Large-Scale Log Management Deployment

Prepared by: Juraj Kosik, Infrastructure Security Technical Lead, DT Pan-Net

13 June 2018

SUMMARY

Objective

Document describes hands-on journey of building large-scale log management system in the cloud environment.

Goals

Goal is to share experience in DevOps project based on Open Source technologies. Document is about to provide insights into real world problems when implementing and operating log management platform in large-scale touching also some additional technologies as containers, message streaming platform, own development and guidance how to put it all together in an efficient manner.

Environment

Described use case refers to the deployment of log management system in pan-European company with capability to consume load from multiple data centres exceeding 1 TB / day.

INTRO

Starting the Project…

Log management & SIEM systems have a reputation of mandatory element in every organisation with problematic business case and high complexity introduced by a need to be always ready to consume unpredictable amount of data and evaluate it in real-time in a meaningful manner.
By following this goal we had to fulfil several additional conditions due to nature of the target environment – large-scale cloud company operating in DevOps mode using Open Source platforms.

Selecting Deployment Approach

Before selecting technology it is important to define development and operational model (DOs and DON'Ts). Changing such fundamental base attributes later on might lead to complete reshaping of the platform and processes around.

We decided for the following in the very beginning:

- All components to be virtualized or containerized in the Openstack cloud
- Open Source used as much as possible
- DevOps
- CICD
- Infrastructure as a Code (IaaC)

Above-mentioned principles are widely used within the company and necessary toolset was already at our disposal, e.g. Gitlab and Gitlab Runners for running CICD jobs and wide adoption of Ansible and Openstack Heat.

Infrastructure as a Code approach is always recommended when you expect your environment to be dynamic, i.e. you plan repetitive re-deployments and migrations of your platform. In fact, being in multi data centre cloud there is no other option.
Every change of the system should be defined as a code avoiding manual ssh sessions to any components. IaaC results into clean and traceable system maintenance, easy collaboration and supports CICD.

For creating virtual resources there are several options we have been considering – Openstack Heat, Ansible, Python modules for Openstack, Terraform. We use Openstack Heat since it is a native component of the underlying Cloud OS. However also other options may fit as well.
For platform configuration we use exclusively Ansible reaching one-click deployment.

DESIGN

Selecting Technology

Open Source community is providing various options for Log management & SIEM platforms. Frequently used options are ELK Stack and Graylog. There are also several other nice and powerful alternatives. All the options have certain pros and cons. But decision has to be made in one point in time. We have selected **Graylog**.

Graylog offers multiple installation methods with descriptive and well organized documentation. Installation is possible as:

- Virtual Machine Appliance
- Operating System Packages
  - Ubuntu installation
  - Debian installation
  - CentOS installation
  - SLES installation
- Chef, Puppet, Ansible
- Docker
- Vagrant
- OpenStack
- Amazon Web Services
- Microsoft Windows
- Manual Setup

Openstack and Ansible installation methods looked very appealing for our use-case. We decided to utilize available Openstack images for Graylog. However we developed Ansible playbooks and Ansible roles from scratch to fit to our setup, to have better flexibility later on and to really understand what is going on under the hood.

We have been considering also Docker installation option. There were doubts about benefits, since in our scale the container would fill in the whole VM (container host) with a single Graylog node. It means that scaling of containerized Graylog or Elasticsearch node would need scaling of the container infrastructure first. We might get benefits out of container orchestration platform, e.g. Docker Swarm or Kubernetes in terms of scaling and auto-healing. However our experience with Elasticsearch is that when you need to restore corrupted node you already have a problem with data indexing or other internal issue which would be needed to fix apart of node recovery anyhow. Reallocating tens or hundreds of shards can be a real issue. For smaller installation Docker might fit very well though. We are operating container orchestration platform and we already use this layer for smaller components as load balancers, application firewalls or performance monitoring systems and we recognized its indisputable powers. You should definitely not ignore this alternative.

Architecture

Prior sizing the VMs properly lets elaborate more on Graylog components. The whole functionality consists of several services:

- Graylog application
- Nginx web server for UI
- Etcd – service discovery for cluster setup
- MongoDB – configuration DB

- Elasticsearch – data storage for logs

All components can reside on the same machine or can be split to separate VMs. Our approach was to keep Elasticsearch on dedicated “Data Nodes” due to its specific role and different usage of virtual resources as external Cinder volumes. Everything rest was deployed on “Application Nodes”. It could be possible to push also MongoDB to separate nodes, however benefits are questionable and always at the expense of design simplicity. We never identified to be a problem to have MongoDB and Graylog application on the same nodes.

Platform Sizing

In the beginning it is difficult to estimate the future load and proper platform sizing can be difficult. Nevertheless the most efficient way is empiric experience. There are too many variables in the matrix to find one calculation to fit all. Most relevant factors which can distort any rigid formula are as follows:

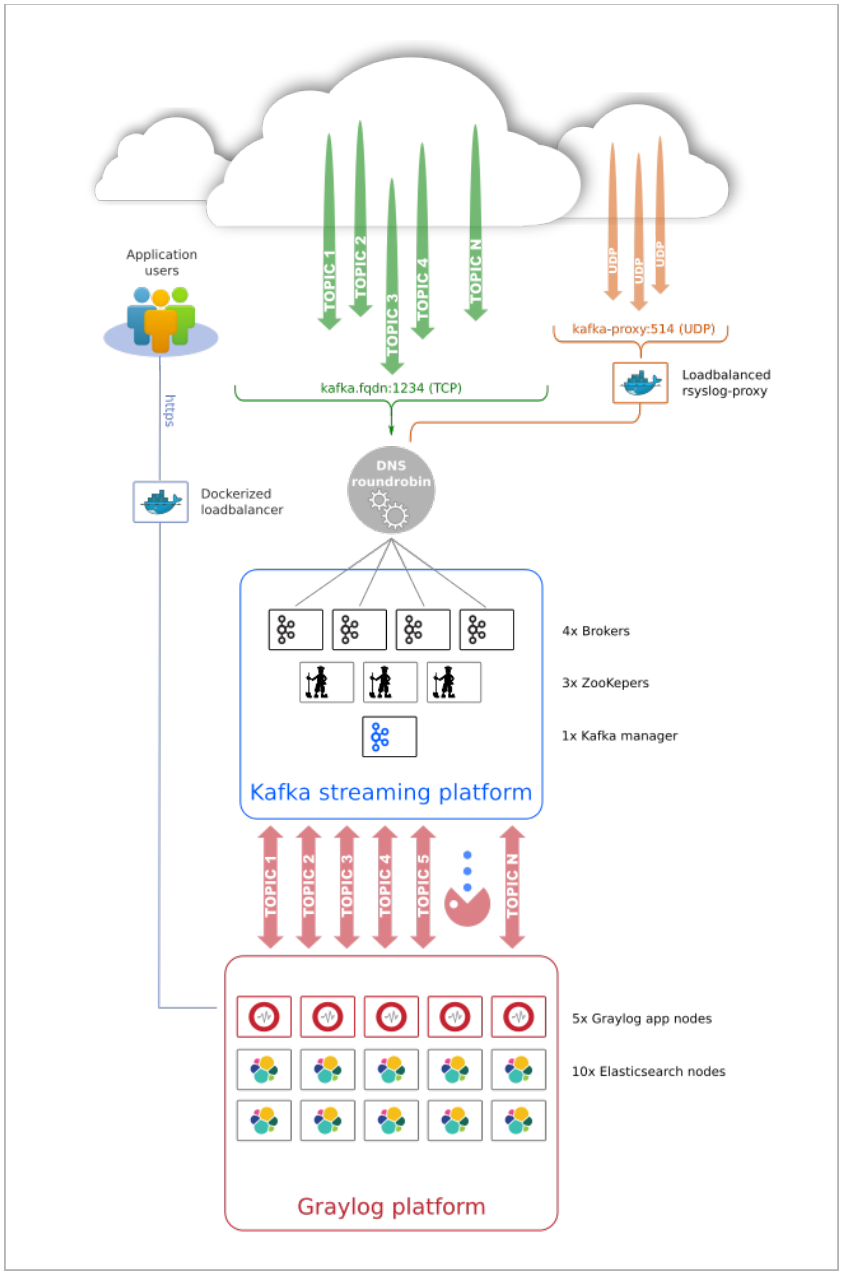
- Virtualization
- Hypervisor sizing
- Type of backend storage
- The amount of logic implemented in Graylog – extractors, pipelines, external plugins…
- Incoming data load
- Splitting the Graylog components on separate nodes

Do not be worried to resize the platform later on if you find a need for that. We went from a single-node through 1+3, 3+3 and 5+5 node cluster to final setup of 5+10 node cluster (5 Application Nodes, 10 Data Nodes). Provisioning automation using Infrastructure as a Code makes it very easy to redeploy all components. We built also playbooks for adding/removing application and data nodes on ad hoc basis. As the data load went up we resisted temptation to include more and more nodes. On contrary, we decided to increase processing efficiency a) by using log aggregation layer in front of the Graylog, b) via optimization of Graylog extractors (parsers), c) using smart integrations with external data sources, etc.
Our empiric experience led us to the following sizing:

	Application Node	Data Node
vCPU	16x	8x
vRAM	32 GB	32 GB
vHDD	100 GB	100 GB
External Volume	-	1 TB

Several dependencies has to be taken into consideration. Elasticsearch Index size should not exceed half of vRAM size. Mind this recommendation when doing your math. Also mind hypervisor sizing when planning the resources. Too much of vCPUs or vRAM might simply not fit or could even slow down the operation. In our setup we could not afford to allocate more than 32 GB of vRAM per VM, that is why we had to stick to maximum of 16 GB of Elasticsearch Index size and rather customize Index count, Shard count and Shard replica count to optimize performance. We tried having 64GB of vRAM per Data Node, but Openstack/ hypervisors had several issues with such high sizing and scheduling.

