Observability and Troubleshooting

Health and Status Checks

Logging, Auditing, and Troubleshooting

kubernetes

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

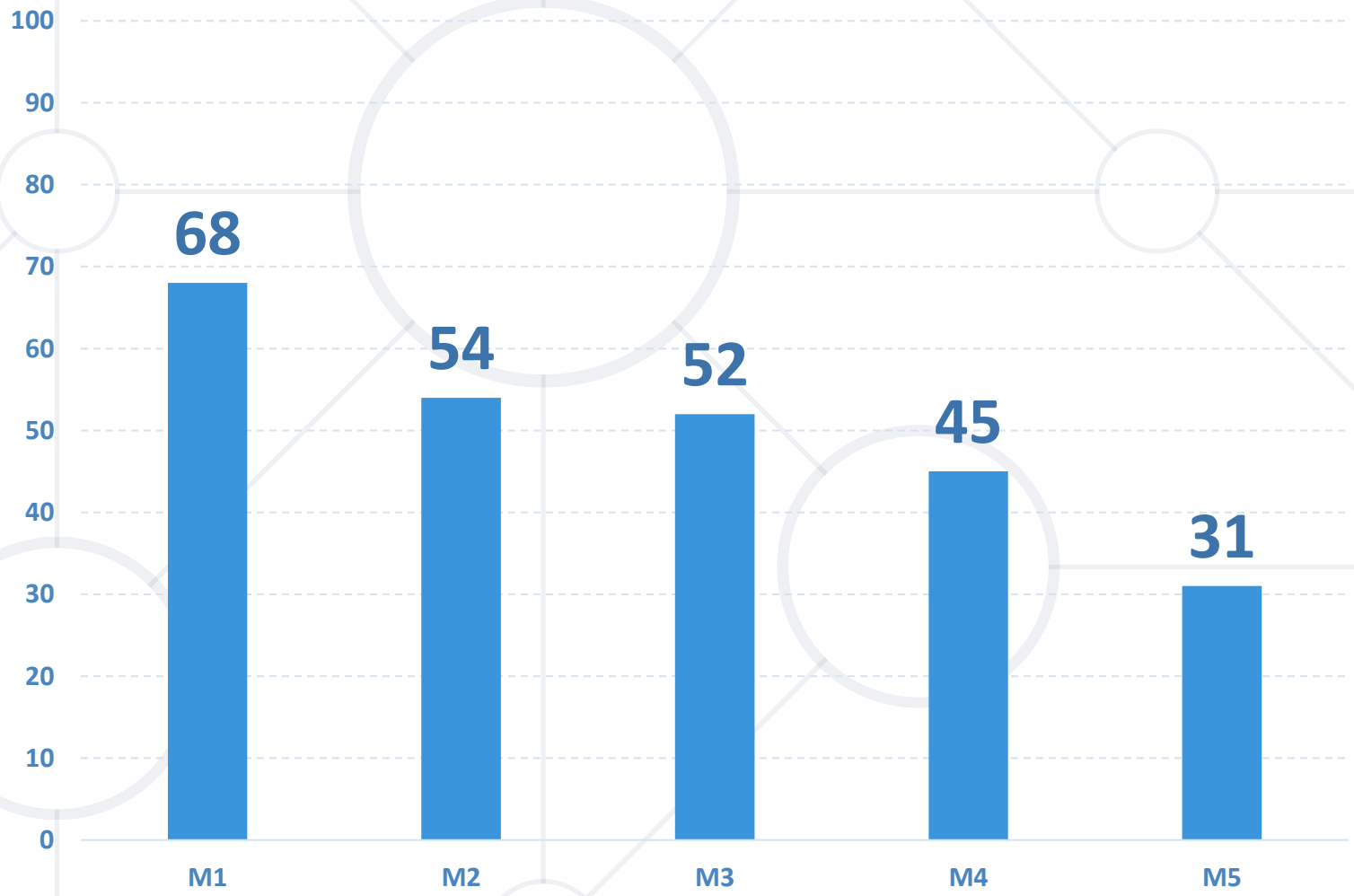
Software University

sli.do

#Kubernetes

facebook.com
/groups/kubernetesnovember2025

Homework Progress



Submit M5
until 23:59:59
on 09.12.2025

Submit M6
until 23:59:59
on 16.12.2025

Previous Module (M5)

