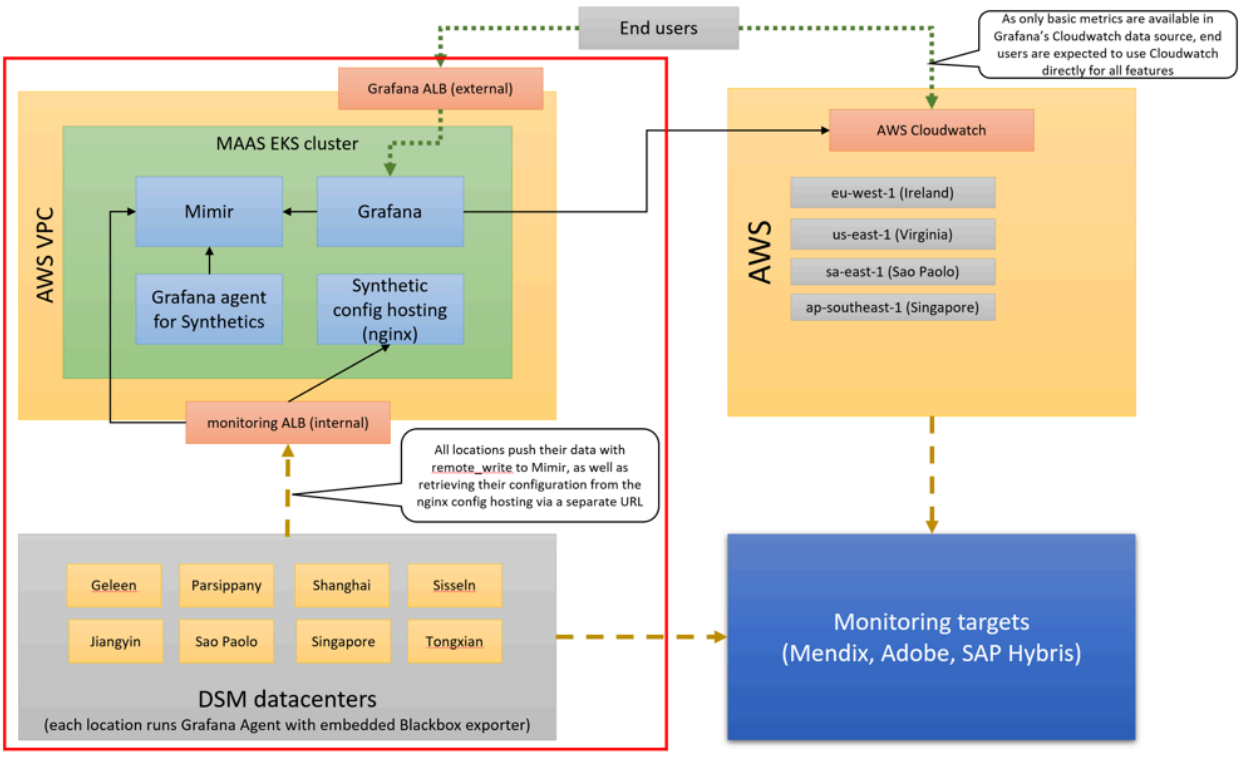
Grafana Synthetics (HTTP based)

The synthetic monitors are continuously monitoring a given set of websites in a set interval and report metrics such as page load times, time to first byte, transfer times, etc. Within the synthetics space we define two types of monitors; HTTP based and browser based. HTTP based is a simple HTTP call, usually an HTTP GET, and checks the HTTP status code along with response times. Browser based, or clickpath based as it was known in Dynatrace, utilize a headless browser to perform the same actions an actual user will be doing. It uses a Chromium-based browser and, unlike a HTTP-based monitor, also renders JavaScript and allows you to programmatically interact with the website itself. Browser based monitors are more advanced and offer a deeper level of monitoring than HTTP based monitors will be able to do. As you read in the parent page, we use CloudWatch Synthetics for browser-based monitors and Grafana alloy with the blackbox exporter plugin for the HTTP based ones. The latter also runs on DSM private locations at this time of writing, the former only runs in AWS regions. In the future we will look if we can add Grafana K6 as part of browser-based stack.

> This page focuses solely on the Grafana Synthetics portion of the stack; please review the other pages for CloudWatch Synthetics based monitoring.