As the traffic grows find your own scale-out limit and if the platform is still not performing well consider implementing log preprocessing layer. Log preprocessing layer acts as a shock absorber protecting the Graylog application against overloading. This functionality can be delivered by various pub/sub (publish/subscribe) message brokers – we selected Kafka for that functionality. The overall architecture looks as follows. Details and lessons learned to be explained in the next chapters:



LESSONS LEARNED

Issue #1 CICD

CICD process requires multiple environments to be in place to allow Development (DEV), execution of Staging tests (STAGE) and customer-facing Production (PROD). Environments represent Graylog clusters in dedicated Openstack tenants, sometimes event in different data centres.

When speaking “Development” (executed in DEV environment), we mean:

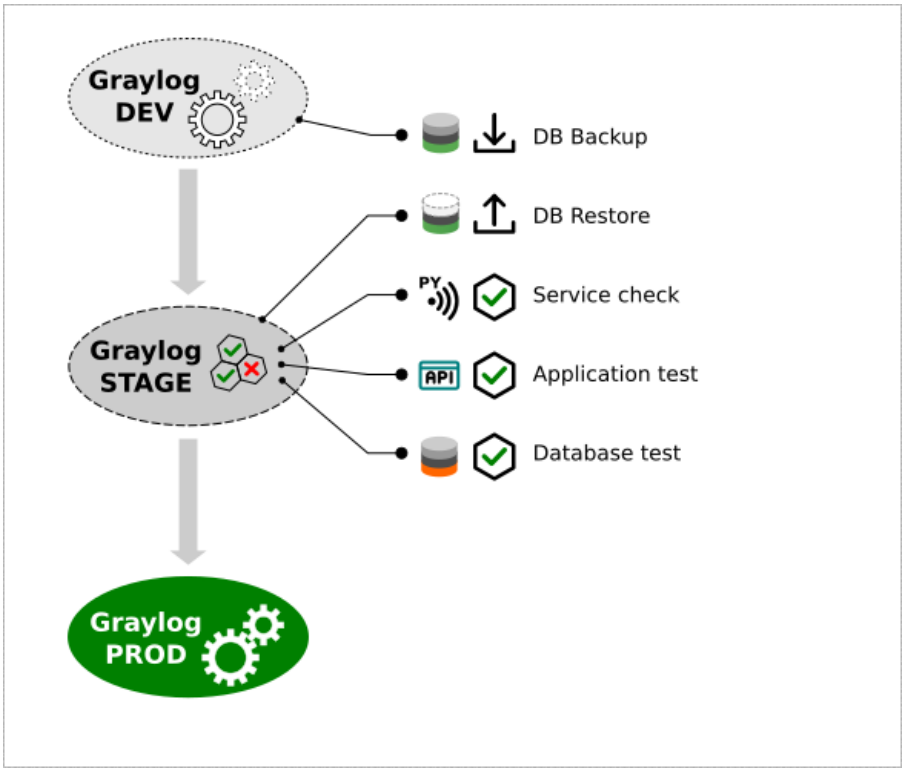
- building the internal SIEM logic inside Graylog application,
- introducing new plugins,
- building log parsers,
- data processing pipelines,
- managing alerting and notifications,
- including external data sources for log correlations,
- etc.

We follow the rule to do all admin-level changes on DEV environment only and propagate changes to PROD via CICD cycles to keep consistency within all environments.

DEV
Changes from DEV will be propagated through STAGE to PROD. i.e. all environments should be eventually equal from configuration perspective in certain points in time. Different sizing is allowed and makes sense in larger deployments to save on resources. Our DEV environment is just a singlenode VM with several adjustment comparing to PROD as retention policies or Index size. All adjusted parameters are automatically changed back during CICD run to fit to PROD needs. These changes can be executed directly in MongoDB, or using Ansible module for MongoDB management or using Ansible raw commands against MongoDB via CLI MongoDB client on Graylog nodes. Idempotency of MongoDB changes can be in some cases reached also without Ansible. In most of the cases repetitive MongoDB CLI calls do not append anything to the configuration which would violate idempotency - you can change retention policy n-times and it will just stay as originally.

STAGE
Staging environment is used for running staging tests to verify that newly introduced DEV changes will not have negative impact on PROD environment. It happened to us that misconfigurations were not detected in DEV due to limited amount of incoming logs. However PROD environment consumes real production load and misconfigurations may pop up unexpectedly.

PROD
PROD environment is basically a replica of STAGE with enabled user access. Changes are pushed from STAGE to PROD after all staging tests pass.



Implemented CICD helps also to keep up with Graylog release cycles in an easy way to avoid surprises when upgrading across multiple versions. Propagation of changes from DEV to STAGE to PROD is implemented within Gitlab Pipelines and run via Gitlab Runners.

Issue #2 Data Replication across Environments

For building a SIEM logic (e.g. aggregation rules, alerting, parsing) in DEV it is needed to have the real data (logs) in DEV as well. This is a challenge when you are using standard UDP or TCP Inputs for log reception - push method. Logs use to arrive to the load balancers in front of PROD cluster and subsequently to defined backend Graylog cluster. Sending the same data to multiple backends (e.g. DEV and PROD) is not a standard task of a load balancer.
Load balancers in front of the infrastructure can still act as a point where we can replicate and fork incoming logs to as many platforms as we need. We utilized this option using home-made scripts deployed on load balancers. However it was rather tweaking than final solution. Logs coming to the load balancers were natively delivered to the configured backend nodes (PROD cluster). Besides load balancing daemon there was an additional service running tcpdump and subsequently udpreplay to e.g. DEV environment.

Forcing log producers to replicate data on their side and to send them to multiple clusters is a customer unfriendly no-go.

Graylog allows sending data to other Graylog instances using Outputs - log forwarding. This approach is not recommended for log replication among Graylog clusters due to:

- additional load on the sending cluster,

- many complications when having various Inputs methods/types. Output methods are limited comparing to Input methods thus unwanted data transformations would be needed on receiving cluster. In such situation we will have different Inputs across environments.
- loosing equal configuration on all environments.

Note: Load balancers can also be used as point where you can implement ingress throttling when your infrastructure is overloaded. Iptables can suit this purpose very well and allows selective packet dropping based on custom statistic rules. This approach is literally dropping the traffic and should be used only when no other option is in place:

```
# To enable throttling on LB (drops 50% of the traffic)
$ iptables -A INPUT -p udp -m statistic --mode random --probability 0.5 -j DROP

# To disable throttling on LB
$ iptables -D INPUT -p udp -m statistic --mode random --probability 0.5 -j DROP

# To cut specific data on LB
$ iptables -A INPUT -p udp  --dport 5002 -j DROP
```

Our final design contains more intelligent log replication using message broker Apache Kafka (more on the topic later on). Kafka message broker is pull-based and enables arbitrary amount of environments (consumers) to pull simultaneously, even with different pace by using quotas and managing separate offsets per consumer.

Issue #3 Propagating Changes from DEV to PROD

CICD migrates changes from DEV environment to PROD. Changes introduced in DEV should be ideally defined “as a code”, e.g. introducing all changes via repeatable API calls. Even though Graylog has very nice API support which we use for various purposes, not everything is possible to do API-way. Sometimes it is really needed to use GUI to introduce change using visual support, e.g. creating a dashboard or parser.
We propagate changes from DEV in multiple steps:

- Backup MongoDB on DEV and restore on STAGE. MongoDB contains the whole configuration of the Graylog application.
- Replicate non-MongoDB changes, mainly OS level changes, e.g. installation of new plugins or setting various application parameters. This part is done exclusively “as a code”.
- Final reset of various attributes due to different sizing of source and target environment, e.g. modifying amount of Elasticsearch Shards or retention policies.
- After successful restoration in STAGE, we execute staging tests and repeat the workflow against PROD environment.

All the steps are defined as a cicd deployment code run via Gitlab pipeline. Code is mainly Ansible and Python-based.

MongoDB should be set on multi-node clusters as a Replica Set based on best practise. Backup of MongoDB can be executed on any MongoDB Replica member. MongoDB restore must be done on MongoDB primary node within the Replica set. When doing backup/restores, always check MongoDB Replica set setup to avoid problems. This check can be executed easily on the target node using MongoDB CLI tools. Check can be run via Ansible module for MongoDB or raw Ansible command using mongo binaries on the target system.

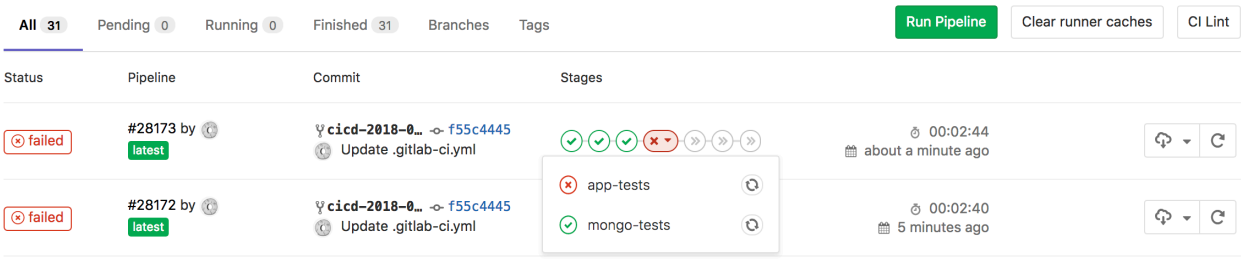
In ideal world we would need more than three (DEV, STAGE, PROD) environments. We need five environments - SANDBOX, DEV, STAGE, PROD-A, PROD-B.
SANDBOX for raw testing, and PROD-A and PROD-B for blue-green deployment. It is necessary to always be aware of which PROD cluster is at the moment active and which is idle. Otherwise you may push changes to the active PROD by accident affecting users connected to the Graylog at that moment. We rather need to propagate changes to the idle PROD and then do - what we call - “blue-green switch”, i.e. switching active and idle PROD clusters between each other so that new users will seamlessly access the new cluster without outage after next http request. There is no problem with user still connected to the PROD branch with became idle, since the same data are on both PROD branches due to usage of message broker replicating data in front of the Graylog.

