Kubernetes Operator. Логирование и трейсы. ElasticSearch, FluentD, Kibana

Kubernetes Operator

Kubernetes Operator — это расширение Kubernetes, которое использует пользовательские ресурсы (Custom Resource Definitions, CRD) для автоматизации управления сложными приложениями и их жизненным циклом. Операторы позволяют описывать, развертывать и управлять состоянием приложений в Kubernetes, используя декларативный подход.

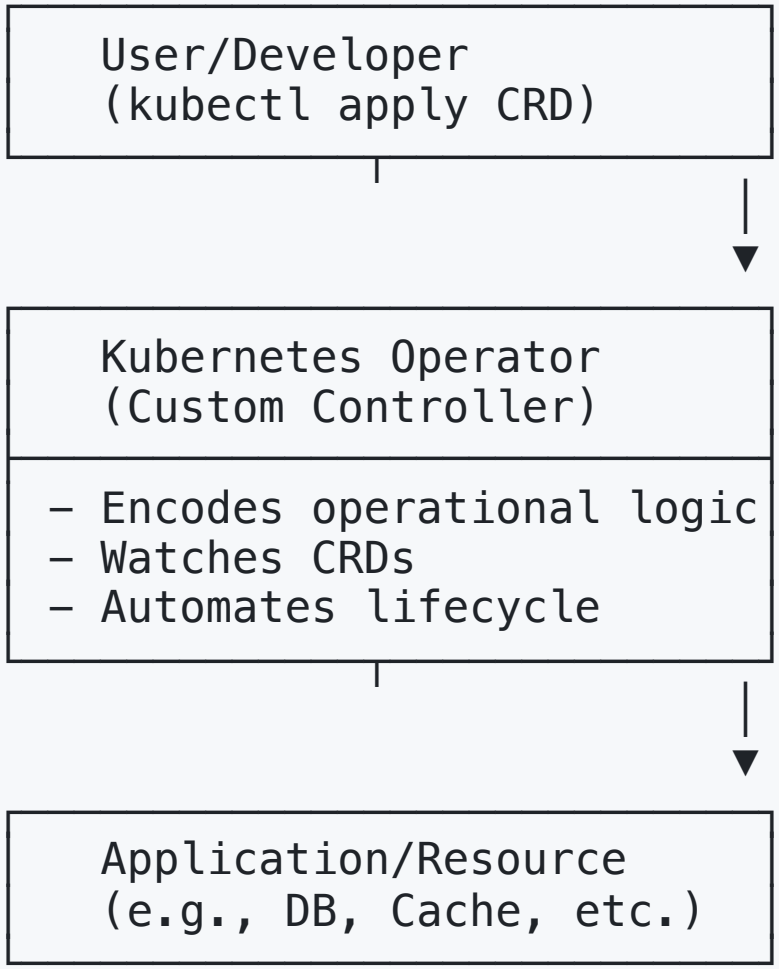
Custom Resource Definition (CRD) в Kubernetes

Custom Resource Definition (CRD) — это механизм в Kubernetes, который позволяет пользователям определять свои собственные типы ресурсов. CRD расширяет API Kubernetes, добавляя новые типы объектов, которые можно использовать так же, как и встроенные ресурсы Kubernetes, такие как Pods, Services и Deployments.

4 Problems Kubernetes Operators Can Solve

- **Reduce Complexity**
  Операторы инкапсулируют операционные знания, упрощая развертывание и управление сложными приложениями.
- **Enable Custom Logic**
  Операторы расширяют возможности Kubernetes, добавляя пользовательскую автоматизацию для специфических задач приложений.
- **Support Custom Resources**
  Операторы вводят пользовательские определения ресурсов (CRD), чтобы представлять и управлять нестандартными ресурсами.
- **Automate Ongoing Management**
  Операторы автоматически выполняют обновления, резервное копирование, масштабирование и другие задачи жизненного цикла CRD.

[Operator Actions Diagram]



Tips for Working with Kubernetes Operators

- Use finalizers for cleanup tasks
- Leverage CRD validation
- Implement leader election
- Use versioning for custom resources
- Integrate with Prometheus for monitoring

How Operators Manage Kubernetes Applications

- **Domain-specific knowledge**
  - Операторы инкапсулируют экспертизу для конкретных доменов (например, базы данных, брокеры сообщений).
  - Упрощают развертывание и управление сложными приложениями.
- **Automate manual tasks**
  - Автоматически выполняет обновления, резервное копирование, масштабирование и другие задачи.
  - Снижает ручную нагрузку для команд.
- **Easier deployment of foundation services**
  - Упрощает развертывание базовой инфраструктуры (например, баз данных, очередей).
  - Экономит время при запуске новых приложений.

How Operators Improve Reliability and Consistency

- **Consistent software distribution**
  - Обеспечивает единообразное развертывание приложений во всех кластерах.
- **Reduce support burdens**
  - Проактивно выявляет и устраняет проблемы.
  - Снижает операционные издержки.
- **Implement SRE principles**
  - Поддерживает лучшие практики обеспечения надежности и доступности.

Popular Kubernetes Operators

- **Prometheus Operator**
  - Разворачивает и управляет экземплярами мониторинга Prometheus в Kubernetes.
  - Автоматизирует настройку, масштабирование и управление Prometheus, Alertmanager и связанными ресурсами.
  - Позволяет настраивать мониторинг декларативно с помощью Kubernetes-манифестов.
- **Grafana Operator**
  - Управляет экземплярами Grafana и дашбордами внутри Kubernetes.
  - Упрощает развертывание, настройку и обновление Grafana.
  - Интегрируется с Prometheus для расширенной визуализации.

Popular Kubernetes Operators

- **Elastic Cloud on Kubernetes (ECK) Operator**
  - Разворачивает и управляет кластерами Elasticsearch и Kibana в Kubernetes.
  - Поддерживает автоматическое масштабирование, поэтапные обновления и восстановление после сбоев.
  - Предоставляет пользовательские ресурсы для удобного управления компонентами Elastic Stack.
- **RBAC Manager Operator**
  - Упрощает управление правилами контроля доступа на основе ролей (RBAC) в Kubernetes.
  - Предоставляет пользовательские ресурсы для описания и синхронизации RBAC-политик.
  - Гарантирует единообразное применение RBAC-правил по всему кластеру.

What Is the Operator SDK?

- **Toolkit for building Kubernetes operators**
  - Предоставляет CLI-инструменты, библиотеки и средства автоматизации для разработки операторов.
  - Поддерживает несколько языков (Go, Ansible, Helm).

- **Key Components:**
  - **CLI** — команды для создания шаблонов, сборки, тестирования и деплоя операторов.
    - Пример: `operator-sdk init --domain=mydomain.com --repo=github.com/example/my-operator`
  - **Автоматизация сборки через Make** — используются Makefile для автоматизации сборки, тестирования и деплоя.
    - Пример: `make docker-build` для сборки образа оператора.
  - **Готовые команды Make** — типовые задачи, такие как генерация кода, создание манифестов и тестирование, стандартизированы.
    - Пример: `make manifests` для генерации CRD.
  - **Operator Lifecycle Manager (OLM)** —
    - Управляет установкой, обновлением и удалением операторов в кластере.