Grafana alloy Synthetics overview

The Grafana alloy based monitoring is a little more complicated than the CloudWatch one because there are more moving parts involved. As we were building this we wanted to have the capability to centrally control the configuration of all remote locations without having to redeploy the alloy configuration on remote systems. This because they are primarily run from DSM's own datacenters and are considered to be private locations with restricted access. The Grafana alloy for Synthetics, as mentioned in the diagram above, also executes synthetic checks for every Ingress object in the Mendix namespace, but this is not explicitly depicted in the diagram. Note that this Grafana alloy is separate from the Grafana alloy Daemonset that performs the infrastructure monitoring of the EKS cluster itself (it also has a separate configuration). The monitoring ALB is an internal ALB (so it is not publicly accessible) which has multiple target groups connected to it; one is Mimir and the other one is the configuration hosting used by the remote locations in the DSM datacenters. It's not used by the in-cluster Grafana alloy as that dynamically discovers its configuration via the Kubernetes API.

The config hosting is nothing but an nginx that serves up JSON files that contain the configuration for each location. The intention in the future is to replace the static hosting with a REST API of some sorts. The reason this is a static nginx and not an S3 bucket is because of lack of functionality in the Grafana alloy and the types system that Flow mode introduces. Once it can dynamically reload configuration with something like `discovery.s3` (this does not exist at this time of writing) then this can go onto an S3 bucket and not having to run an additional pod. The config hosting has a small Helm chart that is part of the pipeline (all it does is create configmaps out of each JSON file). Another thing to point out is that we have a blackbox module within the Grafana alloy configuration, which serves as a wrapper to make dynamic discovery out of a file possible with the embedded blackbox exporter. You can consider this as a generic configuration for the blackbox exporter applicable for all locations. Its configuration is also dynamically retrieved from the config hosting so you can modify it when needed.

Note that the remote locations of DSM are **Windows Server** based. Any logging will be appearing in the Windows Event Viewer.

Private location configuration

In order to understand the exact flow, let's dissect the various configurations that there are for this. Within the *monitoring* repository, there is a path `helm/grafana-alloy/synthetics`. This contains all of the configuration specifically for Grafana alloy based synthetics. In this folder there are three subfolders:

- `config-hosting`, which refers to the Helm chart of the nginx-based config hosting as mentioned above.
- `in-cluster`, which refers to the Grafana alloy for Synthetics and runs with its own configuration for Kubernetes API based discovery.
- `remote-locations`, which contains the Grafana alloy configuration for the DSM datacenters, including their JSON file.

First of all, you don't typically change the various `config.river` files found in the `remote-locations` folder, you only change the JSON file or the Blackbox module if you want to make generic changes. If you do have to make changes to the `config.river`, you will need to reach out to the monitoring team to deploy these changes on the remote servers; there is no other way to deploy these changes at this time of writing. Therefore it is advised to work around that and put any changes in the JSON file and / or the blackbox exporter module if you can; that one is dynamically read and updated.

The JSON file that contains the targets has a generic format like this. Note you should consider anything between brackets as a choice and not a literal string. So for example, the value for the project key must be either mendix, adobe, or sap-hybris.

```
[
    {
        "labels": {
            "module": "http_2xx",
            "project": "<mendix/adobe/sap-hybris>",
            "app_type": "<critica/major/minor>"
        },
        "targets": [
            <list-of-targets-without-http-or-https-prefix>
        ]
    },
    {
        "labels": {
            "module": "http_2xx",
            "project": "<mendix/adobe/sap-hybris>",
            "app_type": "<critica/major/minor>"
        },
        "targets": [
```

```
            <another-list-of-targets-without-http-or-https-prefix-with-different-labels>
        ]
    }
]
```

Basically you create several sections for several targets with different labels. This is how you can identify Mendix applications or Adobe applications, for example.

> ⚠ Make sure you do **not** add a URI scheme like http:// or https:// to the list of targets, because this will fail. The blackbox module will always set this to https at the right time.

End-to-end flow for a synthetic monitor

To describe the end-to-end flow, let's take the Geleen remote location configuration as an example and the same applies for each location. I will selectively put snippets of configuration in here and describe its function.

```
discovery.http "config" {
  url = "https://synthetics-config.mx.dsm.app/geleen.json"
}
discovery.relabel "config" {
  targets = discovery.http.config.targets

  rule {
    source_labels = ["module"]
    target_label = "__param_module"
  }

  rule {
    source_labels = ["__param_target"]
    target_label = "target"
  }
}
```

First of all, this is the JSON file from which the Grafana alloy discovers targets. This URL points to the static nginx hosting and uses the format mentioned above. This discovered list of targets is then, after some relabeling, passed onto the blackbox exporter module like so:

```
remote.http "blackbox_config" {
  url = "https://synthetics-config.mx.dsm.app/blackbox_exporter.yaml"
}

module.http "blackbox_exporter" {
  url = "https://synthetics-config.mx.dsm.app/blackbox.river"
  arguments {
    config = remote.http.blackbox_config.content
    targets = discovery.relabel.config.output
    location = "geleen"
  }
}
```

We retrieve the blackbox exporter config from the static config hosting, as well as the module code itself. We then pass along the discovered targets to the module and set the location for this alloy, which becomes a label in the final dataset. The blackbox module converts the discovered targets as a valid target to pass onto the embedded blackbox exporter, which has a specific format that it must be executed on. As this is a so-called *multi-target exporter* in Prometheus terminology, the basic configuration for the blackbox exporter is the same but the arguments are different each time (hence the target terminology). Otherwise you'd have to fully write out an entire valid Prometheus scrape config (which is what the Grafana alloy uses internally) for each URL. A multi-target exporter avoids this by being more dynamic in nature.

```
prometheus.scrape "synthetics_config" {
  targets = module.http.blackbox_exporter.exports.targets
  forward_to = [prometheus.remote_write.mimir.receiver]
  scrape_interval = "5m"
}

prometheus.remote_write "mimir" {
  endpoint {
    url = "https://mimir.mx.dsm.app/api/v1/push"
  }
}
```

Finally, we instruct the Grafana alloy to scrape the targets as exposed the blackbox module every 5 minutes, and forward the results to the Mimir endpoint that accepts remote writes. This will then be ingested by Mimir and stored accordingly.

> Both the discovery.http and remote.http periodically poll for changes, so you don't need to restart anything for config changes to be taken into effect. If nothing happens you may have a configuration error, in which case you must check out the event viewer logs on the remote location.

Blackbox exporter modules

One of the things you will see in the JSON file format is that there is a `"module": "http_2xx"` - at this time of writing we only have a single blackbox exporter module called http_2xx. This simulates an HTTP GET. You can create more advanced modules that do POST with form data or even retrieve multiple URLs. This, of course, has the inherit limitation that you must craft an exact POST response as the backend would expect it. In JavaScript heavy applications this could be challenging. At this time of writing we are also not using any other module other than http_2xx, but if you'd want you can add additional modules

and their parameters to `helm/grafana-alloy/synthetics/config-hosting/files/blackbox_exporter.yaml`. You can then reference this modules in the JSON file and the Grafana alloy will start executing the check using this module then.

> ⚠ A Blackbox exporter module is something else than the Blackbox module used in the Grafana alloy. The latter is the modules concept coming from the Grafana alloy, the former is conceptually coming out of the Blackbox Exporter, so they operate on different levels. Make sure you understand this difference!

In-cluster configuration

The in-cluster configuration is now considered to be exactly the same as a remote location, as mentioned above. The next section is no longer applicable but kept for future reference.

The in-cluster configuration applies the same concepts as the remote-location configuration except that all configuration is coming from a single Configmap rather than from the static nginx hosting. It also relies on discovery via the Kubernetes api using `discovery.kubernetes` rather than `discovery.http` One thing that is kind of hacky and should be replaced, once the old to new cluster migration is complete, is setting the `app_type` label to determine their criticality. At this time of writing this is more or less explicitly hardcoded in the Grafana alloy configuration like so:

```
1     // The app_type label should come out of Kubernetes directly (by setting a Kubernetes label on the deployment object),
2     // but modifying the old pipelines to support this would be too time consuming. Once the migration is complete, move
3     // this logic to the helm chart instead.
4     rule {
5       source_labels = ["__address__"]
6       regex         = "(airfreight|allocation|animaltrials|bls-assesment|budelpack|complaintmgmt|customermdm|dfs-globalqc|dici-practices|dnpblockedstock|fss-icservices|hof-ic|hrss-vacation|mdm-vendor|phagebook|sap-maint-plan|smartaudit|strinsights|trust-it).+"
7       target_label  = "app_type"
8       replacement   = "critical"
9     }
10    rule {
11      source_labels = ["app_type"]
12      regex         = ""
13      target_label  = "app_type"
14      replacement   = "minor"
15    }
```

Once the migration is complete it is recommended to set this via the Helm chart in the *kubernetes-mendix-deployment* repository and remove this hardcoded reference. Other than that it uses similar configuration concepts like the blackbox module and the configuration is similar to how it is for the remote configurations.

Further reading

- Understanding and using the multi-target exporter pattern | Prometheus
- Grafana Alloy documentation
- Components reference | Grafana alloy documentation
- prometheus/blackbox_exporter: Blackbox prober exporter (github.com) (note the Grafana alloy runs an embedded version of this)