For troubleshooting we are using Slack bots to verify which PROD branch is active and which is idle. ChatOps approach is used also for other L1 and L2 task, e.g. checking actual load on the nodes, restarting the nodes and so on.

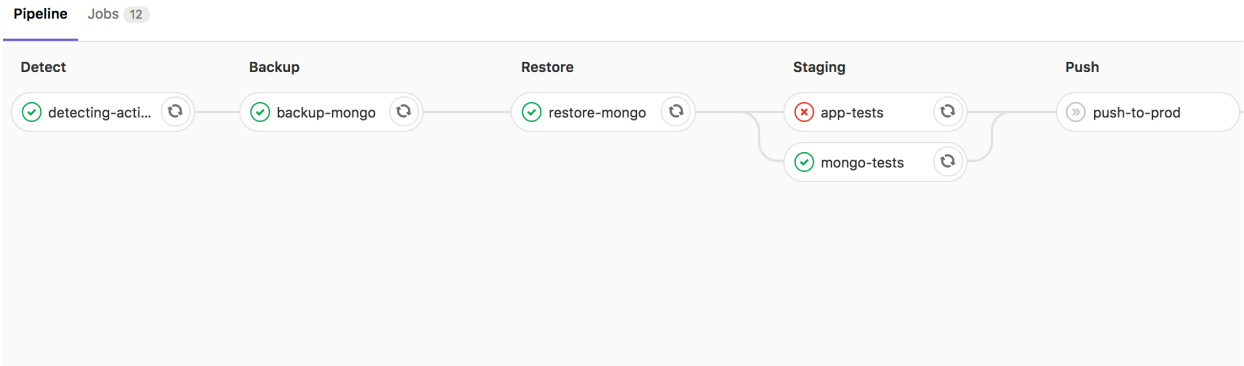
Propagation of changes from DEV to PROD can be sumarized as following Gitlab pipeline stages:

- Detect - detect which PROD (A/B) is active to propagate changes to idle PROD only
- Backup - back up MongoDB on DEV
- Restore - restore MongoDB on STAGE and replicate non-MongoDB (OS level) changes on STAGE
- Staging - execute staging tests
- Push - push changes to idle PROD
- Switch - switch idle PROD cluster to active PROD cluster and vice versa
- E2E Tests - final tests

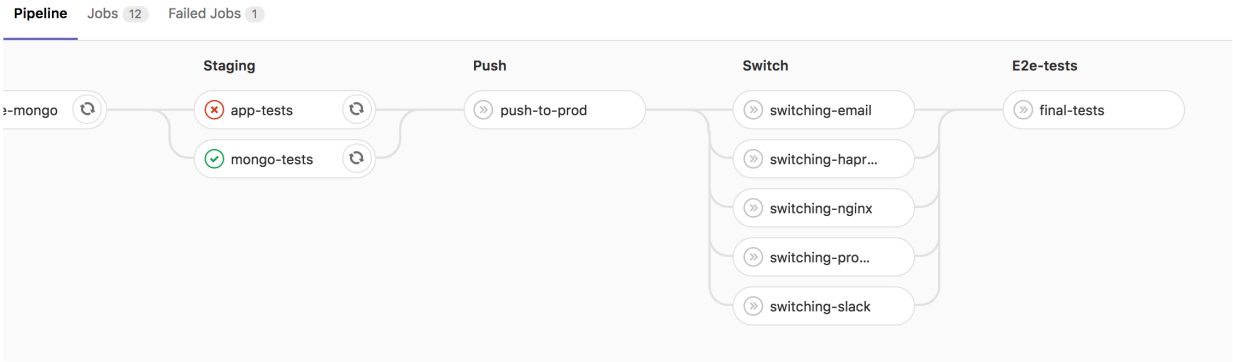
All pipeline stages run sequentially. The next stage runs only when the previous one pass. Each stage can have 1-N jobs which run in parallel by default. In case you can run certain tasks in parallel and there are no dependencies, just run them within the same stage - mostly suitable for parallel staging tests.
Example below shows failure on a specific staging test:



Pipeline stages at Gitlab UI:



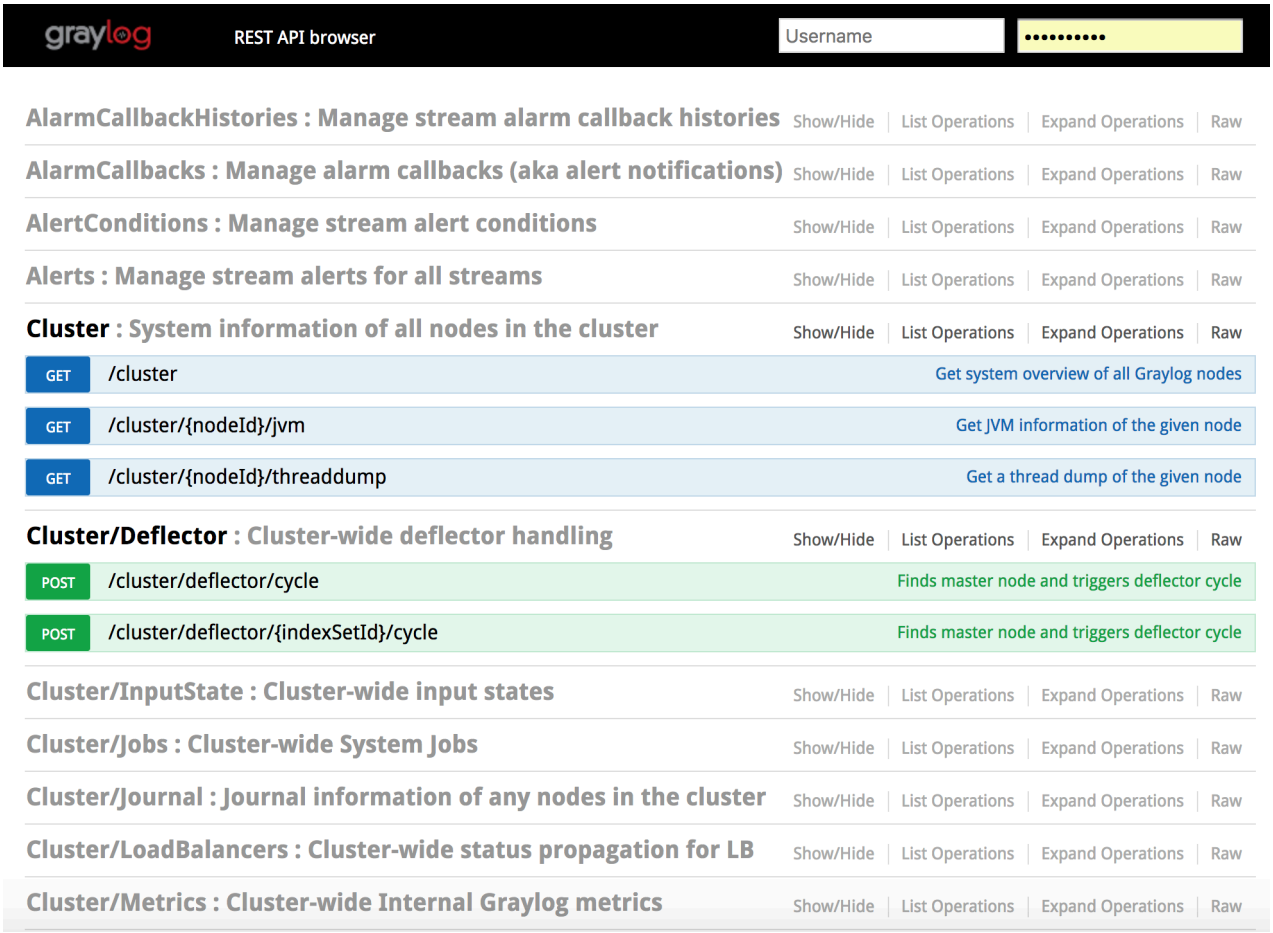
Remaining part of the pipeline did not run due to failed staging test. I.e. issues have been correctly caught in STAGE environment and not propagated to PROD:



Issue #4 Staging Tests

Staging tests are one of the core components of CICD. Without them you can not be sure that what you do in DEV will work also in PROD.
Basic service checks are already part of Ansible playbooks and all service restart are followed by respective *wait_for* clause (http://docs.ansible.com/ansible/latest/modules/wait_for_module.html). It is possible to do this check at the end of every relevant tasks. Our preference is to do it immediately after service restarts to catch the problem immediately.

It is recommended also to include application-level checks. Graylog APIs can help significantly and you can build various tests that way. We use Graylog APIs also for various other purposes, e.g. to detect which cluster is active at the moment or to quickly collect performance information. Graylog API is defined using Swagger framework.



The goal is not to test API functionality itself, rather imply from API response what does work and what does not on application level. Few examples follow:

Test1:
Send unique messages via API to all Graylog nodes, e.g. message1 => node1, message2 => node2, etc.
Search unique message across Graylog nodes, e.g. message1 on node2, message2 on node1.
Expected result:
Success means that all nodes can receive logs, process them and all have consistent view onto Elasticsearch database.

Test2:
a) Create user on node1
b) List user on node2
c) Delete user on node3
d) Check user deletion on node4
Expected results:
Success means that MongoDB cluster is consistent and configuration changes are replicated within the MongoDB Replica set properly.

Test3:
Identify amount of unprocessed messages.
Expected result:
Low number of unprocessed messages means healthy and sustainable operation.

Issue No. 5: Parsing

Most of the critical issues we have identified were related to the applied SIEM logic - log parsing or log processing via Graylog Pipelines (not to confuse with Gitlab pipelines). Graylog Extractors are in fact log parsers which can be applied in multiple ways as regex, Grok parsers, JSON parsers, etc. It is recommended to invest time to get familiar with writing effective regular expression to avoid computation demanding regex-es which might impact the infrastructure significantly, causing even a node failure.