Example CRD resource

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: sampledbs.app.example.com
spec:
  group: app.example.com
  names:
    kind: SampleDB
    plural: sampledbs
  scope: Namespaced
  version: v1
```

What is the Prometheus Operator?

- **Prometheus Operator** — это Kubernetes-оператор, который автоматизирует установку и настройку Prometheus — одной из ведущих time-series баз данных для мониторинга.
- Упрощает развертывание Prometheus в Kubernetes с помощью **Custom Resource Definitions (CRD)** для конфигурации.
- Обеспечивает **Kubernetes-нативный опыт управления** экземплярами Prometheus.
- Автоматизирует развертывание, настройку и обнаружение целей для Prometheus и связанных компонентов.

Prometheus vs Prometheus Operator

- **Prometheus**:
  - Открытая time-series база данных для хранения, запросов и визуализации метрик.
  - Может быть развернут в различных средах (в виде бинарника, из исходников, в контейнере).
- **Prometheus Operator**:
  - Специализированный инструмент для Kubernetes для развертывания и управления Prometheus.
  - Использует CRD для упрощения конфигурации и управления.
  - Не обязателен для запуска Prometheus, но значительно облегчает интеграцию с Kubernetes.

Prometheus Operator vs Prometheus Adapter

- **Prometheus Operator**:
  - Разворачивает и управляет Prometheus в Kubernetes с помощью CRD.
  - Отвечает за конфигурацию, масштабирование и управление жизненным циклом.
- **Prometheus Adapter**:
  - Экспортирует метрики Prometheus в API метрик Kubernetes.
  - Позволяет использовать возможности Kubernetes, такие как **Horizontal Pod Autoscaling** на основе пользовательских метрик.
  - Используется, когда нужно, чтобы Kubernetes принимал решения (например, масштабирование) на основе данных Prometheus.

Prometheus Operator: Use Cases & Benefits

- **Simplicity**:
  - Разворачивает Prometheus за считанные минуты с помощью готовых манифестов.
  - Настройка через CRD — не нужно изучать синтаксис конфигурации Prometheus.
- **Scalability**:
  - Поддерживает бесшовные, динамические обновления без простоев.
- **Kubernetes-native**:
  - Полная интеграция с Kubernetes для нативного и удобного опыта эксплуатации.

Prometheus Operator: Use Cases & Benefits

- **Automatic Target Discovery**:
  - Находит и автоматически собирает метрики с ресурсов по меткам.
- **Easy Configuration**:
  - Все компоненты (таргеты, правила, алерты) управляются через CRD.
- **Complete Monitoring Stack**:
  - Разворачивает Prometheus, Alertmanager и Grafana для полной наблюдаемости.

Prometheus Operator CRDs Overview

- **Prometheus**: Main Prometheus server configuration (storage, replication, etc.)
- **PrometheusAgent**: Runs Prometheus in agent mode for remote write.
- **Alertmanager**: Configures alert routing and notifications.
- **ThanosRuler**: Evaluates alerting and recording rules with Thanos.
- **ServiceMonitor**: Scrapes metrics from Kubernetes Services.
- **PodMonitor**: Scrapes metrics from specific Pods.
- **Probe**: Configures static targets and Ingress scraping.
- **ScrapeConfig**: Custom scraping configuration (often for external resources).
- **PrometheusRule**: Defines alerting and recording rules.
- **AlertmanagerConfig**: Specifies alert routing and receivers.

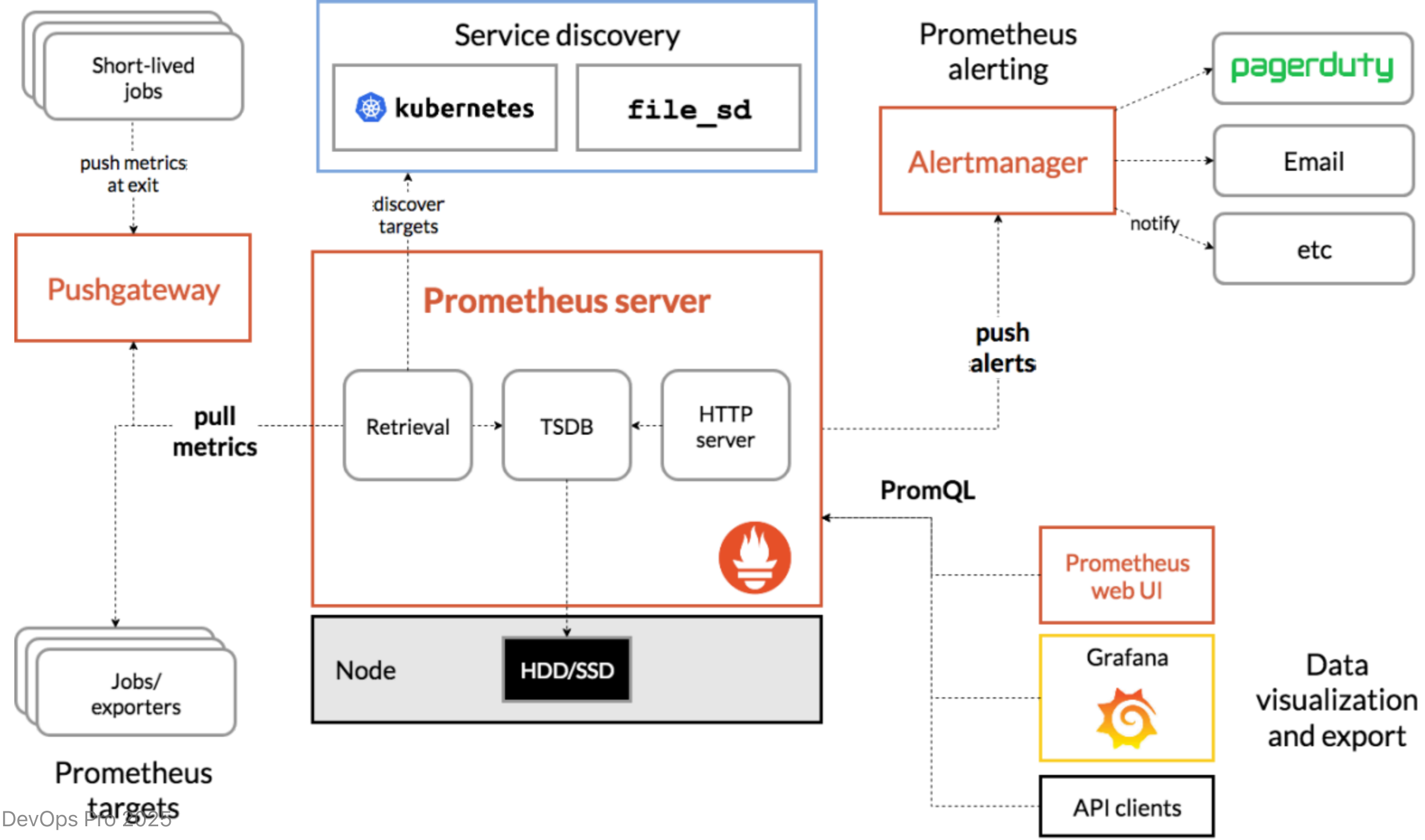
Example: ServiceMonitor CRD

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
    name: example-servicemonitor
spec:
    selector:
        matchLabels:
            app: my-app
    endpoints:
        - port: http
```

- **ServiceMonitor** tells Prometheus to scrape metrics from services with the label `app: my-app` on the `http` port.

