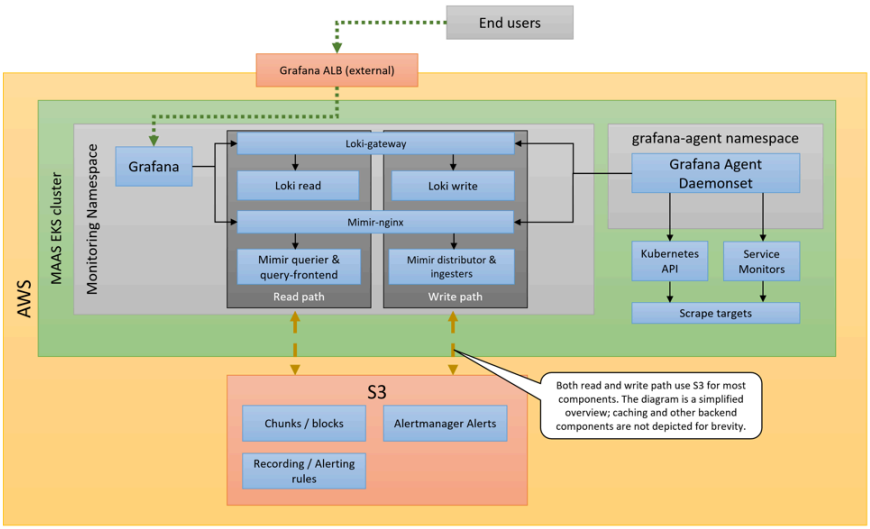
Infrastructure Monitoring

This page describes the core monitoring system and how the underlying infrastructure is monitored, most notably Mimir, Loki and the Grafana alloy for infrastructure monitoring. It assumes knowledge of these three components and does not explain it. Please see the further reading section at the bottom of this page to get more detailed information on how these systems work. For completion's sake, Mimir and Loki have similar architecture and components hence some of the terminology you will find on those pages are interchangeable.

Keep in mind that Mimir is essentially Prometheus (in fact, it uses Prometheus itself for a lot of its functionality) and it's also called as such within the company. Within the Grafana datasource configuration you also see Prometheus as a datasource name, which points to Mimir.



We are running a fairly standard setup for both Mimir and Loki, both using the defaults provided by their respective Helm charts albeit with lowered resource allocation. The defaults would basically allocate dedicated nodes for some of the components, which is not what we want nor what we need; both systems can run at quite a large scale but this is not something we are currently doing. Deployment-wise, Loki supports a mode called "Simple Scalable Mode" and this is also the mode that we run. Basically it means that the components that make up the read path are combined in a single pod, and the components that make up the write path are combined in a single pod, therefore reducing deployment complexity. At this time of writing Mimir does not support this yet but work is underway. For Mimir it is either a monolith mode (so everything into one pod) or distributed mode (which is called microservices mode in Loki). We use the latter. This also runs a couple of caching pods in the form of memcached instances as it is part of the standard deployment.

The general flow is that services (either the Grafana alloy for writes or Grafana for reads) will connect to the mimir-nginx or the loki-gateway for both. They are both nginx processes that proxy requests to the right backend component based on the URL. Both also have an ingress object attached to their Kubernetes service to create an internal ALB (note that this is not depicted in the diagram). This is required for components outside of the cluster (like the synthetics) to be able to push their data into either Mimir or Loki. As this is an internal ALB it is not reachable via the internet and only via the internal DSM network.

Another difference between Loki and Mimir is that Mimir has awareness of availability zones. It means that during an update of the Helm chart, the write path is restarted per availability zone. There is also a rollout operator pod running as a part of this. The reason

for this is two-fold; one to optimize the write path within the availability zone and also to ensure the write-path becomes unavailable during too many pods restarting. Loki does not seem to be supporting this.

> ⚠ You need at least 3 writer nodes or 3 ingesters in Mimir for the write path to function. The workers will refuse writes if this is any less than this number. This is not necessarily a problem for shorter periods of time as the Grafana alloy also buffers results if either Mimir or Loki are unavailable.

Grafana alloy

The Grafana alloy runs as a daemonset and operates in flow mode (this is different than the static mode). It's deployed via the standard Helm chart and the *monitoring* repository contains the values.yaml for each environment . It monitors the cluster nodes itself, as well as all the pods, ingress objects etc. It also relies on cadvisor and kube-state-metrics, which it expects to be present and enabled in the cluster. Kube-state-metrics is a separate Helm chart, its values.yaml can also be found in the *monitoring* repo. Cadvisor is exposed by the kubelet and requires no additional installation. The Grafana alloy also scrapes the kube-proxy to extract node statistics. As the Grafana alloy supports the Prometheus Operator CRDs, it also responds to ServiceMonitors and PodMonitors. We use ServiceMonitors as part of the Mendix deployments to enable metric collection of OpenTelemetry.