Do not parse everything. Do not try to split every possible log message up to the lowest atomic item. Parse only what you need for log analysis and for building alerting rules. Suboptimal regex syntax can lead to big increase of log message processing time. This problem may be overlooked in low volume environment as DEV and may have deadly impact in PROD where extractor execution time is multiplied by the factor of N depending on the production load.

You can catch parsing problems in STAGE by collecting Graylog application metrics (e.g. Extractor execution times). However this approach is still not fail proof since during staging test you may miss peaks in the load which in PROD might do the difference. Another situation relates to special log messages as e.g. complex multi-line logs or other types of logs that simply do not arrive within the staging test time frame. That is why we also introduced continual metrics analysis on PROD besides ad-hoc checks during CICD staging phase. As a tool we selected InfluxDB and Grafana as part of our monitoring ecosystem.
Besides performance impact log parsing can also introduce logical errors. Graylog tries to parse certain fields automatically and expects certain data type at certain default fields.
E.g. JSON parser splits all the key-value pairs automatically. Multiple keys are transformed automatically to a dedicated message fields. Based on the message fields you can then execute searches or set alerts within the Graylog application. There are several custom fields as Message, Source, Facility, Level or Timestamp detected directly by Graylog Extractors. Graylog allows you to create own message fields as part of the parsing process or within pipeline processing.
Problem emerges when a message payload contains a string as a value when expecting numeric value for a well-known key (or vice versa).
For example, when parsing Linux logs, the key “level” contains numeric value and JSON parser pick it up properly and puts into respective field in Graylog/Elasticsearch. Windows logs may store to the same key a string value. The JSON parser does its job, however Graylog identifies in its well-known field unexpected value (string).

Raw message from Windows:


```
{"@timestamp":"2018-04-23T13:40:06.484Z","@metadata":
{"beat":"winlogbeat","type":"doc","version":"6.2.3","topic":"xxxxxxxxxxxxxx"},"
provider_guid":"{xxxxxxxxxxxxxx}","keywords":["Classic"],"thread_id":
16456,"source_name":"Service Control
Manager","level":"Information","type":"wineventlog","log_name":"System","comput
er_name":"xxxxxxxxxxxxxx","message":"The WMI Performance Adapter service
entered the stopped state.","process_id":840,"event_data":{"param1":"WMI
Performance
Adapter","param2":"stopped","Binary":"xxxxxxxxxxxxxx"},"record_number":"230181"
```

```
,"event_id":7036,"beat":
{"hostname":"xxxxxxxxxxxxx","version":"6.2.3","name":"xxxxxxxxxxxxx"}}
```


Indexer failures detected in Elasticsearch log:


```
43 minutes ago    graylog_84    d3878612-46fb-11e8-b940-fa163eb446ac
{"type":"mapper_parsing_exception","reason":"failed to parse
[level]","caused_by":{"type":"number_format_exception","reason":"For input
string: \"Information\""}}
```


Such issues have to be fixed using Extractor modifications:

`Condition`

- `Will only attempt to run if the message includes the string "type":"wineventlog"`

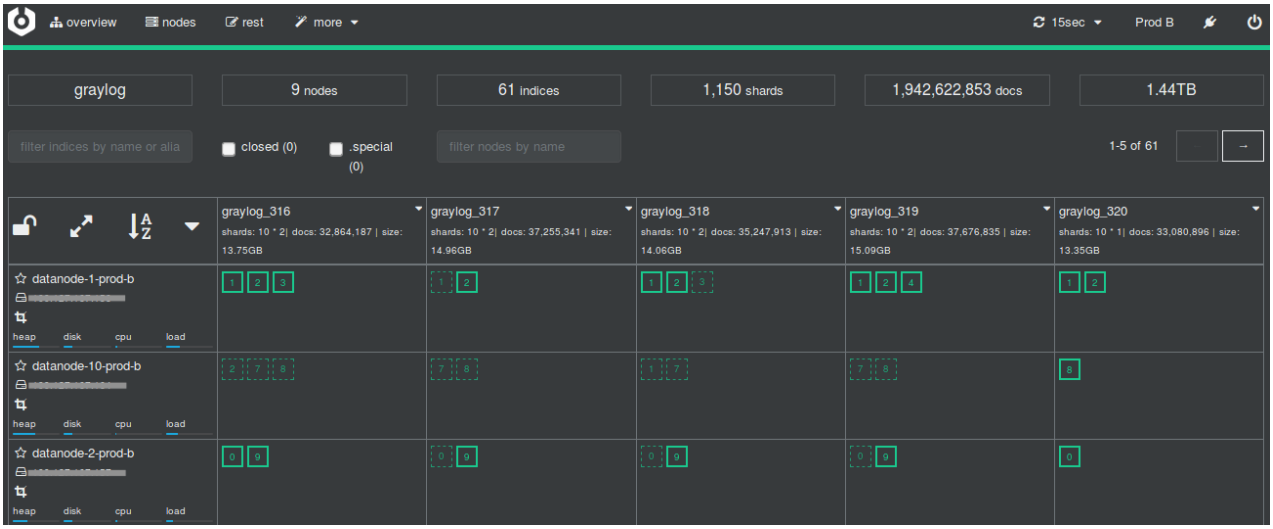
`Configuration`

- `replacement: "win_level":`
- `regex: "level":`

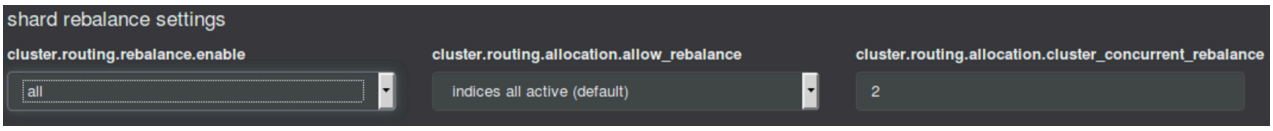
Issue No. 6: Detailed Service Monitoring

Some parameters should be monitored continuously to have full visibility when having operational problems. We decided to implement multiple monitoring tools to have detailed overview. We have not identified a single tool which would serve all use cases.

Elasticsearch monitoring:
For detailed monitoring we use ElasticHQ and Cerebro. Cerebro (replaced Kopf which used to be part of Openstack images for Graylog). Both are external tools connecting to Elasticsearch nodes on port 9200/tcp. ElasticHQ provides more detailed statistics comparing to Cerebro.
However we use Cerebro more due to nice visualisation of Shard allocation per Elasticsearch node. This overview helped us when recovering Elasticsearch nodes and checking how Shards are being reallocated.



This way we quickly identified the problem of having most of the Shards allocated on a single node. Even after rotating the Active write Index we still had most of the Shards allocated to one specific node. We had to rotate Active write Index multiple times, since Graylog has a limit of reallocating maximum of two Shards in time by default when rotating the Index. Settings from Cerebro:



Based on behaviour detected in Cerebro we adjusted also several other parameters to prevent too many Shard reallocations when not needed:

```
# Recover Recover only after the given number of nodes have joined the cluster.
Can be seen as "minimum number of nodes to attempt recovery at all".
gateway.recover_after_nodes: 8
# Time to wait for additional nodes after recover_after_nodes is met.
gateway.recover_after_time: 5m
# Inform ElasticSearch how many nodes form a full cluster. If this number is
met, start up immediately.
gateway.expected_nodes: 10
```

Elasticsearch documentation might come in handy when fine tuning the setup: http://docs.graylog.org/en/2.4/pages/configuration/elasticsearch.html#configuring-es

Performance monitoring:
There are many options for performance monitoring as Nagios, Prometheus, etc. Our setup is as follows:

a) Graylog/Elasticsearch	=>	Prometheus	=>	Grafana	=>	Slack
b) Graylog/Elasticsearch	=>	Prometheus	=>	PagerDuty	=>	Slack
c) Graylog/Elasticsearch	=>	Python scripts	=>	InfluxDB	=>	Grafana

a) Graylog nodes run Prometheus Node Exporters and service-specific Exporters. Grafana is configured with Prometheus as a Data Source and visualizes various metrics trends. Grafana can also provide notifications to our main ChatOps communication system Slack including graph screenshots.
b) Alerts/notifications are being in parallel sent to PagerDuty system for central alert management.

Note: We identified several limitations of Prometheus plugin for Graylog. It was not possible to get only the selected metrics, resulting in tons of data being sent to Prometheus and thus overloading it. We applied countermeasure on Prometheus side. We scrape all the metrics and delete later on inside Prometheus to minimise the impact.

c) Prometheus plugin for Graylog is able to scrape wide range of metrics. However these metrics do not always provide human-readable format needed for further investigation. In our use case it relates mainly to identification of Streams in descriptive format instead of Stream ID. Extractor and Stream monitoring is of utmost importance to us, that is why we collect these information on top using custom scripts run against Graylog APIs. We send these data to a time-series database InfluxDB.

Issue No. 7: Data Storage - CEPH

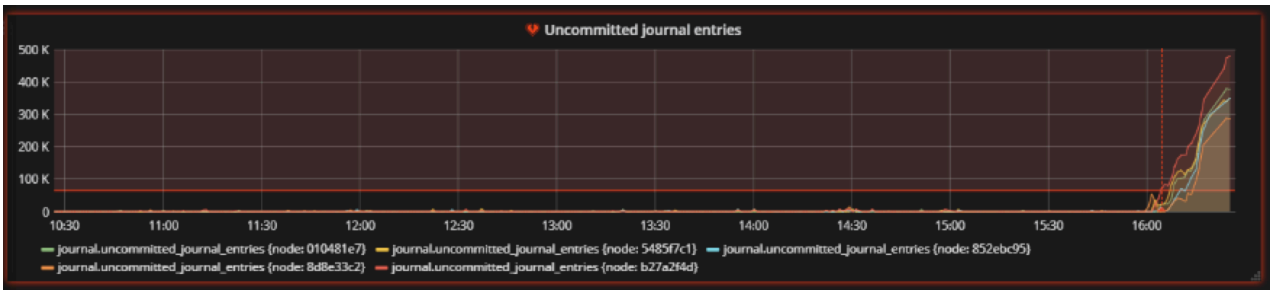
CEPH is a fault-tolerant distributed storage platform which we use within the Openstack deployment.