Key Takeaways

- **Prometheus Operator** — это рекомендуемый способ запуска Prometheus в Kubernetes.
- Обеспечивает автоматизацию, масштабируемость и простоту использования через CRD.
- Интегрируется с экосистемой Kubernetes для бесшовного мониторинга и алертинга.
- Используйте CRD, такие как ServiceMonitor и PrometheusRule, чтобы управлять мониторингом как кодом.


Short-lived jobs
push metrics at exit
Pushgateway
Service discovery
kubernetes
file_sd
discover targets
Prometheus server
Retrieval
TSDB
HTTP server
pull metrics
Node
HDD/SSD
Jobs/ exporters
Prometheus targets
Prometheus alerting
Alertmanager
pagerduty
Email
etc
notify
push alerts
PromQL
Prometheus web UI
Grafana
API clients
Data visualization and export

Основные компоненты Prometheus Stack

Prometheus Stack — это комплексное решение для мониторинга Kubernetes, включающее все необходимые компоненты для сбора, хранения, визуализации и оповещения на основе метрик.

- **Prometheus Operator** — автоматизирует установку, настройку и управление всеми компонентами мониторинга в Kubernetes.
- **GitOps подход** — конфигурация мониторинга и алертинга хранится как код (YAML-манифесты), что обеспечивает прозрачность, версионирование и воспроизводимость.

GitOps: Конфигурация как код

- Все настройки Prometheus, Grafana, Alertmanager и других компонентов описываются в YAML-файлах.
- Использование Git-репозитория для хранения и управления конфигурациями.
- Автоматическое применение изменений через CI/CD пайплайны
- Преимущества:
  - Аудит изменений
  - Быстрое восстановление и откат
  - Единый источник правды для всей инфраструктуры мониторинга

Grafana: Визуализация метрик

- **Grafana** — инструмент для построения дашбордов и визуализации метрик из Prometheus и не только.
- Метрики из Prometheus автоматически доступны в Grafana через datasource.
- **Grafana дашборды как код**:
  - Дашборды описываются в JSON/YAML и хранятся в репозитории.
  - Используется механизм provisioning для автоматической загрузки дашбордов при старте Grafana.
  - На проекте реализовано через хранение дашбордов в git и автоматическую доставку в кластер.
- Преимущества:
  - Версионирование и совместная работа над дашбордами
  - Быстрое восстановление и переносимость

Alertmanager: Алертинг и управление оповещениями

- **Alertmanager** — компонент для обработки алертов, поступающих из Prometheus.
- **Алерты как код**:
  - Правила алертинга описываются в PrometheusRule CRD и хранятся в git.
  - Любое изменение алертов проходит через pull request и review.
- **Роутинг алертов**:
  - Гибкая маршрутизация алертов по получателям (email, Slack, Telegram и др.) на основе лейблов.
- **Сайленсы и автоматизация**:
  - Сайленсы (временное отключение алертов) можно создавать вручную или автоматически через API.
  - Интеграция с CI/CD для автоматического создания сайленсов на время

Экспортеры: Сбор метрик

- **Node Exporter** — собирает метрики с хостов (CPU, память, диски, сеть).
- **Blackbox Exporter** — проверяет доступность внешних сервисов (HTTP, TCP, ICMP).
- **Kube-state-metrics** — предоставляет метрики о состоянии объектов Kubernetes (Pods, Deployments, Nodes и др.).
- Экспортеры разворачиваются как отдельные поды или DaemonSet'ы и автоматически обнаруживаются Prometheus через ServiceMonitor/PodMonitor CRD.

Полный список доступных экспортеров:
https://prometheus.io/docs/instrumenting/exporters/

Prometheus: Сердце мониторинга

- Сбор, хранение и агрегация метрик со всех экспортеров и приложений.
- Гибкая система запросов (PromQL) для анализа данных.
- Хранение метрик по умолчанию — 15 дней (можно настраивать).

Thanos: Долговременное хранение метрик

- **Thanos** расширяет возможности Prometheus:
  - Долговременное хранение метрик в облачных хранилищах (S3, GCS и др.).
  - Объединение данных из нескольких Prometheus инстансов (federation).
  - Высокая доступность и отказоустойчивость.
- Используется для хранения метрик за месяцы и годы, а также для построения глобальных дашбордов.

Ключевые преимущества подхода

- **Автоматизация**: Все компоненты управляются оператором и описываются как код.
- **Масштабируемость**: Легко добавлять новые источники метрик и алерты.
- **Надежность**: Долговременное хранение и высокая доступность с помощью Thanos.
- **Гибкость**: Быстрое внедрение изменений через GitOps и CI/CD.

Проблемы эксплуатации Prometheus и их решения

1. Высокое потребление памяти

- **Проблема:**
  - Prometheus может потреблять много памяти, особенно при большом количестве метрик и длинном сроке хранения.
- **Решения:**
  - Оптимизируйте retention (параметр `--storage.tsdb.retention.time`).
  - Используйте remote storage (например, Thanos, VictoriaMetrics) для долговременного хранения.
  - Уменьшайте частоту сбора метрик для менее важных источников.
  - Следите за количеством time series и используйте label drop/keep в scrape-конфиге.

2. Срок хранения метрик

- **Проблема:**
  - По умолчанию Prometheus хранит метрики 15 дней, что может быть недостаточно для анализа трендов.
- **Решения:**
  - Увеличьте retention, если хватает ресурсов (`--storage.tsdb.retention.time=30d`).
  - Для долгосрочного хранения используйте Thanos, Cortex или VictoriaMetrics.
  - Регулярно архивируйте старые данные.

3. High Cardinality (Высокая кардинальность метрик)

- **Проблема:**
  - Метрики с большим количеством уникальных комбинаций label (например, user_id, request_id) резко увеличивают нагрузку на память и диск.
- **Решения:**
  - Не используйте динамические значения (user_id, uuid) в label.
  - Используйте label drop/keep в relabeling.
  - Агрегируйте метрики на стороне приложения.
  - Мониторьте количество time series (`prometheus_tsdb_head_series`).

4. Label Clash (Конфликты label)

- **Проблема:**
  - Одинаковые label с разным смыслом или ошибочные label приводят к путанице и ошибкам в запросах.
- **Решения:**
  - Внедрите naming convention для label.
  - Используйте linting/валидацию метрик на этапе CI/CD.
  - Проверяйте label на уникальность и корректность.

5. Отсутствие авторизации "из коробки"

- **Проблема:**
  - Prometheus web-интерфейс и API доступны всем без авторизации.
- **Решения:**
  - Используйте reverse proxy с авторизацией (например, oauth2-proxy, nginx + basic auth).
  - Пример с oauth2-proxy:
    - Разверните oauth2-proxy перед Prometheus.
    - Настройте интеграцию с SSO (Google, GitHub, LDAP).
    - Ограничьте доступ к / и /api/v1/* только авторизованным пользователям.

6. Другие распространённые проблемы

- **Потеря метрик при рестарте/сбое**
  - Используйте remote write для резервирования данных.
- **Сложность управления большим количеством targets**
  - Используйте сервис-дискавери и шаблоны ServiceMonitor/PodMonitor.
- **Проблемы с производительностью при большом количестве запросов**
  - Ограничьте сложные запросы, используйте recording rules.
  - Настройте query timeout и limit.

Краткие рекомендации

- Следите за количеством time series и label cardinality.
- Не используйте динамические значения в label.
- Используйте внешнее хранилище для долгосрочных метрик.
- Ограничивайте доступ к Prometheus через авторизацию.
- Регулярно обновляйте Prometheus и экспортеры.