The configuration itself, being part of the values.yaml, is mostly duplicated amongst the clusters at this time of writing. This is not ideal and needs to be improved. In addition, there is filtering happening on certain metrics (via relabel actions) to drop them or rewrite them. Metrics are usually dropped because of high cardinality or because they don't add any value, or they are rewritten so label names line up with what dashboards expects them to be called. We also add a `cluster` label on most places because dashboards expect that label to be present in their queries. Unlike with synthetics, modules or remote connections are not used within this Grafana alloy setup because everything is discovered from within the Kubernetes cluster itself.

Finally, the flow-mode of the Grafana alloy takes some getting used to and has a learning curve. It's recommended that, if you need to make serious changes on it, to spend some time understanding how it works first before you make many changes to it. Make sure you use the Grafana alloy web-ui for understanding what comes in at every step and what its output is. Remember that flow mode expects types on objects and they must line up, just like with a strongly typed programming language.

> ℹ The CRDs are also found in the *monitoring* repository which you only need to apply once on a new cluster. At this time of writing applying them is not part of any automation.

Alerting

Alerting is a little finnicky as we are bound to the constraints of DSM and the possibilities that Alertmanager (part of Mimir) has. To do alerting there are basically three possible sources: Mimir itself, Loki and Grafana. Mimir and Loki both run a component called the Ruler, which evaluates both alerting rules and recording rules and registers them with their respective backend system. This uses S3 as a permanent storage mechanism (simply because that was the simplest to do in terms of configuration). Note that at this time of writing we are not using the Loki Ruler directly, because there is no need for them currently. We are using the Mimir Ruler extensively as part of the mixins (see below). Grafana also allows for creation of alerts and it can also send alerts by itself. We decided to have one consistent mechanism and have all alerts go via Alertmanager rather than having separate systems sending notifications separately. This means that Grafana is configured to send alerts via Alertmanager and not send out alerts directly by itself.

As e-mail is the preferred notification mechanism at DSM, we have two options for this; use SNS to send emails or use an DSM-approved SMTP relay host that we can use. At this time of writing we use the former and the latter is a work in progress (it requires approval and technical registration before you can use the relay host). SNS comes with limitations, most notably the subject must be less than 100 characters. We have modified the alerting notification template of Alertmanager to include the summary of the alert in the subject, rather than using the default subject (which would lead to alerts being rejected by SNS because of this 100 character limit). This should reduce the likelyhood of alerts being rejected but ideally this should be truncated. At this time of writing it is not known if this is an actual problem that needs fixing or if it's even possible to truncate.

The SNS topics themselves are created by the CDK code living in the *monitoring* repository. Along with it is a YAML configuration file found in `configuration/topics.yaml` Here you can add email addresses to topics. After running the pipeline, the user

should receive an email asking to confirm their subscription to the topic. Once confirmed, the user should start to receive emails from SNS.

⚠ At this time of writing only a single topic is used, the default one, even though multiple groups are mentioned in the topics. This is because the alertmanager configuration itself is not setup for this. This will come in the near future, so this warning may no longer be relevant by the time you read this.

Monitoring mixins & recording rules

The Mixins are an encompassing name for open source Grafana dashboards and alerting / recording rules (so they are three types of resources). They are community driven and have a certain logic to generate; it's based on jsonnet therefore the syntax is different than others. The code for this lives in the *monitoring* repository inside of the `mixins` folder. This has a little different setup than everything else as it has some custom scripts in place to streamline the generation and deployment of the generated resources. They are essentially just wrapper scripts around the various commands that need to run.

There are a few scripts in the `scripts` folder that download, generate and apply the dashboards and the recording / alerting rules. The generated dashboards and rules are stored in the `compiled` folder (not committed in the repo) and is expected to be ephemeral. There is also a `customizations` folder, which contains jsonnet files with updated code. This is sometimes necessary to match up labels in the dashboard with what we are using, for example. The `build.sh` applies these customizations before running `mimirtool` to generate the dashboards, recording rules and alerting rules. Deployment wise there are two steps. For dashboards, there is a simple helm chart used. For recording rules / alert rules there's something called `mimirtool` which calls the Ruler API and applies the recording / alerting rules. The `apply.sh` only applies the latter; the dashboards are expected to use a simple `helm install`-like command. It expects a kubernetes label to be set on the `ConfigMap` which is then picked up by one of the Grafana sidecar containers to dynamically load.

Applying the recording rules and alerting rules requires performing an HTTP POST directly on the ruler as the API endpoint is not proxied by the mimir-gateway. Currently the `apply.sh` will do a `kubectl port-forward`, which is not ideal but the most practical thing to do.

ℹ Note that there is also a `dsm` folder with JSON files, which are saved models directly from Grafana. In theory it should work but the actual end-to-end flow for anything other than jsonnet files coming out of open source projects is not tested at this time of writing.

Further reading

- Prometheus Monitoring Mixins | Monitoring Mixins
- Alertmanager | Prometheus
- Grafana Mimir architecture | Grafana Mimir documentation
- Loki overview | Grafana Loki documentation
-