All the VMs may use:

- local disk space taken from Hypervisor nodes as part of VM flavor (not part of CEPH cluster),
- or VMs can use CEPH-based Cinder mounts. Cinder is a Block Storage service for Openstack.

Cinder-mounted volumes are in our case physically attached to SSD-based storage nodes and are part of a CEPH cluster. Graylog application nodes are relying on local non-CEPH disks only. Elasticsearch nodes do have attached Cinder volumes of 1 TB per node.

We experienced several problems with Graylog not being able to write to Elasticsearch DB running on CEPH volumes. Problems were identified without any apparent cause since Graylog PROD cluster was not touched in any way and load was also stable.



Later we correlated this incident to Openstack resource deletions (deleting large Heat stacks) when several terabytes of data have been suddenly available for CEPH and CEPH was doing too many operations to reallocate and rebalance own data and thus slowing down the Graylog performance. This problem would not be detected on common systems with lower load. However busy systems as Graylog recognize every latency increase immediately. By scaling the backend CEPH infrastructure the problem became less and less significant.

Another problem we identified was CEPH not reclaiming unused blocks. As Graylog was filling data into the Elasticsearch DB, CEPH disks got full despite Index rotation and retention policies properly set in Graylog. Data areas that supposed to be considered unused were not properly reclaimed and were considered by CEPH as stil used. In this situation CEPH continued to allocate new blocks in infinite loop exhausting all the storage resources.

It was needed to fix this problem on VM side. When building or updating the VM image it is needed to modify hw_scsi_model and hw_disk_bus within VM's metadata in the Openstack:

```
$ glance image-create --name='graylog-2.4.3-1-virtio-scsi' --container-
format=bare --disk-format=qcow2 --property='hw_scsi_model=virtio-scsi' --
property='hw_disk_bus=scsi' --file graylog-2.4.3-1.qcow2
```

Additional condition has to be met as well. On OS level it is needed to mount volumes using *discard* flag in the /etc/fstab:

```
/dev/sdb     /var/opt/graylog/data     auto     defaults,nofail,discard 0 3
```

After modification of VM metadata, disk device changes from default /dev/vda to /dev/sda(b).

Another usable command is manual ad-hoc fstrim. It is used on a mounted filesystem to discard (or "trim") blocks which are not in use by the filesystem. Real command usage is disputable due to very long runtime on large data volumes.

```
$ sudo fstrim /var/opt/graylog/data
```

Ad-hoc volume remounting with modified discard/nodiscard option is possible as well:

```
$ mount -o remount,nodiscard <mountpoint>
```

To display actual mount option setting use:

```
$ cat /proc/mounts
```

Note: `mount` *command is not fully precise and shows both discard and nodiscard options together when remounted disk manually and provides ambiguous output.*

Issue No. 8: Log Aggregation

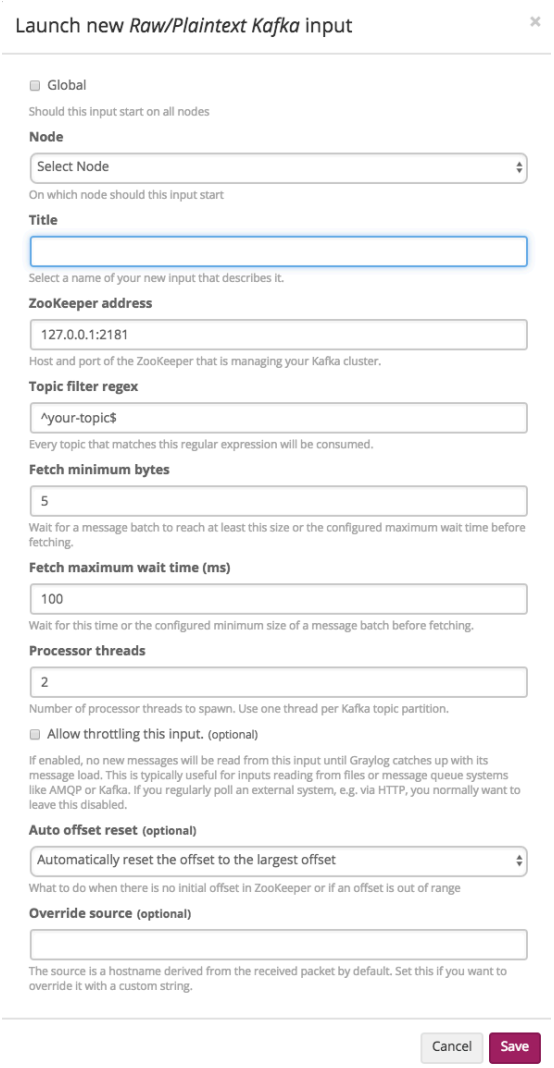
Small Graylog instances or even small clusters are pretty fine with collecting logs directly from log producers via e.g. direct syslog delivery to UDP or TCP Inputs in Graylog.
However, you can not be always sure that the amount of logs coming in is stable. Sometimes log producers simply set log level to *debug* due to internal troubleshooting and your logging platform may end up in big problems. Shock absorber is needed here and we selected Apache Kafka message broker for this purpose. Message broker platform brings also several additional valuable functionalities as data replication, load management, data preprocessing, long-term efficient archival, etc.

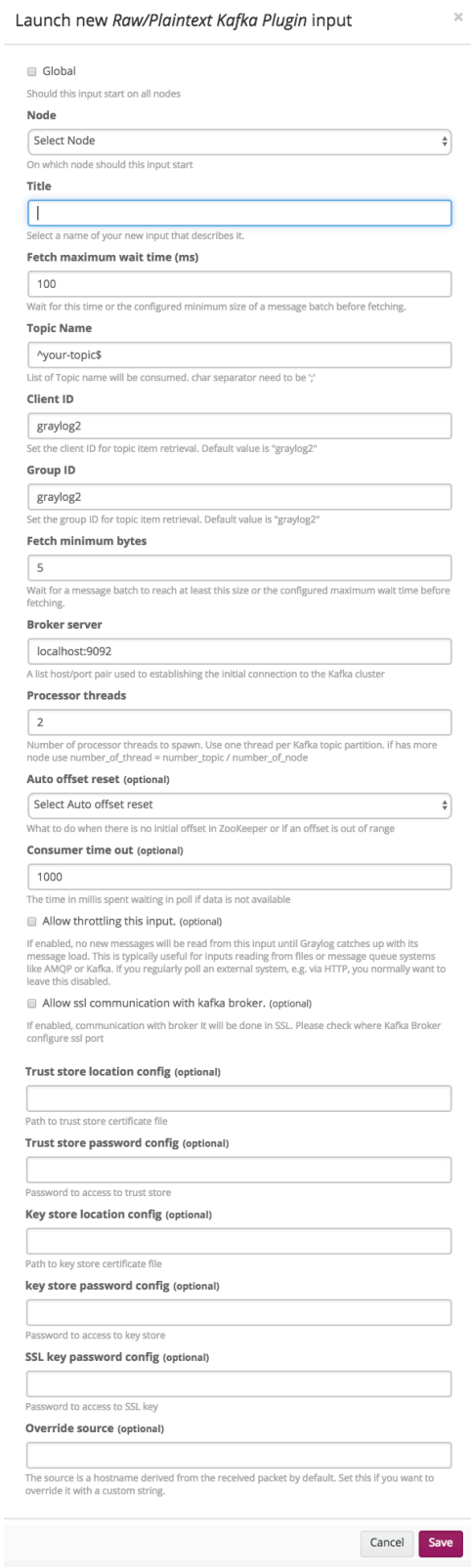
Data replication:
When it is needed to provide the same data/logs to various systems to keep them in sync, e.g. DEV, STAGE, PROD environment.
Load management:
To throttle how much data can be pulled by specific consumers (e.g. Graylog) based on their performance limits.
Data preprocessing:
To save resources on Graylog side and to do some heavy lifting on Kafka side, e.g. deleting unneeded data.
Long-term storage:
To store data in efficient manner with longer retention policies than Graylog sizing allows.

Graylog supports Kafka-based log delivery using native Kafka Input. Kafka Inputs can be set as any other Inputs globally or per-node and thus pulling data from Kafka directly. Native Kafka Input provides limited options and is suitable only for small deployments and simple use cases. We had to utilize external Kafka plugin from the community and we included several custom improvements on a code level to fit to our needs.

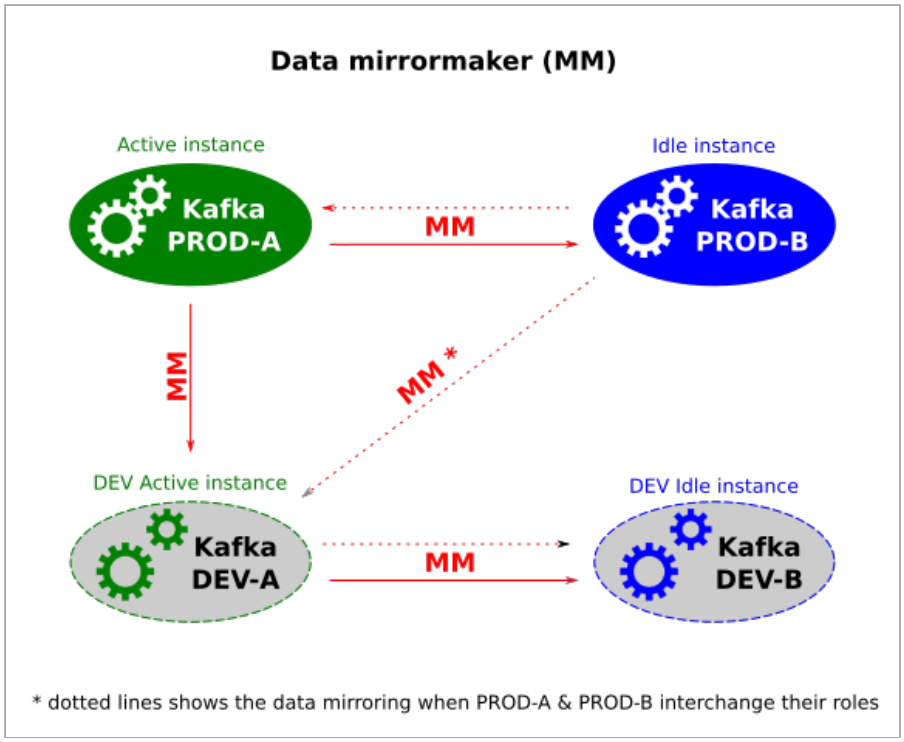
Native Kafka plugin (image on the left below) does not support SSL, Bootstrap protocol and has hardcoded several parameters which we needed to adjust, e.g. Client ID to apply quotas on Kafka side or Consumer Group ID to enable parallel log consumption by separate Graylog clusters.