Альтернативное решение: VictoriaMetrics

VictoriaMetrics — это высокопроизводительное решение для хранения и обработки метрик, полностью совместимое с Prometheus API. Оно может использоваться как drop-in замена Prometheus, обеспечивая масштабируемость, долговременное хранение и высокую доступность.

Основные компоненты VictoriaMetrics Stack

- **VMStack** — аналог Prometheus Stack, деплоит все компоненты мониторинга с помощью VictoriaMetrics Operator.
- **Полная поддержка Prometheus API** — можно использовать существующие экспортеры, ServiceMonitor, PrometheusRule и другие CRD.

Single Version (для большинства сценариев)

- **vmsingle**
  - Централизованное долговременное хранилище метрик.
  - Поддерживает эффективную компрессию и быстрые запросы.
- **vmagent**
  - Скрапит (собирает) метрики с таргетов и отправляет их в vmsingle.
  - Может использовать локальный кэш на диске для временного хранения, если основное хранилище недоступно.
- **vmalert**
  - Выполняет PromQL-запросы к хранилищу и отправляет алерты в Alertmanager.
  - Поддерживает PrometheusRule CRD для описания правил алертинга.

Cluster Version (для высокой нагрузки и отказоустойчивости)

VictoriaMetrics поддерживает кластерный режим для масштабирования и высокой доступности:

- **vmstorage**
  Хранит метрики, масштабируется горизонтально. Поддерживает репликацию и шардирование данных.
- **vminsert**
  Принимает метрики от vmagent и распределяет их по vmstorage.
- **vmselect**
  Обрабатывает запросы на чтение (PromQL) и агрегирует данные из vmstorage.
- **vmagent/vmalert**
  Аналогично single-версии, могут быть развернуты в нескольких экземплярах для отказоустойчивости.

Преимущества VictoriaMetrics

- **Масштабируемость** — легко масштабируется горизонтально в кластерном режиме.
- **Долговременное хранение** — эффективная компрессия, поддержка хранения метрик за годы.
- **Высокая производительность** — быстрые запросы и низкое потребление ресурсов.
- **Совместимость** — поддержка Prometheus API, экспортеров и инструментов визуализации (Grafana).

VictoriaMetrics: Итоги

- **VictoriaMetrics** — современная альтернатива Prometheus для хранения и обработки метрик.
- Поддерживает все привычные инструменты и подходы (экспортеры, CRD, Grafana).
- Обеспечивает масштабируемость, высокую производительность и долговременное хранение.
- Позволяет избежать типовых проблем Prometheus за счет оптимизаций и гибкой архитектуры.

VictoriaMetrics CRDs и их аналоги в Prometheus

VictoriaMetrics Operator поддерживает собственные CRD для управления мониторингом, а также умеет работать с объектами Prometheus Operator. Это позволяет легко мигрировать с Prometheus на VictoriaMetrics без изменения существующих манифестов.

> VictoriaMetrics Operator может автоматически обнаруживать и использовать объекты ServiceMonitor, PodMonitor, PrometheusRule и Probe, созданные для Prometheus Operator.
> При создании или удалении объектов Prometheus CRD, VictoriaMetrics Operator создает или удаляет соответствующие объекты у себя.

Сопоставление CRD: VictoriaMetrics vs Prometheus

VictoriaMetrics CRD	Аналог в Prometheus Operator
VMServiceScrape	ServiceMonitor
VMPodScrape	PodMonitor
VMRule	PrometheusRule
VMAlert	Alertmanager
VMNodeScrape	-
VMProbe	Probe
VMStaticScrape	-
operator.victoriametrics.com/v1	monitoring.coreos.com

Как перейти с Prometheus Operator на VictoriaMetrics Operator

1. **Совместимость CRD**
   - VictoriaMetrics Operator "понимает" CRD Prometheus Operator (`ServiceMonitor`, `PodMonitor`, `PrometheusRule` и др.).
   - Нет необходимости переписывать существующие манифесты — можно использовать те же объекты.

2. **Автоматическая конвертация**
   - При создании объектов Prometheus CRD, VictoriaMetrics Operator автоматически создает у себя аналоги (`VMServiceScrape` , `VMPodScrape` , `VMRule` и др.).
   - При удалении — удаляет соответствующие объекты.

3. **Управление объектами**
    - VictoriaMetrics Operator может взять на себя управление всеми объектами мониторинга, созданными для Prometheus.
    - Это позволяет плавно мигрировать без простоев и изменений в инфраструктуре.

Пример: Использование ServiceMonitor

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  endpoints:
    - port: http
```

> Такой объект будет автоматически обработан VictoriaMetrics Operator и преобразован во внутренний объект `VMServiceScrape`.

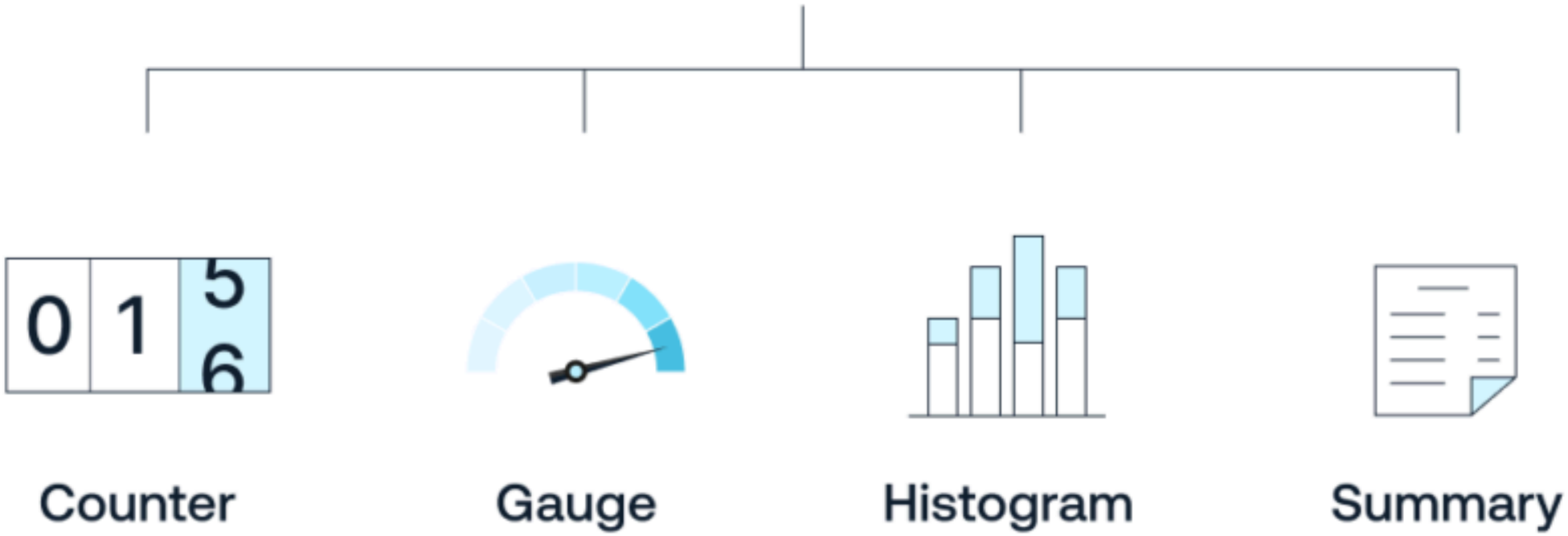
Преимущества подхода

- **Бесшовная миграция**: можно использовать существующие манифесты Prometheus Operator.
- **Минимум изменений**: не требуется переписывать инфраструктуру мониторинга.
- **Гибкость**: поддержка как собственных CRD VictoriaMetrics, так и стандартных CRD Prometheus Operator.
- **Автоматизация**: управление объектами мониторинга полностью автоматизировано оператором.

Итог

VictoriaMetrics Operator обеспечивает полную совместимость с объектами Prometheus Operator, что делает миграцию простой и быстрой. Вы можете использовать привычные CRD, а оператор сам возьмет на себя их обработку и управление.

Prometheus Metrics Types