Quick overview

Table of Contents

This Module (M6)

Table of Contents

Health and Status Checks

Health and Status Checks

Software University

- Periodic checks executed by the **kubelet** against containers
- Those checks are known as **probes**
- They can be **liveness**, **readiness**, and **startup** probes
- Their status can be either **Success**, **Failure**, or **Unknown**
- Used for a better control over the container and pod lifecycle and better integration with other objects

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Check Methods

Software University

- Each probe type can use either **Exec**, **HTTP**, or **TCP** method
- **Exec** is used to **exec a specified command** inside the container. It is considered **successful** if the **return code** is **0**
- **HTTP** makes a **GET request** against the pod's **IP address** on a **specified port** and **path**. It is considered **successful** if the **status code is between 200 and 399**
- **TCP** performs a check against the pod's **IP address** on a **specified port**. It is considered **successful** if the **port is open**

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Liveness Probes (livenessProbe)

Software University

- Indicate whether a **container is running**
- If it fails, then **kubelet** kills the container
- After that, the container is **subject** to the **restart policy**
- It can be **Always**, **OnFailure**, and **Never**. The **default** is **Always**
- The **restart policy** is defined on **pod level** and applicable to **all containers** in the pod
- If no liveness probe is provided it is considered as if it was there and the return status is **Success**

Readiness Probes (readinessProbe)

- Indicate whether a container is **ready to respond to requests**
- If it **fails**, then the **endpoints controller removes** the **pod's IP address** from the **endpoints** of **all services** that match the pod
- A pod is considered ready when all its containers are ready
- The default state, before the initial delay is Failure
- If no readiness probe is provided it is considered as if it was there and the return status is **Success**

Startup Probes (startupProbe)

Software University

- Indicate whether the application in the container is **started**
- If it fails, then **kubelet** kills the container
- After that, the container is **subject** to the **restart policy**
- All **other probes** are **disabled** if a startup probe is present **until it succeeds**
- If no startup probe is provided it is considered as if it was there and the return status is **Success**

Common Fields

- **initialDelaySeconds** sets the number of seconds to wait before a probe to be initiated. **Defaults** to **0** with **minimal value** of **0**
- **periodSeconds** sets how often (in seconds) a probe to be performed. **Defaults** to **10** with **minimal value** of **1**
- **timeoutSeconds** sets the number of seconds before a probe times out. **Defaults** to **1** with **minimal value** of **1**
- **successThreshold** sets the minimum consecutive successes for a probe to be considered successful after a failure. **Defaults** to **1** with **minimal value** of **1**
- **failureThreshold** sets the number of times for Kubernetes to try failing probe before giving up (for **liveness** – **restart**, for **readiness** – **unready**). **Defaults** to **3** with **minimal value** of **1**

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#configure-probes

Practice

Live Exercise in Class (Lab)

Auditing

Auditing

- Actions in the cluster are captured in chronological order
- They can be initiated by the users, applications, or control plane
- Answers who did what and when on what and what happened
- Audit records begin their existence in the **kube-apiserver**
- Each request on each stage of its execution generates an event
- It is pre-processed according to the policy and send to a backend
- Following stages are available **RequestReceived**, **ResponseStarted**, **ResponseComplete**, and **Panic**
- Audit logging may increase the memory consumption

https://kubernetes.io/docs/tasks/debug-application-cluster/audit/

Audit Policy

- Defines the rules about what events should be captured and what data they should include
- During processing, an event is compared against the list of rules in order
- First match sets the audit level of the event
- The available audit levels are **None**, **Metadata**, **Request**, and **RequestResponse**
- A policy to be valid, should have at least one rule