Our modified Kafka plugin (image on the right below) was enriched by SSL encryption and Bootstrap protocol support. We added parametrization of Client ID, Consumer Group ID, Offset reset policies and SSL authentication.
This way we reached a goal of reliable, controlled and secure log consumption.


Launch new *Raw/Plaintext Kafka* input
Global
Should this input start on all nodes
Node
Select Node
On which node should this input start
Title
Select a name of your new input that describes it.
ZooKeeper address
127.0.0.1:2181
Host and port of the ZooKeeper that is managing your Kafka cluster.
Topic filter regex
^your-topic$
Every topic that matches this regular expression will be consumed.
Fetch minimum bytes
5
Wait for a message batch to reach at least this size or the configured maximum wait time before fetching.
Fetch maximum wait time (ms)
100
Wait for this time or the configured minimum size of a message batch before fetching.
Processor threads
2
Number of processor threads to spawn. Use one thread per Kafka topic partition.
Allow throttling this input. (optional)
If enabled, no new messages will be read from this input until Graylog catches up with its message load. This is typically useful for inputs reading from files or message queue systems like AMQP or Kafka. If you regularly poll an external system, e.g. via HTTP, you normally want to leave this disabled.
Auto offset reset (optional)
Automatically reset the offset to the largest offset
What to do when there is no initial offset in ZooKeeper or if an offset is out of range
Override source (optional)
The source is a hostname derived from the received packet by default. Set this if you want to override it with a custom string.
Cancel
Save


Launch new *Raw/Plaintext Kafka Plugin* input
Global
Should this input start on all nodes
Node
Select Node
On which node should this input start
Title
Select a name of your new input that describes it.
Fetch maximum wait time (ms)
100
Wait for this time or the configured minimum size of a message batch before fetching.
Topic Name
^your-topic$
List of Topic name will be consumed. char separator need to be ';'
Client ID
graylog2
Set the client ID for topic item retrieval. Default value is "graylog2"
Group ID
graylog2
Set the group ID for topic item retrieval. Default value is "graylog2"
Fetch minimum bytes
5
Wait for a message batch to reach at least this size or the configured maximum wait time before fetching.
Broker server
localhost:9092
A list host/port pair used to establishing the initial connection to the Kafka cluster
Processor threads
2
Number of processor threads to spawn. Use one thread per Kafka topic partition. if has more node use number_of_thread = number_topic / number_of_node
Auto offset reset (optional)
Select Auto offset reset
What to do when there is no initial offset in ZooKeeper or if an offset is out of range
Consumer time out (optional)
1000
The time in millis spent waiting in poll if data is not available
Allow throttling this input. (optional)
If enabled, no new messages will be read from this input until Graylog catches up with its message load. This is typically useful for inputs reading from files or message queue systems like AMQP or Kafka. If you regularly poll an external system, e.g. via HTTP, you normally want to leave this disabled.
Allow ssl communication with kafka broker. (optional)
If enabled, communication with broker it will be done in SSL. Please check where Kafka Broker configure ssl port
Trust store location config (optional)
Path to trust store certificate file
Trust store password config (optional)
Password to access to trust store
Key store location config (optional)
Path to key store certificate file
key store password config (optional)
Password to access to key store
SSL key password config (optional)
Password to access to SSL key
Override source (optional)
The source is a hostname derived from the received packet by default. Set this if you want to override it with a custom string.
Cancel
Save

The same CICD principles we use for Graylog are adopted for Kafka as well, i.e. one-click deployment via Ansible and multi-environment setup. As we need to replicate data among Graylog clusters we need the same for Kafka cluster. For this purpose we use Kafka's native mirroring capability - MirrorMaker.



By having the same data on all Kafka environments we can safely apply blue-green setup and test changes in Kafka DEV under data load.

However not every log producer can speak to Kafka directly via kafka module for rsyslog or Elastic Beats. To collect logs from devices like firewalls, switches and routers it is needed to allow also simplest log delivery method as a standard syslog-based UDP delivery.
One of the options is to create Syslog Inputs in Graylog besides Kafka Inputs for legacy producers. It creates a need for a load balancer in front of Graylog nodes to balance the incoming (push-based) traffic and data replication among multiple Graylog cluster would have to be solved.
More suitable option is to have Kafka Inputs in Graylog only to have homogeneous setup and to create Syslog-to-Kafka transformation element for legacy systems. In our infrastructure we call this transformation point Syslog proxy. From customer's perspective it is a standard syslog endpoint exposed on 514/udp. Syslog proxy acts simultaneously as a Kafka producer and all logs it gets are forwarded to a dedicated "syslog-proxy" topic in Kafka. In the real operation there are multiple producers sending logs via Syslog proxy and we want these data to be separated in Graylog.

Options as as follows:
a) To have multiple Syslog proxies for every log producer and sending data to producer-specific Kafka topic.
b) To have one Syslog proxy and to split data there and send them to multiple producer-specific Kafka topics.
c) To have one Syslog proxy and one Kafka topic and splitting data using Graylog Streams inside Graylog.

d) One Syslog proxy and one main receiving Kafka topic (syslog-proxy topic) and splitting data in front of Graylog in a preprocessing layer so that Graylog will receive prefiltered and properly split messages to dedicated Inputs.

We decided for option d) to split syslog-proxy topic within preprocessing layer which splits Kafka topic into multiple Kafka (sub)topics which are then consumed by separate Kafka Inputs on Graylog side. This is reached using Kafka Streams which is elaborated more in next chapters. Kafka Streams provide API-based data transformation and sorting capabilities. Even more complex use cases can be delivered by integrating Kafka to Apache Spark, Apache Storm or Apache Beam.

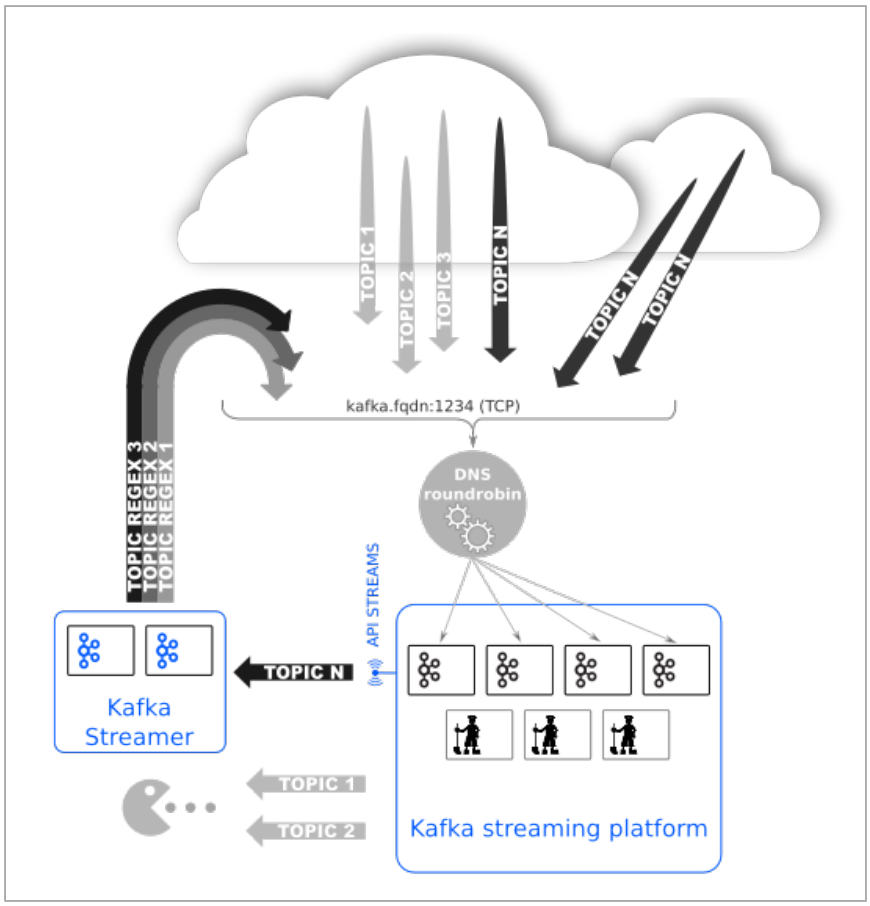
Data Archival
Raw messages in Elasticsearch database occupy a lot of space and having a longer retention policy may be a problem. For forensics and archival purposes we decided to set generous retention policies in Kafka, however still having in mind platform resiliency for cases of Kafka node outage. Kafka is able to compress stored data and act as an efficient long-term repository.
In case operator needs to get older data back to Graylog, it is just need to create Kafka Input with proper Kafka topic selection and to set offset reset policy to *Earliest*. This Input configuration will pick up the whole history of the specified Kafka topic. For this purpose we use a dedicated Graylog instance not to interfere with PROD Graylog cluster.
Another option for log archival is offered by Graylog directly as part of Enterprise subscription.

