 是一种强大的服务网格平台，负责微服务间的流量管理、安全控制和监控。使

用 Istio 可以帮助确保服务之间的通信是安全的、可靠的，并且能够进行高效的流量调度和故障恢复。特别是在多环境和多集群的情况下，Istio 提供了统一的流量控制、访问策略、熔断、重试和流量路由等机制。本规范文档定义了通过 Istio 配置互联网访问、服务暴露、流量路由，并指导外部工作室配合运维团队，在 Istio 中配置这些机制时遵循标准化方法，确保服务访问质量。

规范标准

1. **访问网关（IngressGateway）**
   - 使用 **Istio IngressGateway** 作为集群的流量入口点，外部工作室和研发团队需配合确保服务对外暴露的接口符合安全要求。
   - 通过 **NLB（网络负载均衡器）**与 **Istio** 配合，使外部流量能够路由到集群中的服务。
   - 所有外部流量应通过 HTTPS 或加密的通道进入，研发团队需协作提供相关的 SSL/TLS 证书，确保服务的安全性。
2. **服务暴露和路由**
   - 使用 **VirtualService** 进行细粒度的流量路由控制，研发团队需根据业务需求协作定义服务的路由规则。
   - 对于每个服务，按路径、主机或 HTTP 方法定义路由规则。例如，研发团队需要确保在应用的路由配置中添加必要的路径（如 **/api**、**/auth** 等），并确保它们与 **VirtualService** 的配置一致。

研发配合要求：

- **路径路由：**开发团队需要确保应用支持在 **IngressGateway** 中定义的路由路径。例如，**/api** 路由到应用服务，**/auth** 路由到身份认证服务。
- **API 版本控制：**研发团队应确保应用支持根据需求进行灵活的版本控制，并在 **VirtualService** 中配置相应的流量分配规则，支持如 A/B 测试、灰度发布等。

3. **服务负载均衡和策略（DestinationRule）**
   - 使用 **DestinationRule** 配置服务的负载均衡、连接池和出站流量的策略。

- 研发团队需配合配置服务的负载均衡方式（如 **ROUND_ROBIN**）和连接池参数（如最大连接数等）。

研发配合要求：

- **状态码处理：**研发团队需要配合 **DestinationRule** 中的熔断和重试机制，确保应用在出现错误时能返回标准化的 HTTP 状态码（如 5xx 错误）。开发人员应确保服务在处理错误时返回合适的 HTTP 状态码，以触发 Istio 的熔断机制。
- **错误处理与重试机制：**研发团队应协作实现服务端的重试策略。例如，在服务发生瞬时故障时，可以通过 VirtualService 配置自动重试请求，减少对用户体验的影响。

版本管理与发布流程

在版本管理与发布流程的设计中，确保整个流程清晰、高效且安全是非常关键的。以下是根据您的需求及标准的最佳实践整理的版本管理与发布流程规范，涵盖了镜像标签规范、镜像同步、发布流程、审批机制、仓库同步机制、升级与回滚机制等内容。

镜像标签规范

为了统一和标准化镜像版本管理，外部工作室应遵循以下镜像标签（tag）格式。镜像的标签由以下部分组成：

- **时间戳：**使用 yyyyMMddHHmmss 格式表示镜像构建的时间。
- **编号：**在时间戳后加上编号（例如：20241129063005-2），用于标识该时间点下的多个镜像版本。

该格式的规范可以确保镜像的唯一性、可追溯性以及有效的版本管理，避免多个版本混淆或版本不可追踪的问题。

镜像同步流程

1. **镜像构建：**

- 开发团队在完成代码提交和构建之后，会生成包含新功能或修复的镜像，并为其打上符合标签规范的标签。

2. **镜像推送至开发仓库：**

   - 镜像首先推送至开发环境的仓库，通常是外部工作室的 **Harbor 仓库**，用于测试和验证。

3. **同步至运维仓库：**

   - 外部工作室的镜像应通过与运维仓库的镜像同步流程进行同步，以备后续使用。
   - 内部仓库同步机制应确保开发和运维团队能够在统一的仓库中管理和验证镜像版本，避免不同版本在不同仓库中分散。

4. **UAT 环境验证：**

   - 镜像被推送至 **UAT** 环境，经过功能和性能测试验证，确保该版本没有引入新的问题或缺陷。

5. **推送至生产环境：**

   - 在 UAT 环境测试通过后，镜像通过外部团队的发布申请与审批流程推送至生产环境。

发布流程

- **发布计划：** 每次发布都需要外部工作室提供一个详细的发布计划，包含以下内容：

  - 目标环境（如 **UAT**、**生产环境**）。
  - 升级的镜像标签（**tag**）。
  - 回滚镜像的标签（tag），以便回滚使用。
  - 升级验证流程和时间计划。

- **发布申请与审批：**

- 外部工作室根据发布计划向相关部门（如运维部门）提交发布申请。
- 相关部门（如研发 HOD 与运维 HOD）会评审发布计划，确保所有测试和验证工作都已完成，镜像已通过验收。
- 审批通过后，相关团队（运维团队或外部工作室）在规定的时间内执行部署。

升级与回滚机制

1. **镜像标签统一管理：**

   - 确保所有镜像都使用统一的标签规范（如时间戳-编号），避免不同团队或服务使用不同的标签方式。
   - 在发布计划中明确指定标签，不使用 latest 标签，以保证版本的可追溯性。

2. **严格的审批流程：**

   - 所有发布都必须经过严格的审批流程，确保发布前进行充分验证和风险评估。
   - 发布计划中必须包含升级的标签和回滚的标签，以便在出现问题时快速响应。

3. **发布与回滚验证：**

   - 在进行升级和回滚时，需要进行必要的验证步骤，以确保升级后的功能正常，回滚后的服务稳定。
   - 发布计划应包括详细的回滚流程和回滚验证步骤，确保快速恢复生产环境，避免生产服务中断。

4. **仓库同步机制：**

   - 镜像在同步至生产环境前，必须经过统一的仓库验证流程，确保各个环境的镜像版本一致。
   - 需要在发布前进行一次最终确认，确保镜像标签的准确性和版本一致性。

Prometheus Metrics

Prometheus 指标端口与路径

外部工作室团队应确保每个应用容器都必须暴露 **Prometheus** 指标接口，**Prometheus** 会通过这个接口抓取容器的性能指标。对于每个服务和 **Pod**，需要指定 **Prometheus** 指标端口和路径。在 **Kubernetes** 中，**Prometheus** 的默认指标路径通常是 **/metrics**，需要确保应用容器正确暴露这个路径，以便 **kube-prometheus-stack** 能够抓取数据。

端口与路径配置

在应用的容器中，通常会启用 Prometheus 指标端口和路径，默认端口是 **9090**，而指标路径通常是 **/metrics**。您可以根据具体应用的需求，配置不同的端口和路径。

- **Prometheus 默认指标路径：/metrics**
- **Prometheus 默认端口：9090**

示例（以 Java Spring Boot 应用为例）：

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: example-image
    ports:
      - containerPort: 8080  # 应用的服务端口
```

```
    - containerPort: 9090   # Prometheus 指标端口
  env:
    - name: SPRING_METRICS_EXPORT_PROMETHEUS_ENABLED
      value: "true"
```

对于其他类型的应用（如 **Go** 或 **Node.js**），也需要将应用的 **/metrics** 路径暴露出来，确保 **Prometheus** 可以抓取指标。