https://kubernetes.io/docs/tasks/debug-application-cluster/audit/#audit-policy

Audit Backends

- Audit backends persist audit events to an external storage
- Two backends are supported by default
- **Log backend** writes events into the filesystem
  - Writes audit events to a file in **JSONlines** format
  - Requires two volumes and volume mounts
- **Webhook backend** sends events to an **external HTTP API**
  - Both require **kube-apiserve**r flags to be configured

https://kubernetes.io/docs/tasks/debug-application-cluster/audit/#audit-backends

Logging

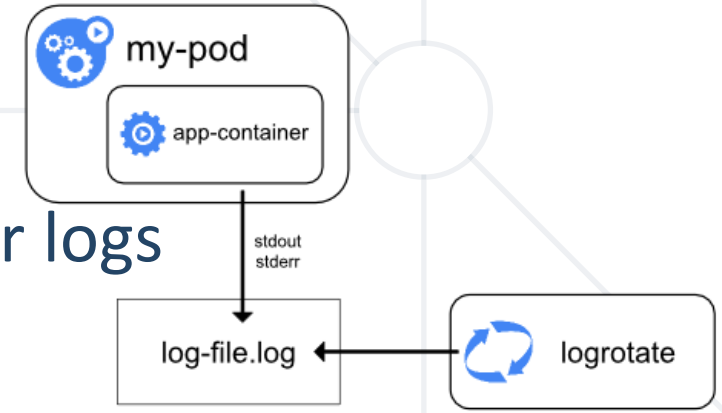
Logging Architecture

- Logs help us understand what is happening in our applications and cluster
- They are used for debugging problems and monitoring activity
- Most applications use logging either on the **stdout**/**stderr** or in a **file**
- Container engines/runtimes even though providing logging capabilities are usually not enough
- We need to access the logs even if and after a container or node crashes
- Thus, we need a **cluster-level logging** solution that will store logs elsewhere and they will have different lifecycle compared to the resources or nodes in the cluster

https://kubernetes.io/docs/concepts/cluster-administration/logging/

Node Level

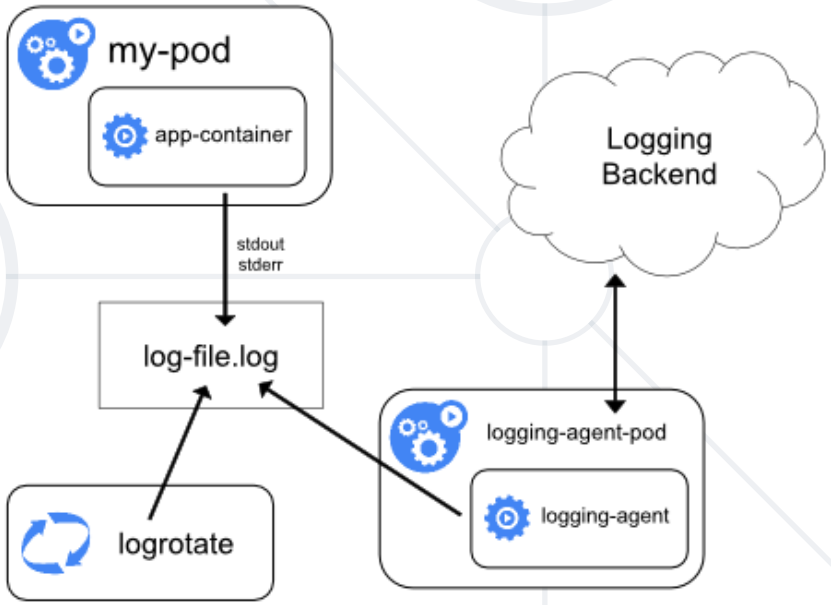
- Even though not an ideal solution it can do the job
- We should pay attention to the following:
  - When a container is restarted, the **kubelet** keeps **one terminated container with its logs**
  - If a pod is evicted, all corresponding containers and their logs are deleted
  - We should set log rotation to do some housekeeping
  - Different container runtimes may have different requirements and capabilities
  - Not all system components are the same, so do their logs
  - Service based components log via **systemd routines**
  - Container based components use files in **/var/log**



https://kubernetes.io/docs/concepts/cluster-administration/logging/#logging-at-the-node-level