Issue No. 9: Log Preprocessing

Kafka acts as a shock absorber, data archival engine and high performant message broker. On top of this Kafka brings multiple integration options with external components for batch and stream data processing as for example Apache Spark, Apache Storm, Apache Beam. Since Apache Kafka v0.10 additional functionality has been added directly to Kafka - Kafka Streams.

Streams API allows us to process any Kafka topic and split it into multiple new topics based on our preference. It is possible to apply whitelists and blacklists using regular expressions and forward the data from source topic to custom (sub)topics. Graylog can then pull only needed data and not the whole data sets from log producers.

To talk to Streams API we developed own application we call **Streamer** for managing the process programatically.



Our lessons learned are as follows:

- Kafka Streams API before version 1.0.x has a buggy exception handling. Might be needed to workaround on code level of the connected application.
- When Kafka is in rebalancing state (e.g. after restart or just as natural behaviour) it can cause a shutdown of the application connected to Streams API. This has to be managed on a code level of the application connected to Streams API.
- Streams API < 1.0.x multi-threading was not working for our Streamer - now it does up to 10 threads for each cluster node.
- With new exception handling Streamer tries 10 times per partition to reconnect, if that fails (e.g. caused by rebalancing or consumer group is marked as dead), it stops, revoke all partitions, restarts by itself and rejoin with a fresh stream --> this "self healing" works with 1.0.1
- Implement input validation when accessing Streams API on application side to avoid problems on Kafka and streaming application side.

Issue No. 10: Data Formatting

Data/Log producers may send data in various data formats. Elastic Beats might send JSON formatted data to Kafka, rsyslog may prefer delimited text and so on. Kafka serializes everything it consumes and Kafka consumers deserializes byte arrays back. When consumer (in our case Graylog) deserializes different types of underlying messages back it is needed to apply particular parsing on a particular pulled topic to have uniform data representation in Graylog. It has a load impact on the consuming system and possibly a certain un-needed overhead for maintaining various parsing approaches.

Another shortcoming of heterogeneous data formats in Kafka emerges when we try to apply big data analytics and want to use uniform parsing algorithms in stream or batch processing platforms which gets data from Kafka. Unified data formats can be reached via formal procedures to instruct producers how to format data on their side. In complex environment it is not the way to go to and we need an enforcement point - for example Apache Avro.

In the time of writing this document we have started experimenting with Avro and also assessing capabilities of our Streamer platform and extending its capabilities to use not only Streams API but also Connect API for message formatting.

Issue No. 11: Load Balancing

Both Graylog and Kafka are multi-node clusters. Sending data in and also getting data out requires load balancing.

Graylog:
Originally we have been sending data to Graylog via UDP-based syslog Inputs. That is why our platform of choice was Nginx which balances UDP pretty well. User access via browser was a bit more complicated due to complexity of Graylog graphical user interface. It is necessary to keep session persistence on GUI to avoid a situation that every component of the web page will be server by a different Graylog node. It was not fully efficient to use simple session persistence methods as ip_hash.
For advanced session persistence as sticky cookies you can go with paid Nginx Plus or HAProxy. We selected HAProxy which resulted in having two separate LB technologies in place. We have not found this as a problem since LB stack was rebuilt several times moving from VMs to Docker containers or considering to substitute HAProxy for Traefik later on. Currently all our load balancers run as Docker containers in a dedicated Docker infrastructure.

Kafka:
Log consumers are pulling data from Kafka directly that is why no load balancer was needed in front of the Graylog for message delivery. All Graylog Inputs should be set as Global Inputs so that all Graylog nodes will load balance data pulls on its own using various mechanisms embedded in the Kafka plugin for Graylog or in Kafka directly:

- Multithreading on Input
- Throttling on Input
- Offset reset policy
- Offset retention set on Kafka side
- Quotas set on Kafka side using Client-IDs

It is strongly recommended to analyze all load balancing and throttling options on Kafka and Graylog side. This area is quite complex and picking the right balance and configuration setting might be a challenge. Diving deep into all possibilities is worth a separate case study.

Issue #12 Zero Downtime Operation

CICD cycle and propagation of DEV changes to PROD may require service restarts, i.e. downtimes. There are techniques to avoid downtimes as for example rolling updates/upgrades. These can be heavily supported by container platforms as Docker Swarm or Kubernetes where containers are restarted in a scheduled manner. Another option (mainly for not containerized components) is Blue-Green setup.

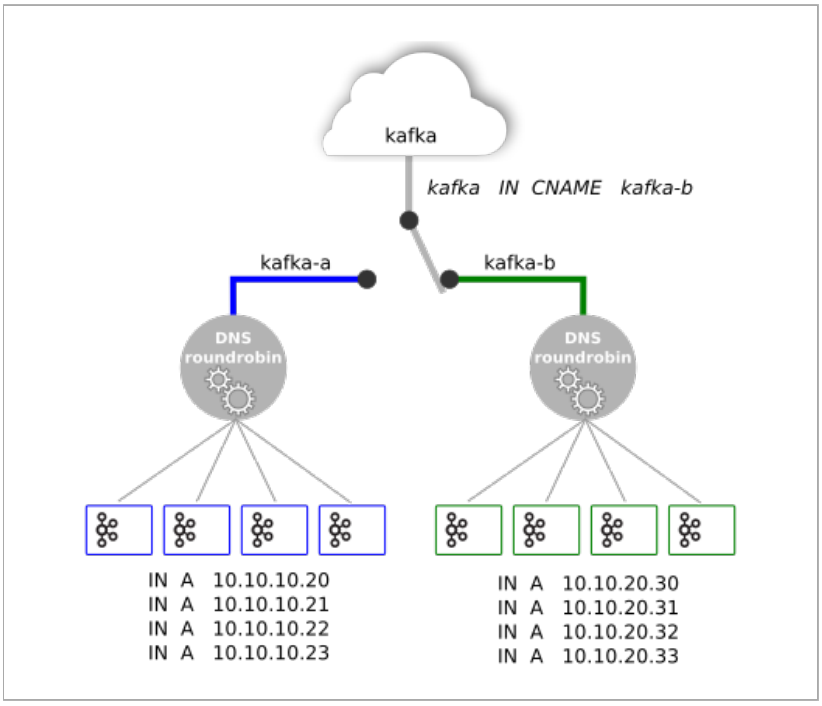
Blue-Green setup simply means to have two identical environments (PROD-A and PROD-B). There is also possibility to utilize STAGE environment as PROD-B in case we have some resource limitations.

One of the PROD environments is active and the other one is idle. CICD propagates changes to the idle one. After CICD cycle runs successfully against idle cluster it is needed to switch operation from idle to active PROD, e.g. from PROD-A to PROD-B.

Blue-Green switch can be provided on:
a) load balancer level by changing the backend configuration,
b) using DNS change when balancing the platform via DNS.

Load balancers should be configured in HA setup and backend configuration change is applied as rolling update within LB couple.

DNS-based switch has several disadvantages - health-checks are not always included and configuration change does not happen immediately due to TTL of the zone records. TTL can be lowered to zero, however real TTL setting is mostly enforced by a DNS resolving client, e.g. browser which do not allow zero TTL and fallbacks to 600 seconds, i.e. 5 minutes. There are several options how to mitigate shortcomings of DNS-based approach, e.g. Amazon Route 53 or Hashicorp Consul. The advantage of DNS-based balancing is simplicity and no need for additional component as load balancer.
DNS-based switch assumes having DNS records set in the DNS zone files for both PROD-A and PROD-B cluster nodes. For the main service FQDN, e.g. graylog.mydomain.com there is a CNAME resource record pointing either to A-nodes or B-nodes in DNS-balanced round-robin fashion. Switch in its essence modifies CNAME resource record to point to the desired cluster. This can be automated by having API-based DNS server or using service discovery platform.



Within our data centres we use both options for various use cases. For Graylog we do blue-green switch for user access on HAProxy load balancers. For Apache Kafka we adopted DNS based switch since load balancing is managed by Kafka itself without need for dedicated LBs.

When switching between Blue and Green there are several options how to cope with active sessions. We can:

- enforce them to drop and reconnect,
- let them continue to work within reasonable time frame and naturally timeout or exit.

Issue No. 13: Geo-redundancy

Graylog, Elasticsearch and MongoDB set as clusters are pretty stable options. In case we want to provide additional level of resiliency we must look beyond single-DC horizon. Even though cluster platform can be resilient by itself, corrupted network in cloudified environment can affect everything deployed within a single data centre. Geographically redundant architecture is the final solution. Geo-clusters are always tricky, especially when dealing with database replication and latency.
Clusters itself are deployed as IaaC (Infrastucture as a Code) so Gitlab Runners can build the same cluster in multiple data centres easily. CICD cycles can propagate changes from DEV environment to as many PROD environments as we need.

We decided to build a geo-cluster without need to sync neither configuration database (MongoDB) nor indexed logs (Elasticsearch) between PRODs. MongoDB is propagated via CICD and pushed to as many target clusters as we need and Kafka can help to keep Elasticsearch databases of dislocated clusters in sync using data forking/replication. Both Graylog PROD clusters act as a separate consumers from Kafka perspective. When talking to Kafka they introduce themselves using separate Consumer Group ID. Kafka

keeps separate offsets for separate Consumer Group IDs. You can put arbitrary amount of nodes to a single Consumer Group and they all will share the load coming from Kafka. In real scenario you would use one Consumer Group ID for Graylog cluster in location A and the other Consumer Group ID in location B.

When Graylog cluster fails in location A, Graylog cluster in location B continues in log processing from Kafka and cab still act as a real-time customer-facing platform. Users are being forwarded to the working cluster using standard blue-green switch methods. Recovered cluster in location A will consume the backlog as soon as it is up again and notifies Kafka about being back again. It will catch up with Graylog cluster in location B soon - depending on the outage duration and amount of accumulated backlog - known as Kafka lag.