```
rpc_durations_histogram_seconds_sum 4.73125563626474
rpc_durations_histogram_seconds_count 484135
# HELP rpc_durations_seconds RPC latency distributions.
# TYPE rpc_durations_seconds summary
rpc_durations_seconds{service="exponential",quantile="0.5"} 7.1548174397379 4e-07
rpc_durations_seconds{service="exponential",quantile="0.9"} 2.4107430862406495e-06
rpc_durations_seconds{service="exponential",quantile="0.99"} 4.765333807409357e-06
rpc_durations_seconds_sum{service="exponential"} 0.725660594967505
rpc_durations_seconds_count{service="exponential"} 725573
rpc_durations_seconds{service="normal",quantile="0.5"} 2.624443173024204e-05
rpc_durations_seconds{service="normal",quantile="0.9"} 0.0002703790327934205 5
rpc_durations_seconds{service="normal",quantile="0.99"} 0.0004809281651158581 5
rpc_durations_seconds_sum{service="normal"} 4.731255636264 74
rpc_durations_seconds_count{service="normal"} 484135
rpc_durations_seconds{service="uniform",quantile="0.5"} 0.0001128602661307069 5
rpc_durations_seconds{service="uniform",quantile="0.9"} 0.0001829265170498833
rpc_durations_seconds{service="uniform",quantile="0.99"} 0.0001983843200488370 6
rpc_durations_seconds_sum{service="uniform"} 36.34058433619449
rpc_durations_seconds_count{service="uniform"} 363327
```

Counter

Пример использования Counter:

```
counter := prometheus.NewCounter(prometheus.CounterOpts{
    Name: "http_requests_total",
    Help: "Total number of HTTP requests",
})
prometheus.MustRegister(counter)

// Увеличение счетчика на 1
counter.Inc()

// Увеличение счетчика на определенное значение
counter.Add(5)
```

Gauge

Пример использования Gauge:

```
gauge := prometheus.NewGauge(prometheus.GaugeOpts{
    Name: "memory_usage_bytes",
    Help: "Current memory usage in bytes",
})
prometheus.MustRegister(gauge)

// Установка значения
gauge.Set(1024)

// Увеличение значения
gauge.Inc()

// Уменьшение значения
gauge.Dec()
```

Histogram

Пример использования Histogram:

```
histogram := prometheus.NewHistogram(prometheus.HistogramOpts{
    Name:    "request_duration_seconds",
    Help:    "Histogram of request durations in seconds",
    Buckets: prometheus.DefBuckets,
})
prometheus.MustRegister(histogram)

// Наблюдение за значением
histogram.Observe(0.5)
```

Summary

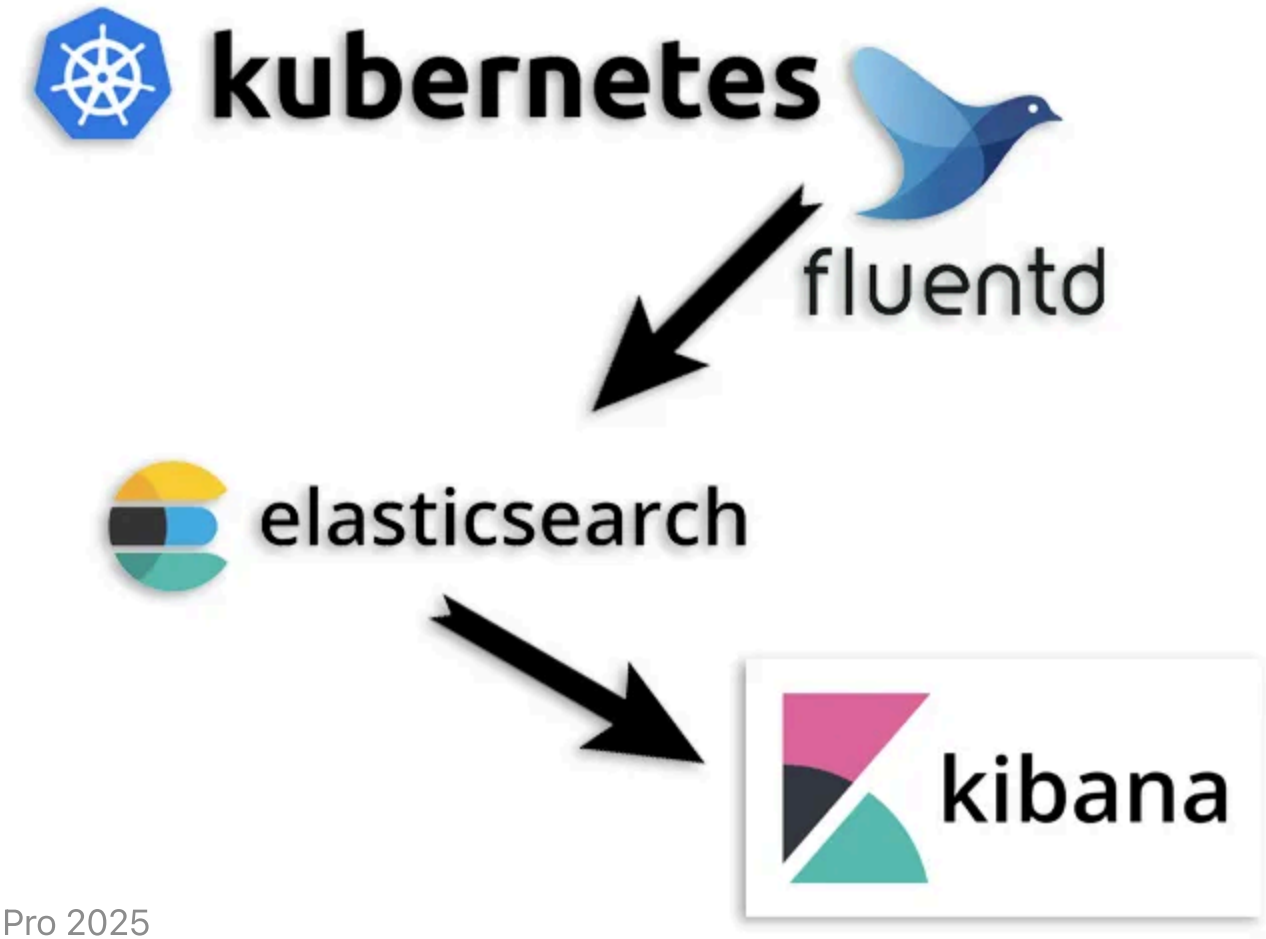
```
summary := prometheus.NewSummary(prometheus.SummaryOpts{
    Name:       "request_duration_seconds_summary",
    Help:       "Summary of request durations in seconds",
    Objectives: map[float64]float64{0.5: 0.05, 0.9: 0.01, 0.99: 0.001},
})
prometheus.MustRegister(summary)

// Наблюдение за значением
summary.Observe(0.5)
```

Metric types

- **Counter**: Counters are used for the cumulative counting of events.
  The value of a counter can only increase or be reset to zero when it is restarted—it will never decrease on its own.
- **Gauge**: Gauges typically represent the latest value of measurement.
  Unlike a counter, a gauge can go up or down depending on what's happening with the endpoint that's being measured.
- **Histogram**: A histogram takes a sample of observations and counts them in buckets that can be a configuration by the user.
- **Summary**: Like a histogram, a summary samples observations in one place.

написать самари подвести итоги почему прометей это хорошо

EFK Stack (Elasticsearch, Fluentd, Kibana)



Elasticsearch

elastic

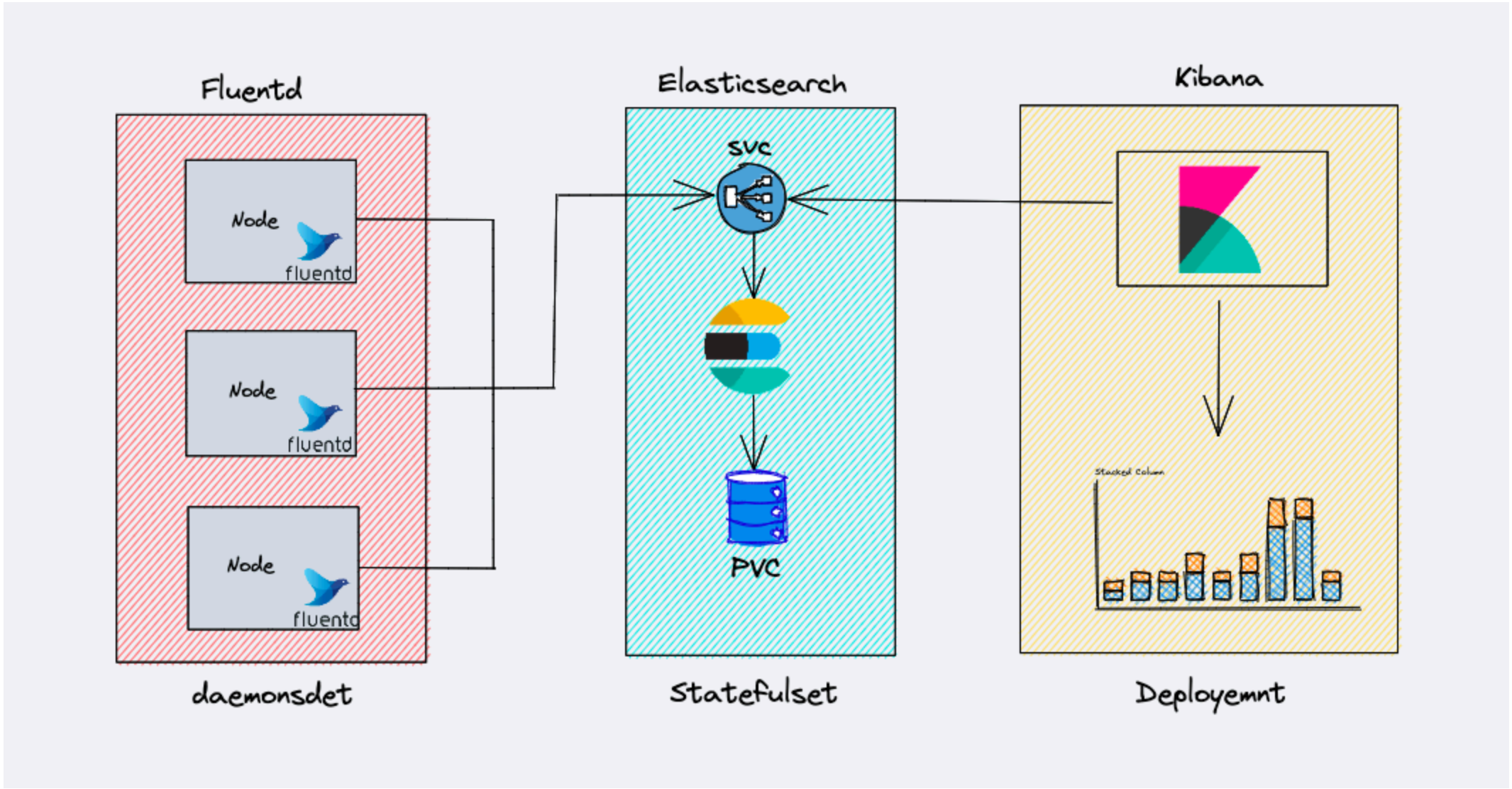
Основные компоненты Elasticsearch

Индекс: Логическая структура, которая содержит документы и их свойства.
Индексы могут быть разбиты на несколько шардов для повышения производительности и масштабируемости.