Node Logging Agent

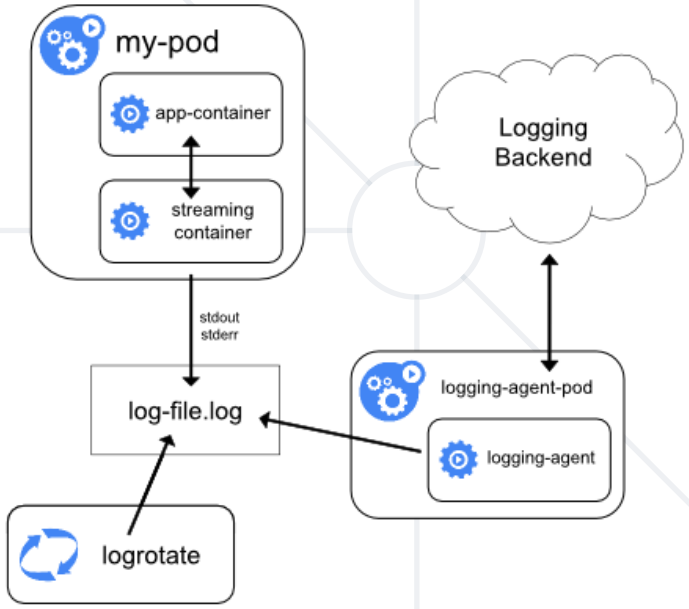
- This is considered **cluster-level** logging approach
- For this, we deploy a node logging agent on each node
- Typically, the logging agent is containerized and deployed via **DaemonSet**
- The agent exposes the logs and pushes them to a backend



https://kubernetes.io/docs/concepts/cluster-administration/logging/#using-a-node-logging-agent

Streaming Sidecar and Logging Agent

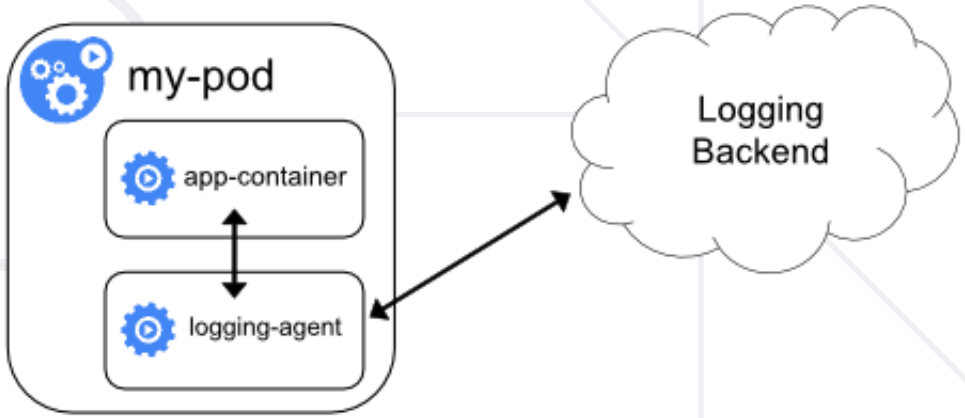
- This is considered **cluster-level** logging approach
- The sidecar container is publishing the log to its **stdout/stderr** and thus making it available for the **kubectl log** command
- Used to overcome limitations like separating multiple logs
- We can have more than one sidecar container



https://kubernetes.io/docs/concepts/cluster-administration/logging/#streaming-sidecar-container