Issue No. 14. SIEM Functions

Plugins:
You can extend the native SIEM functionalities via Graylog Marketplace (https://marketplace.graylog.org) where you can find plenty of interesting add-ons, e.g. for data aggregation, geolocation or threat intelligence. You have to be prepared that some external components might become native Graylog components in one point in time, so do not pick up from Marketplace every possible plugin and be wise following need-driven approach and assess and measure performance impact.
Not all plugins work smoothly and we had to customize code of some of them or found a workarounds to reach claimed functionality of the plugin.

These plugins came in handy for our deployment:

- graylog-plugin-aggregates - crucial plugin for alerting
- graylog-plugin-beats – used for internal logging
- graylog-plugin-collector
- graylog-plugin-kafka-custom – customized community plugin
- graylog-plugin-map-widget
- graylog-plugin-pipeline-processor
- graylog-plugin-slack – notifications to slack
- graylog-plugin-stringfn – hex to dec conversion (own development)
- graylog-plugin-threatintel
- metrics-reporter-prometheus

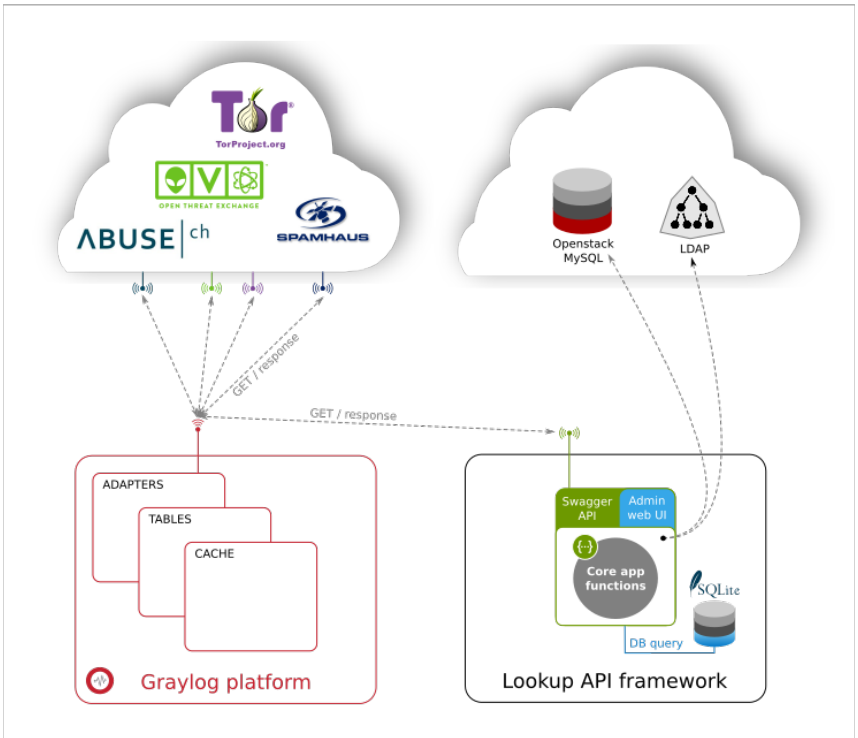
In several environments egress traffic is going out via proxy. Some plugins may have own proxy setting hardcoded inside and code level changes might be needed.

Time Sync:
Solid time source is very important not only for keeping clusters compact. SIEM use-cases and data correlations need precise time as well, not talking about forensics.
Do not underestimate NTP topic for any log management system. Your time source must be build in HA setup, easily reachable and precise.

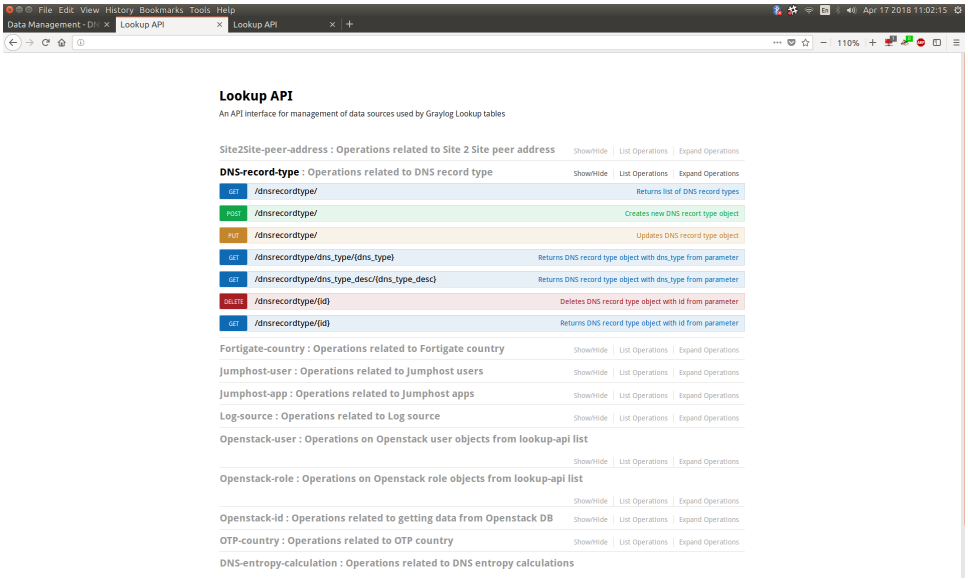
Data Correlations and Enrichment:
Graylog 2.3 introduced powerful feature of Lookup Tables. It allows to search for and translate message field values into new values. New values can be stored into new fields or can overwrite existing message fields. This mechanism can be applied using message Extractors or Pipeline rules.
It is possible to build interesting correlation rules, e.g. to check whether user X present in log messages belongs to the list of legitimate users stored in a dedicated lookup table. Data enrichment is also doable this

way, e.g. to translate virtual machine ID found in the logs into the human-readable VM name stored in lookup table which is generated from an Openstack API data dump.

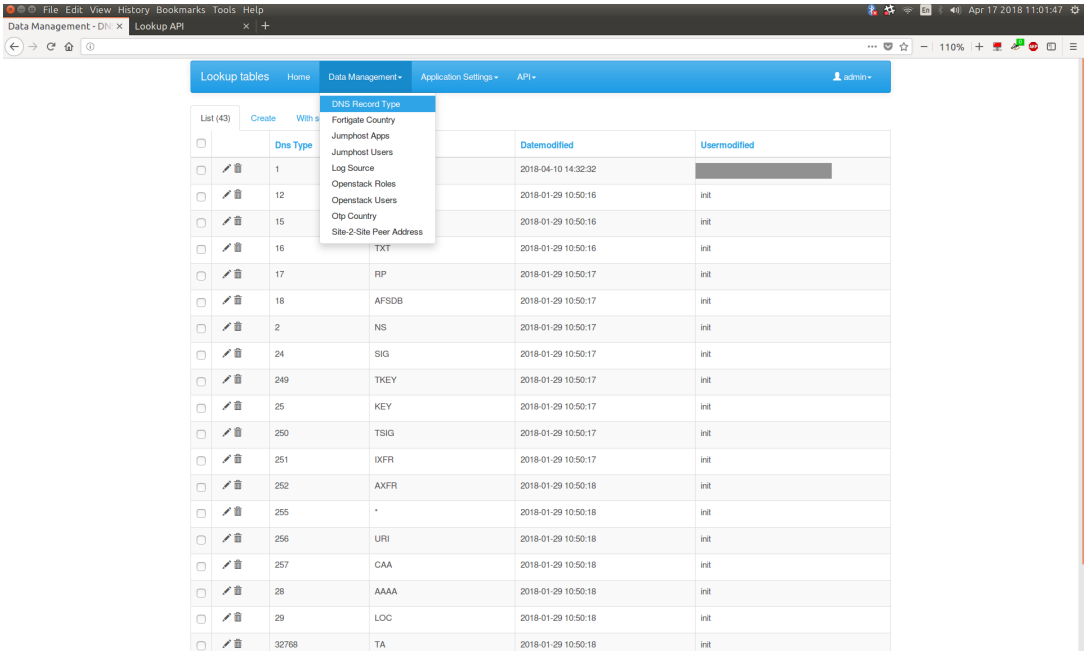
Having lookup tables either as local CSV on Graylog nodes or as static remote data source is not a long-term option. We want our lookup tables to be dynamic and up to date. Since we found a significant benefit of having external data sources for correlations and data enrichment we build a platform which we internally call **Lookup API Framework**.



Lookup API framework aggregates multiple data sources, internal as well as external. Information inside are collected and re-generated on a scheduled basis.
Combining our Lookup API Framework and Threat Intelligence plugin provides solid basis for data analytics and SIEM capabilities for Graylog.
All the data collected within Lookup API Framework are exposed to Graylog via own Swagger based API layer.



Administrator access and data editing is possible via web GUI as well.



Graylog provides basic Alert management capabilities natively. It is worth considering collecting alerts outside the Graylog. We have implemented alert notifications via Slack and also using home-made TTS (Text to Speech) functionality to make daily operation as comfortable as possible. Slack and TTS platform are used by L1 SIEM operators as the primary visual and audio notification interfaces. Subsequent analysis is executed using Graylog application via search queries and various data mining techniques.

FINAL WORDS

Deciding for Open Source platforms for large deployments is a serious step with significant impact on the overall operational model. It makes sense especially when the whole company, all interconnected systems and supporting systems are operated in the same way, e.g. DevOps.

Open Source platforms minimize vendor lock-in. However it has to be combined with a modular design to avoid platform lock-in or tightly coupled infrastructure with little ability to substitute one or the other component.

DT Pan-Net decision for using Open Source supports investment to own personnel and keeping the knowledge in-house. Of course you have to be prepared for longer time to delivery and time consuming troubleshooting, escalating the issues to the community and waiting for fix or own development or code customization. However we have identified much more benefits than drawback, increased flexibility when evolving and extending our infrastructure resulting in overall cost efficiency.