Документ: Основная единица данных в Elasticsearch, хранящаяся в формате JSON.
Шард: Фрагмент индекса, который может быть размещен на разных узлах кластера для распределения нагрузки.
Реплика: Копия шарда, используемая для повышения отказоустойчивости и производительности.

Fluentd

Kibana

Kibana

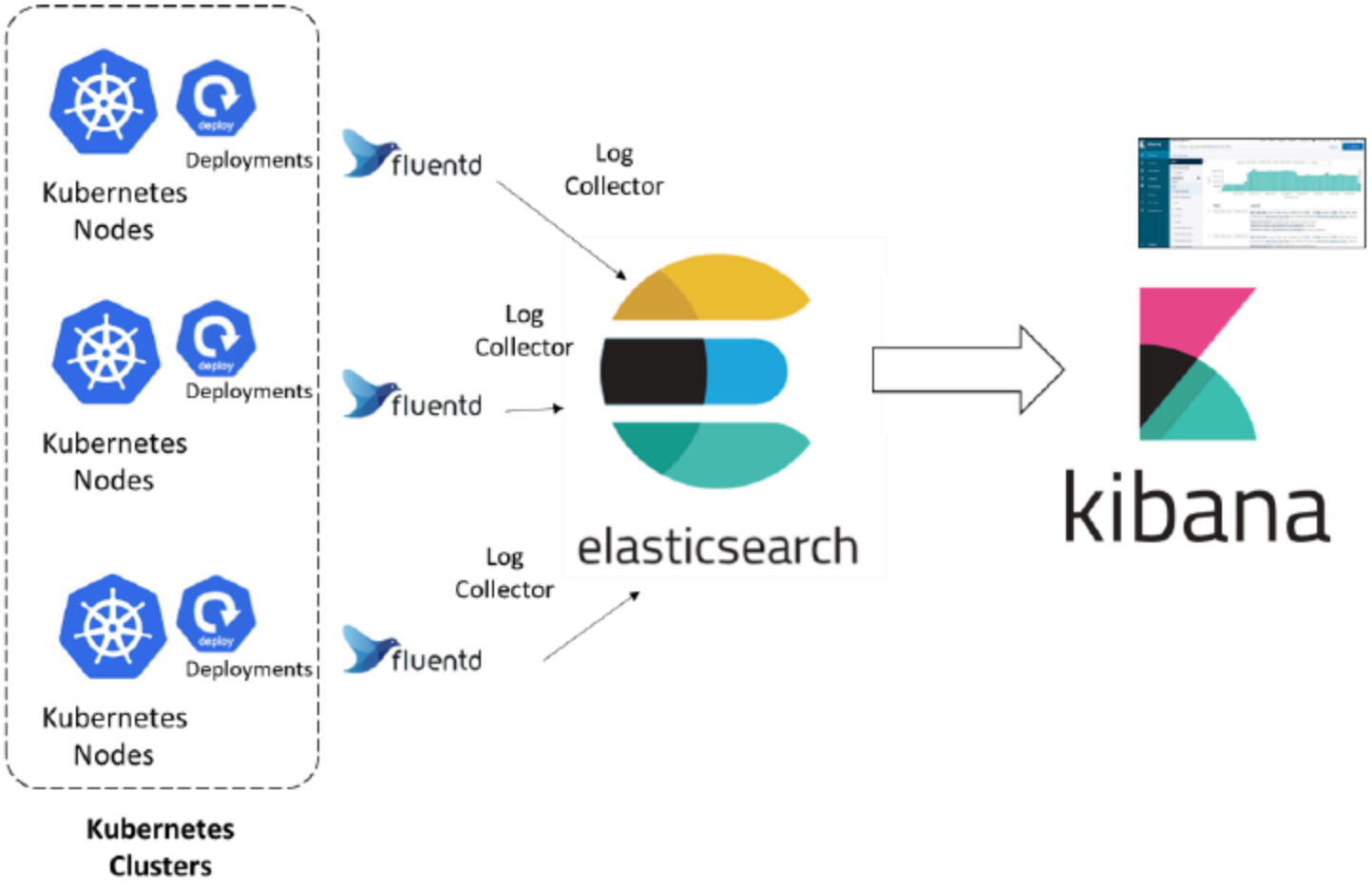

Fluentd
Node
fluentd
Node
fluentd
Node
fluentd
daemonsdet
Elasticsearch
SVC
PVC
Statefulset
Kibana
Stacked Column
Deployemnt

Kubernetes EFK Best practises

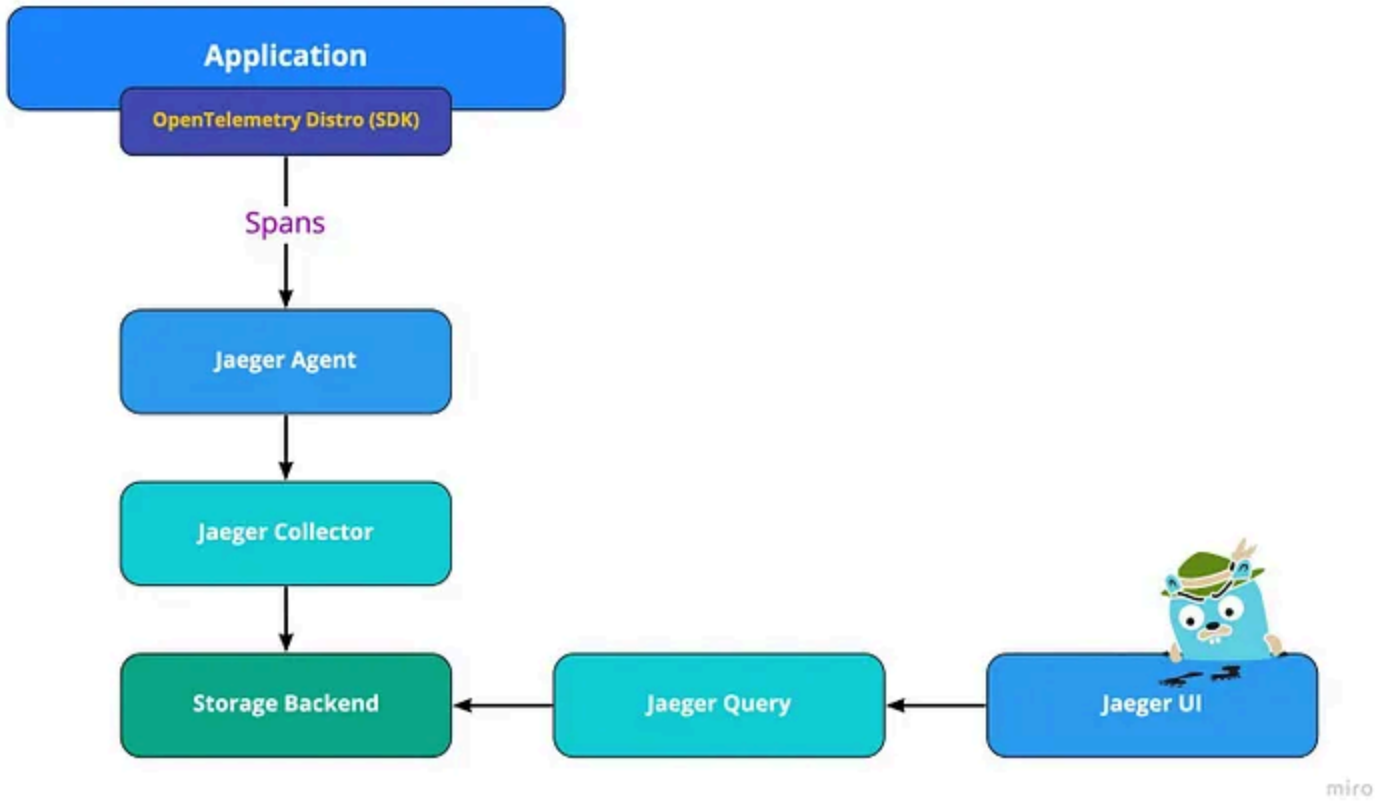
Elasticsearch

- Heap Memory Management
- Index Management
- Replication
- Data Placement
- Archiving
- Node Roles

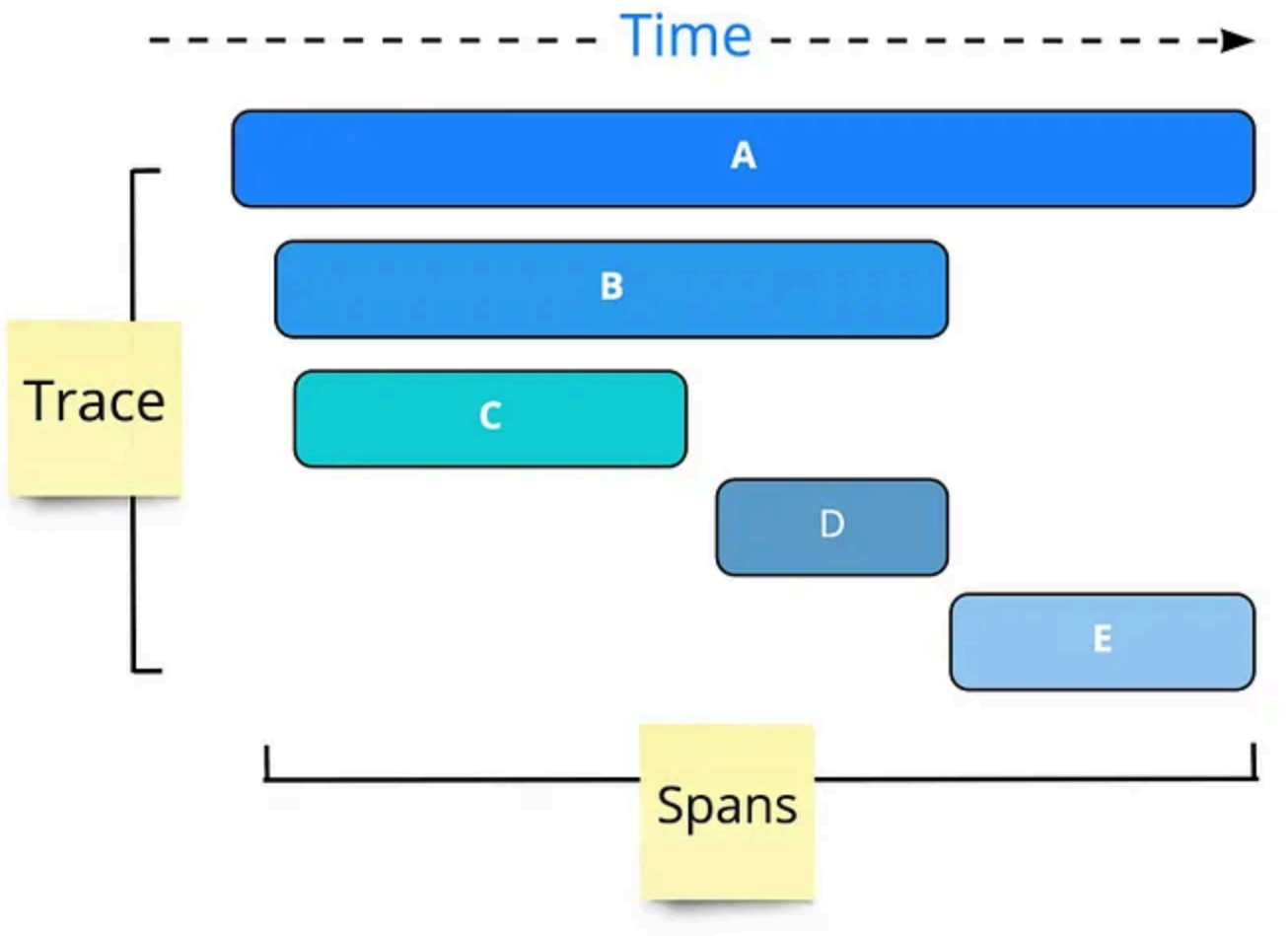
k8s EFK Stack

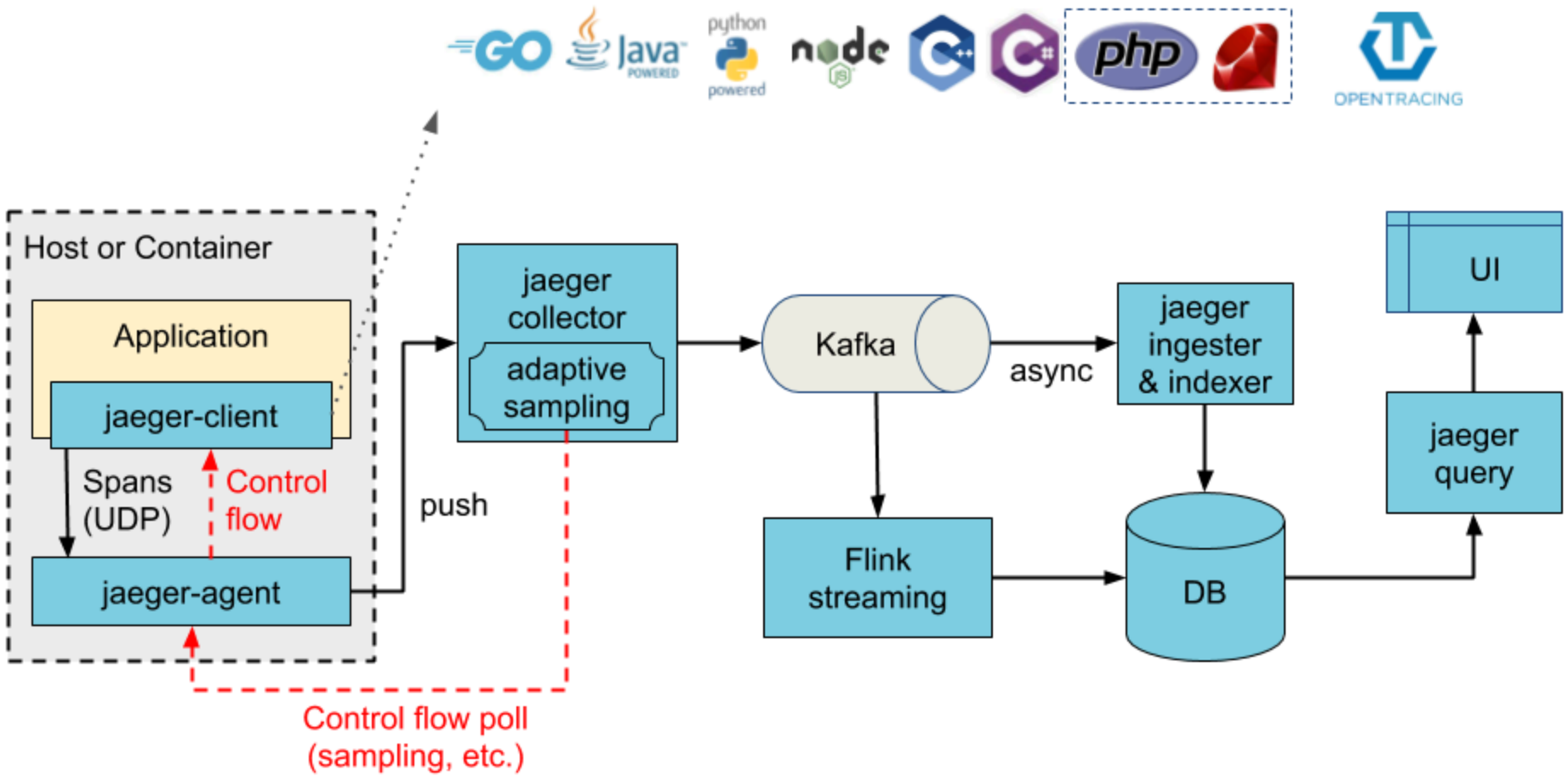


What is Jaeger Tracing?



Span and Trace




Host or Container
Application
jaeger-client
Spans
(UDP)
Control
flow
jaeger-agent
push
jaeger
collector
adaptive
sampling
Kafka
async
jaeger
ingester
& indexer
UI
jaeger
query
Flink
streaming
DB
Control flow poll
(sampling, etc.)
OPENTRACING

Jaeger Tracing Python Example

```
# jaeger_tracing.py
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.sdk.resources import SERVICE_NAME, Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import import BatchSpanProcessortrace.set_tracer_provider(
   TracerProvider(
       resource=Resource.create({SERVICE_NAME: "my-hello-service"})
   )
)jaeger_exporter = JaegerExporter(
   agent_host_name="localhost",
   agent_port=6831,
)trace.get_tracer_provider().add_span_processor(
   BatchSpanProcessor(jaeger_exporter)
)tracer = trace.get_tracer(__name__)with tracer.start_as_current_span("rootSpan"):
   with tracer.start_as_current_span("childSpan"):
           print("Hello world!")
```

Практика

LAB 9: Monitoring and Observability with Prometheus and Grafana

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_minikube_podman.md#lab-9-monitoring-and-observability-with-prometheus-and-grafana