Sidecar with Logging Agent

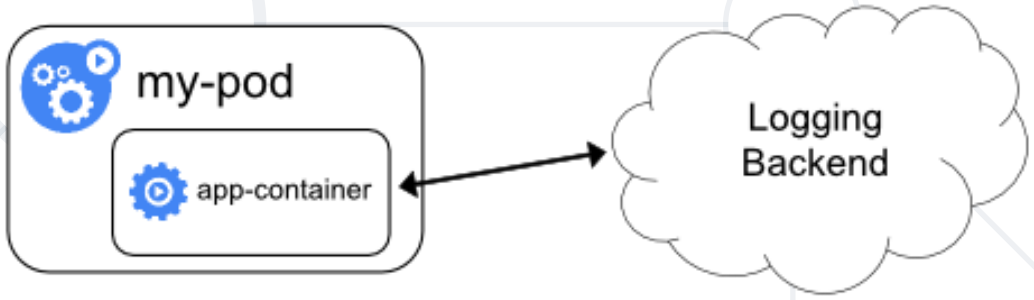
- This is considered **cluster-level** logging approach
- Used when the node-level logging agent doesn't agree well with the application
- For this, we create a sidecar container with a logging agent that is especially configured and adjusted to the application's needs
- These logs are not consumable by the **kubectl log** command



https://kubernetes.io/docs/concepts/cluster-administration/logging/#sidecar-container-with-a-logging-agent

Exposed Directly from the Application

- This is considered **cluster-level** logging approach
- Every application pushes its logs to a backend
- Simple solution which requires every application to support the common backend which may not always be feasible



https://kubernetes.io/docs/concepts/cluster-administration/logging/#exposing-logs-directly-from-the-application

Practice

Live Exercise in Class (Lab)

Troubleshooting

Troubleshooting

- Not always everything is going according to plan and things break
- The process of troubleshooting in Kubernetes is not that much different from other complex platforms
- Here, we have two distinct domains – **cluster** and **application**
- We should always check first the release notes for our version
- Once we narrowed down the cause, we should start applying the corrective measures one at a time until the issue is resolved

https://kubernetes.io/docs/tasks/debug-application-cluster/troubleshooting/

Cluster Troubleshooting *

- Common reasons include node power state, network connectivity, software version misalignment, data loss, bad configuration, etc.
- First, we should check if all nodes are there and operational
- Then, we must check the logs of the system components on the control plane and the workers
- Keep in mind that depending on how a component is deployed (native service or container) the logs may be in different places

https://kubernetes.io/docs/tasks/debug-application-cluster/troubleshooting/

Application Troubleshooting *

- First, we should define where exactly is the problem
- Is it the service
  - It is not reachable
  - It is not returning what is expected
- Or is it the workload object
  - What level and type of object
  - What is its state

https://kubernetes.io/docs/tasks/debug-application-cluster/troubleshooting/

Troubleshooting Pods

Software University

- Start with describing the pod
- Check the state of all containers inside the pod
- If it is in **pending** state, then it cannot be scheduled on a node. Usually, this is because of **lack of resources**
- If it is in **waiting** state, then it is scheduled, but still cannot run. Usually, this is because of a **wrong** or **missing image**
- It is in **running** state but doesn't behave as expected. Usually, this is because of a **manifest error**

https://kubernetes.io/docs/tasks/debug-application-cluster/troubleshooting/

Troubleshooting Services *

- Does the service exist
- Is it defined correctly
- Are there any endpoints at all and how many
- Are those pods working
- Is the service reachable by DNS name and/or IP
- Is the kube-proxy working

https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/

Practice

Live Exercise in Class (Lab)

Questions?

SoftUni

Software University | SoftUni AI | SoftUni Digital | SoftUni Creative | SoftUni Foundation | СофтУни БУДИТЕЛ | Finance Academy

SoftUni Diamond Partners

Software University

Coca-Cola HBC Bulgaria

Postbank

top EMPLOYER България Bulgaria 2025

FOR A BETTER WORLD OF WORK

SUPER HOSTING .BG

encorp.ai

createX

DRAFT KINGS

THE CROWN IS YOURS

INDEAVR

Serving the high achievers

VIVACOM

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg