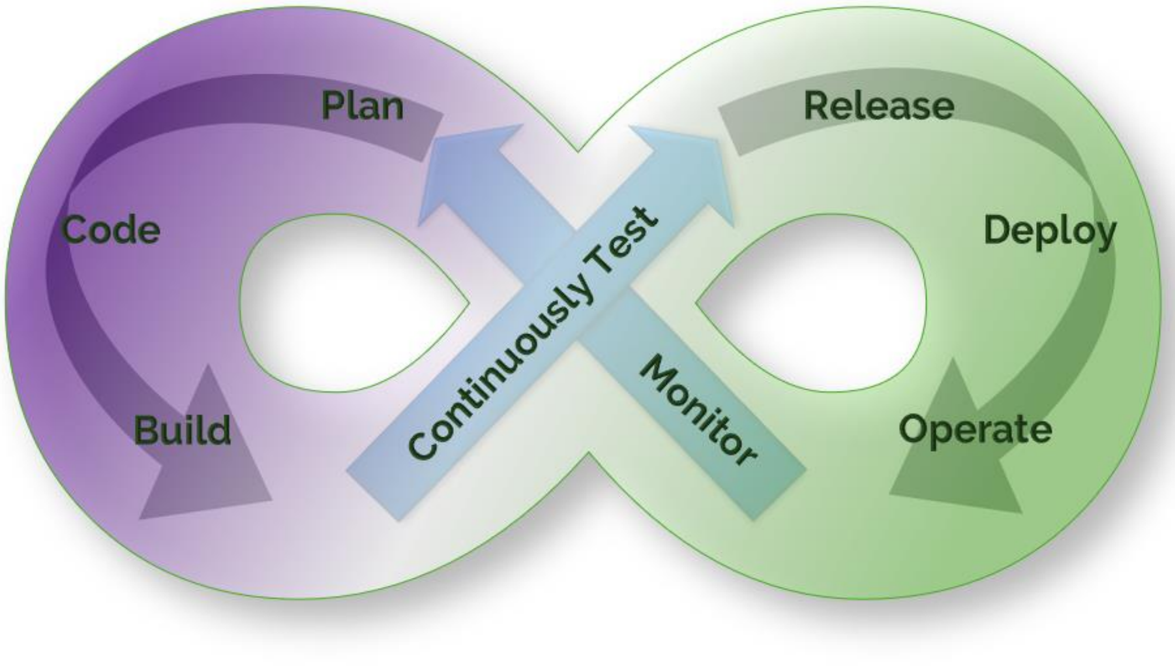
ThingWorx Monitoring and Alerting

Using Prometheus and Grafana

Introduction

As ThingWorx has become a more mature product during the lifetime of the IoT EDC, so too have our dev ops recommendations. As we've stated throughout many posts now, *testing* is a key part of ensuring enterprise readiness, and it occurs at every stage of the process: from unit testing to preserve individual service logic, to integration tests which preserve the functionality of the application as a whole, to user and edge load testing and user experience testing, which ensure enterprise readiness. So testing is a critical component, but *the process of dev ops never stops*. In order to effectively test the system, a comprehensive *monitoring* solution is also required.

Once the application is tested and the changes pushed into production, there is no knowing with certainty that everything will run smoothly indefinitely. Random spikes in usage, server bandwidth or availability, any unforeseeable factors like these can come along and cause issues for a system. If these issues aren't detected and addressed early, then they can very rapidly morph into much larger problems: outages, data loss, inflated data tables which are hard to revert due to their size. **It is critical to detect performance issues on a system as early as possible,** to have *as much information as is necessary* to figure out **where** the problem is heading, and **what** may have started it. **Monitoring is key to a healthy system**.



Caption: CI/CD stands for "Continuous Integration/Continuous Deployment", a never-ending cycle of improvement. Testing just once before the initial go live isn't enough. Each system should have automated tests that run continuously, as well as monitors and alerts which reveal problems *sooner. Diagnostic tools play a role as well, being the bridge from the end of the dev ops process cycle back to the beginning (monitoring into planning). A good CI/CD dev ops process will ensure that problems are found earlier, fixed more rapidly, and fixed for everyone using the system.*

In a fully mature dev ops pipeline, issues are anticipated, discovered and researched before they become production outages or critical issues. These investigations or testing follow-ups produce development tasks (usually bugs, but also features at times) which then start the dev ops cycle all over again. This is why a good, efficient dev ops pipeline is needed, one which allows changes to **quickly and safely** go from development to production.

This is also why diagnostic tools play a role in the monitoring piece of the dev ops process. They are the bridge between monitoring and planning. Tools like Dynatrace can be configured to provide call stacks and take thread dumps when issues *start* to occur, *before* the system is performing so poorly it needs a restart, which happens automatically in a cluster and can clear out any trace of the issue.

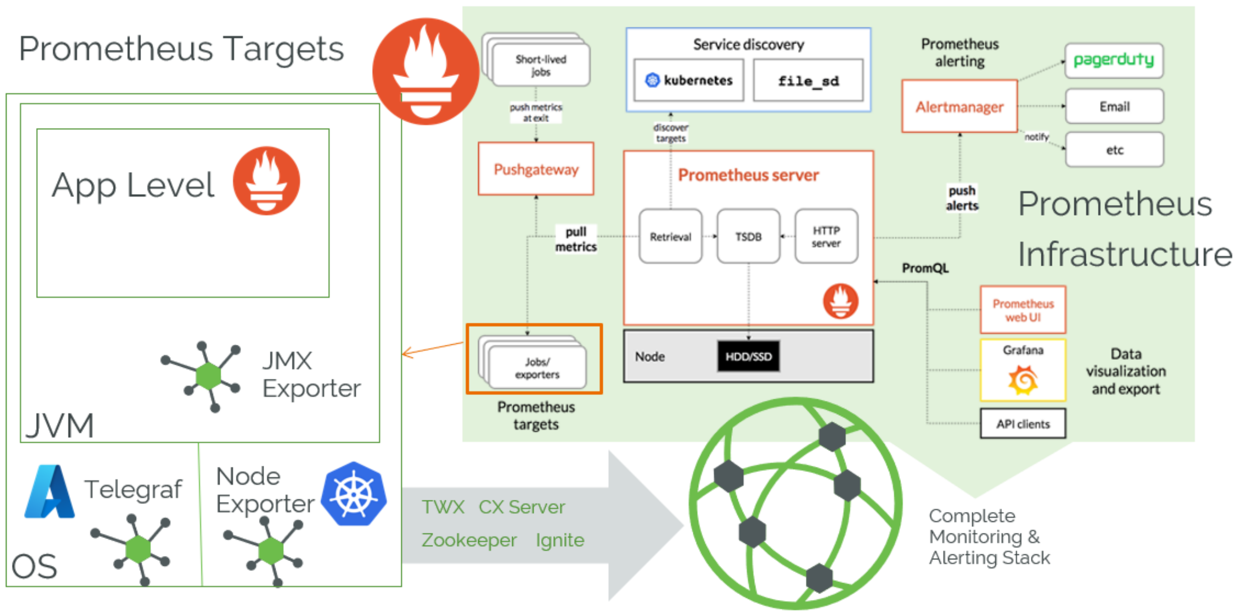
Thread dumps are often necessary to **diagnosing the root cause** of the issue (to permanently fix it), and doing so quickly ensures application stability and availability. That is, after all, the purpose of the dev ops process. *Diagnostics* is therefore an equally important piece of the dev ops Figure-8-shaped pie, and one which deserves its own spotlight in an article to come.

Every piece of the dev ops process **must be viewed as equally important** in its own way, lest the dev ops cycle get hung up on *bottlenecks of its own*. A safe and stable system is not one which never experiences issues, it is one which has a good, efficient plan in place to handle recovery and prevention of repetition. A wholesome dev ops process is a happy dev ops process.

The Monitoring Stack

There are many monitoring options available, but in our experience one of the easiest and most effective monitoring stacks to use with ThingWorx is Prometheus for metrics gathering with Grafana for metrics analysis and review. In a mature monitoring stack, Telegraf is also commonly installed on each VM/host to gather the system metrics (like CPU and Memory usage, things we've stated are good metrics of system performance and stability in past articles on scale and size testing) and output them in Prometheus format.

Prometheus is a highly scalable open-source monitoring framework that contains out of the box monitoring and alert capabilities for Kubernetes-based deployments (not covered in this article). Using Prometheus is very simple because the ThingWorx application exposes a metrics endpoint which is formatted directly for use by Prometheus. There is also built-in alerting in Prometheus, but not the ability to form dashboards for reviewing data or screenshotting it for documentation purposes. That's where Grafana comes into play. Grafana has a preconfigured Prometheus-type data source and many preconfigured dashboard templates for various applications and services. Telegraf is also easily imported into Grafana, as is shown in the section below.



Caption: The Prometheus targets in the larger diagram are expanded out on the left. For each target, some tool exports the data in a syntax which Prometheus can scrape. For VMs, this can be Telegraf, for Kubernetes, the Node Exporter. JVM has a JMX Exporter, and other tools like CX Server use Graphite. Many apps already have a Prometheus endpoint built-in, like ThingWorx and Zookeeper. Telegraf is not strictly necessary; the node exporter can also be used on VMs, but Telegraf is the more common choice since it is a more mature dev ops tool.

Once Prometheus is scraping the targets, alerting on them can be done with OOTB Prometheus functionality, and dashboards for monitoring can be made easily in Grafana (with built-in support as well). This stack does not include the diagnostics piece, something which triggers thread dumps or the like when issues do occur. There are too many ways to conduct a successful diagnostic piece to cover here.

How to Get Started

Getting started monitoring a ThingWorx application is incredibly easy in the latest versions. Simply open up a browser, and type in the ThingWorx URL, followed by "/Metrics". At this endpoint, there is a specially formatted response that can automatically be read by the Prometheus monitoring software which contains subsystem and service data. In addition to the application metrics, Prometheus can be configured to collect metrics from a node exporter at the (virtualized) operating system or container (Kubernetes) level as well.

If you haven't already, install Grafana, install Telegraf as a service, and install Docker Desktop. These are the tools required (in addition to ThingWorx of course) to set-up a simple sandbox system for familiarization with the monitoring stack recommended by PTC. The easiest way to try Prometheus on a local Windows instance is to use Docker. The command for that will be found below, but first open up Docker Desktop to set contextual parameters that the command line will need.

Then, modify the configuration file for Telegraf or create one (called *telegraf.conf* in the same folder as the exe file), and put the following into the file (or uncomment it; the default config file has thousands of lines, so just search for "prometheus"):

```
Output plugin
[[outputs.prometheus_client]]
listen = "0.0.0.0:9125"
```

Alternatively, install the Prometheus Node Exporter tool, which will likely require some additions to the Prometheus config file (not covered here) which we are about to create.

Then, create a configuration file (called *prom_config_localhost_scraper.yml* in the command to come), add the following (assuming a standard localhost installation of ThingWorx):

```
# my global config
global:
  scrape_interval: 45s
  evaluation_interval: 30s
  scrape_timeout: 30s
  # scrape_timeout is set to the global default (10s).

rule_files:
  - prom_config_rules.yml

scrape_configs:
  - job_name: thingworx
    static_configs:
      - targets: ['host.docker.internal:8080']
    basic_auth:
      username: "Administrator"
      password: "admin!123456789"
    metrics_path: /Thingworx/Metrics
    scheme: http
    params:
      x-thingworx-session:
        - "false"

  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']

  - job_name: Telegraf
    # If telegraf is installed, grab stats about the local
    # machine by default.
    static_configs:
      - targets: ['host.docker.internal:9125']
```

This example script file uses the `host.docker.internal` instead of `localhost` for the server target for ThingWorx because it is running outside of the Docker container which contains Prometheus. This yml file configures Prometheus to monitor both ThingWorx and itself, as well as the server metrics coming from Telegraf (as long as they are configured to push). It's a sandbox-only configuration, really, as you wouldn't want to use the Administrator user, or have the password printed in plain text in the config file in a real system. Also note the need for the `x-thingworx-session` parameter, as runaway sessions which spawn every 30s or so (whatever the scrape interval is) will result in memory issues over time (so we don't want to use sessions here).

The rules file given here (*prom_config_rules.yml*) needs to be created separately. This is where all of the alert rules will be defined. This will determine if an alert state is happening, but without configuring the alert manager, there won't be any notification. That isn't covered here but is covered extensively in the Grafana docs. Here is an alert example:

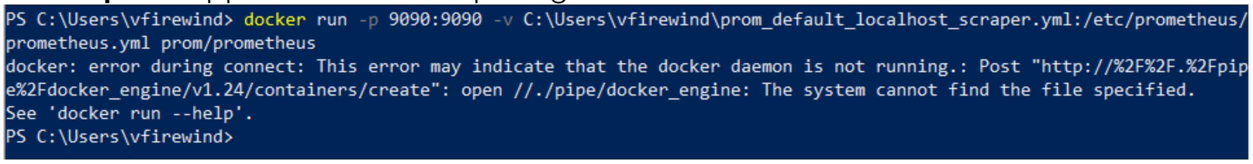
```
groups:
- name: alert.rules
  rules:

  # Alert for any instance that is unreachable for >5 minutes.
  - alert: HighMemory
    expr: mem_used > 14000000
    for: 1s
    labels:
      severity: page
    annotations:
      summary: "High Memory"
      description: "Localhost Memory Usage is High"
```

Now, save these files and use Powershell to run the Docker container:

```
docker run -p 9090:9090 -v
C:\<path_to_document>\prom_config_localhost_scraper.yml:/etc/prometheus
/prometheus.yml prom/prometheus
```

It should download Prometheus and install it in that container (if this is the first time), allowing you to very rapidly deploy it to an endpoint of `localhost:9090` by default.

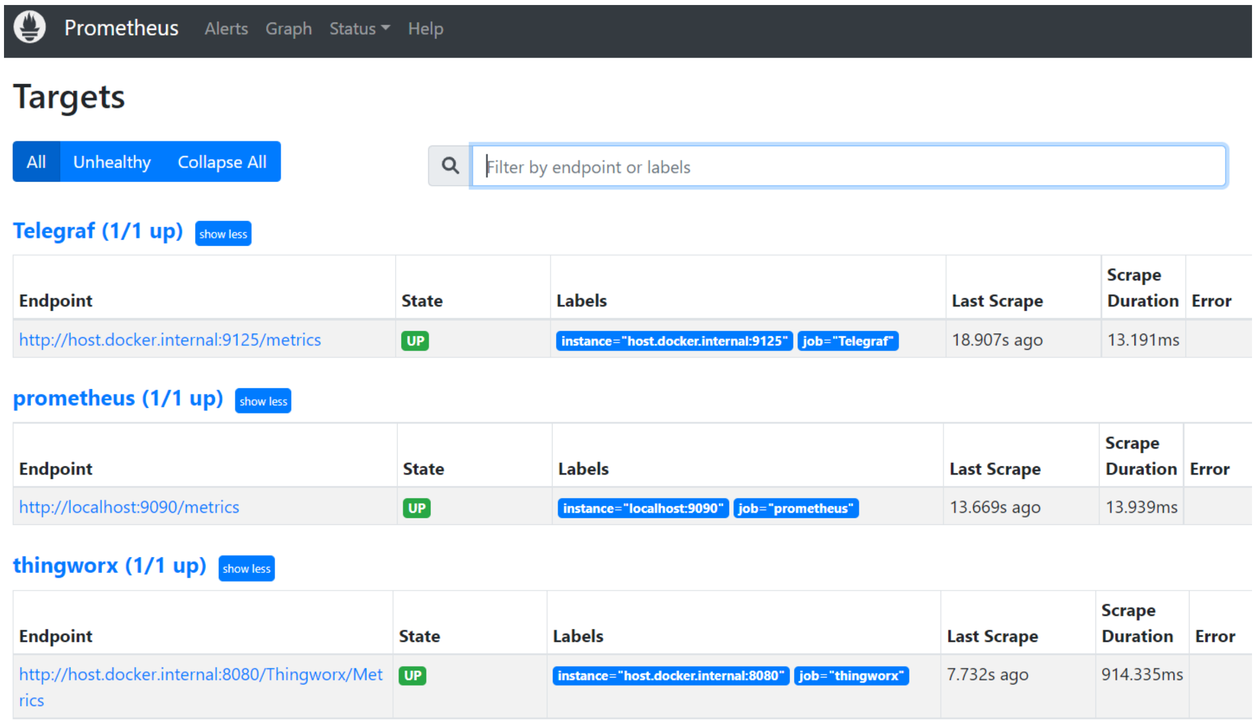
If there is an error like the one shown below, this means that you forgot to start **Docker Desktop** (the application) before opening Powershell.



```
PS C:\Users\vfirewind> docker run -p 9090:9090 -v C:\Users\vfirewind\prom_default_localhost_scraper.yml:/etc/prometheus/
prometheus.yml prom/prometheus
docker: error during connect: This error may indicate that the docker daemon is not running.: Post "http://%2F%2F.%2Fpip
e%2Fdocker_engine/v1.24/containers/create": open //./pipe/docker_engine: The system cannot find the file specified.
See 'docker run --help'.
PS C:\Users\vfirewind>
```

Caption: Docker Desktop sets system parameters required for containers to run in a command line (in Linux, it should work if Docker is installed for use by the command line, simple as that).

The localhost endpoints are accessible in a browser. ThingWorx defaults to `localhost:8080` endpoint. Prometheus defaults to `localhost:9090`. Telegraf is on port `9125`. Open any of these in a browser tab to see the full monitoring stack. You can see easily if Prometheus is working by clicking "Status" > "Targets" at `localhost:9090`:



Targets

Telegraf (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://host.docker.internal:9125/metrics	UP	instance="host.docker.internal:9125" job="Telegraf"	18.907s ago	13.191ms	

prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	13.669s ago	13.939ms	

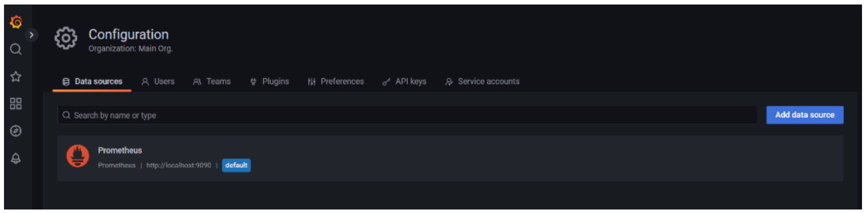
thingworx (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://host.docker.internal:8080/Thingworx/Metrics	UP	instance="host.docker.internal:8080" job="thingworx"	7.732s ago	914.335ms	

If all of the targets appear as blue and say "last scrape" and a time stamp, then they're working as expected. If they don't, ensure you have the right ports, that there aren't any firewall issues (if things aren't all on localhost), and that everything is running without errors.

The last step in the process here is to install a dashboard tool like Grafana. Once this is installed and running on `localhost:3000` (by default), you can display the data from Prometheus with a few configuration steps the Grafana UI.

Highlight over the settings icon in the bottom left of the screen, and then click on "Data sources". Select the "Add data source" button, and then click on Prometheus. You have to type the URL again (`localhost:9090`), but most of the defaults will be ok here, and all you have to do is click "Save and test".

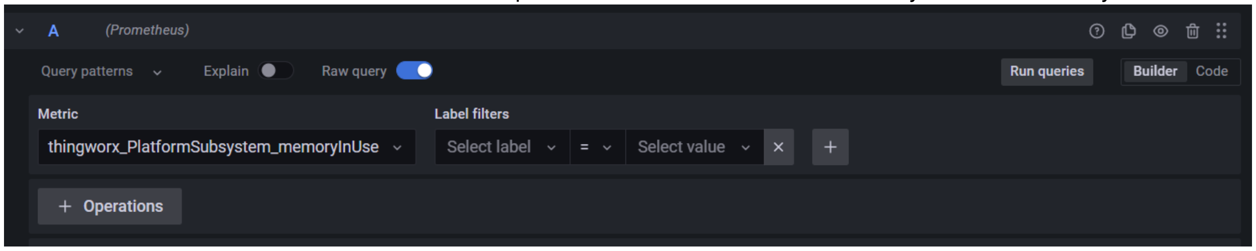


Prometheus
Prometheus | http://localhost:9090

Now both targets should appear within Grafana, with their metrics showing up throughout the Grafana UI. This data source is what allows for the building of monitoring dashboards.

How to Build a Dashboard

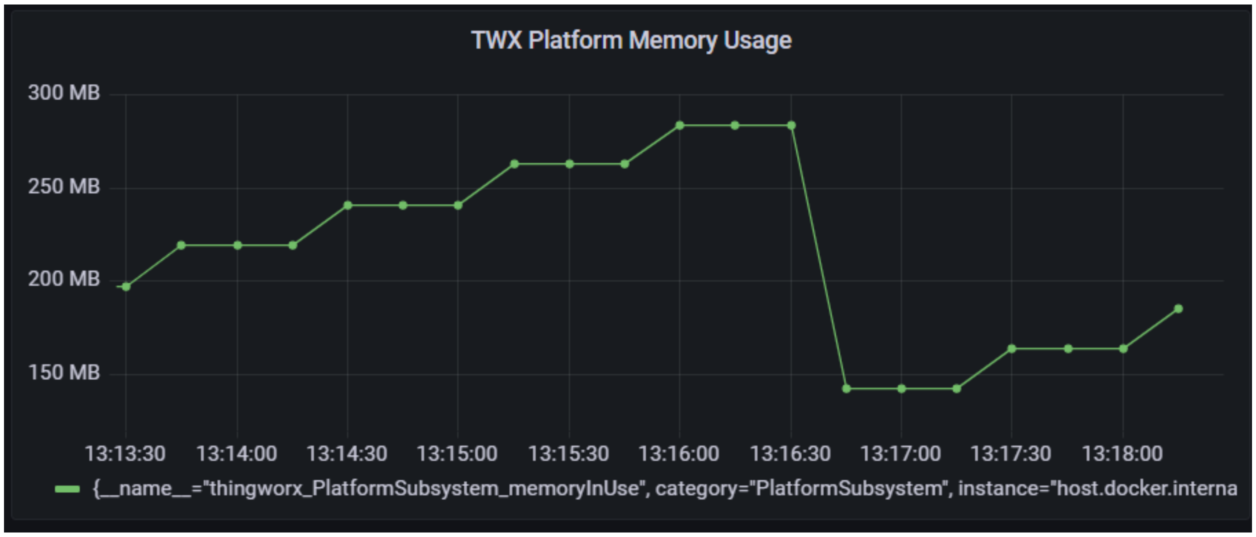
To add a panel which monitors some component of the ThingWorx application to a dashboard in Grafana, click to add a new panel. Under "Metrics" in the box at the bottom of the screen, select what ThingWorx metrics you wish to monitor (type "thingworx" in the search box to see them all). For example, select the Platform Subsystem memory in use:



Metric: thingworx_PlatformSubsystem_memoryInUse

Label filters aren't necessary, though you may want to sort by instance if you are monitoring multiple ones with the same dashboard. You may also want to take some time to format the Y axis, which by default will show in bytes. Go to the formatting panel on the right side and scroll down to the section called "Standard options". For the Unit dropdown, start typing "data" and then select "bytes (SI)". This will automatically determine if the bytes you've provided should really print as MB or GB based on how large the numbers are.

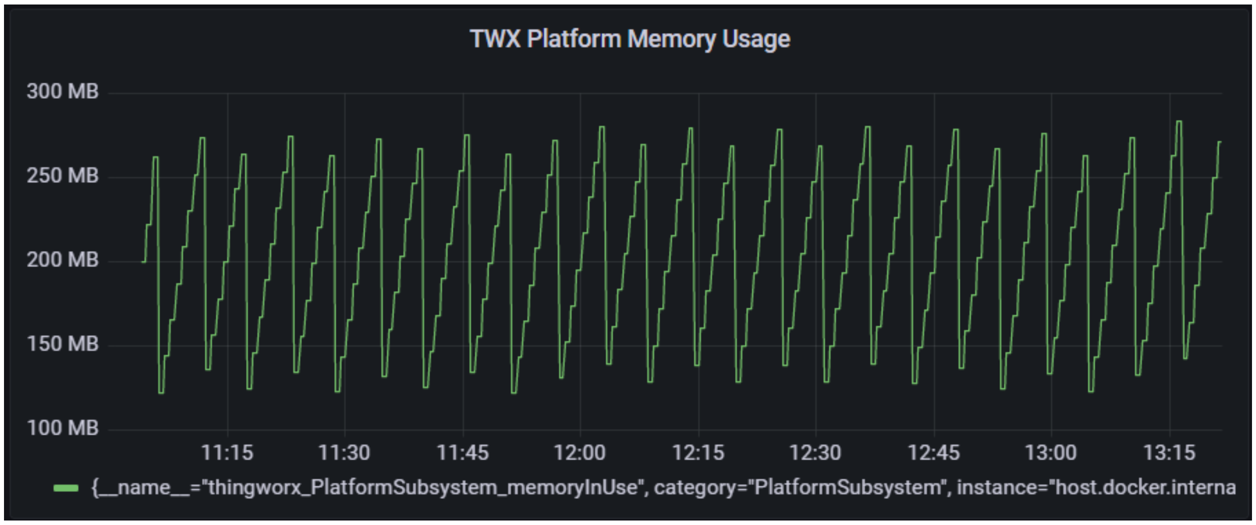
Rename the panel, modify it in any other way desired, and then click Apply (last 5 minutes):



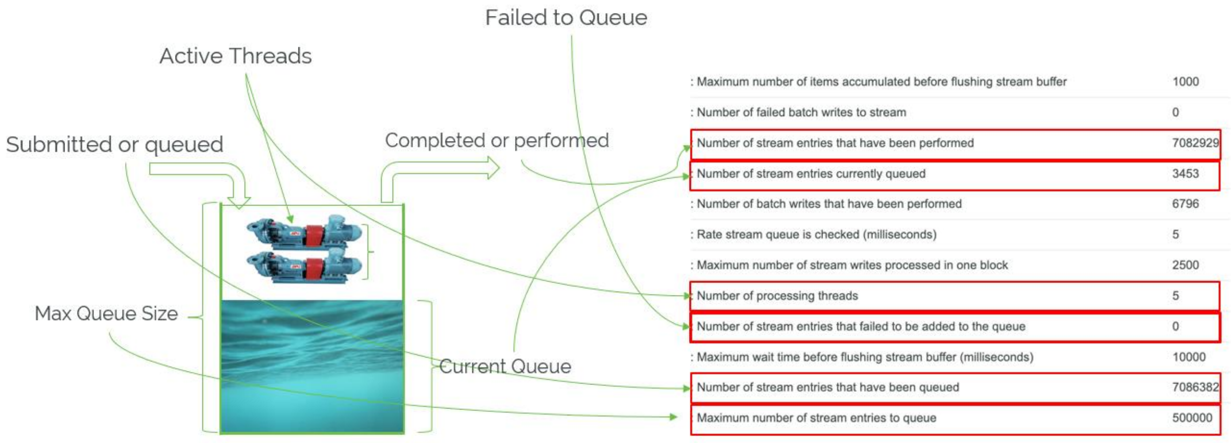
🟩 {__name__="thingworx_PlatformSubsystem_memoryInUse", category="PlatformSubsystem", instance="host.docker.interna

Once you add the panel, you can watch the memory usage as it is scraped by selecting the refresh option (10s or 30s, whatever makes sense based on your scrape interval).

The viewing window is stored in the URL, so that you can generate a report for a specific interval (like when a test was occurring), and then store that result or share it in a more compact way: http://localhost:3000/d/nleucPv4k/thingworx-monitoring?orgId=1&**from=1668528038732&to=1668536503953** (absolute timestamps):



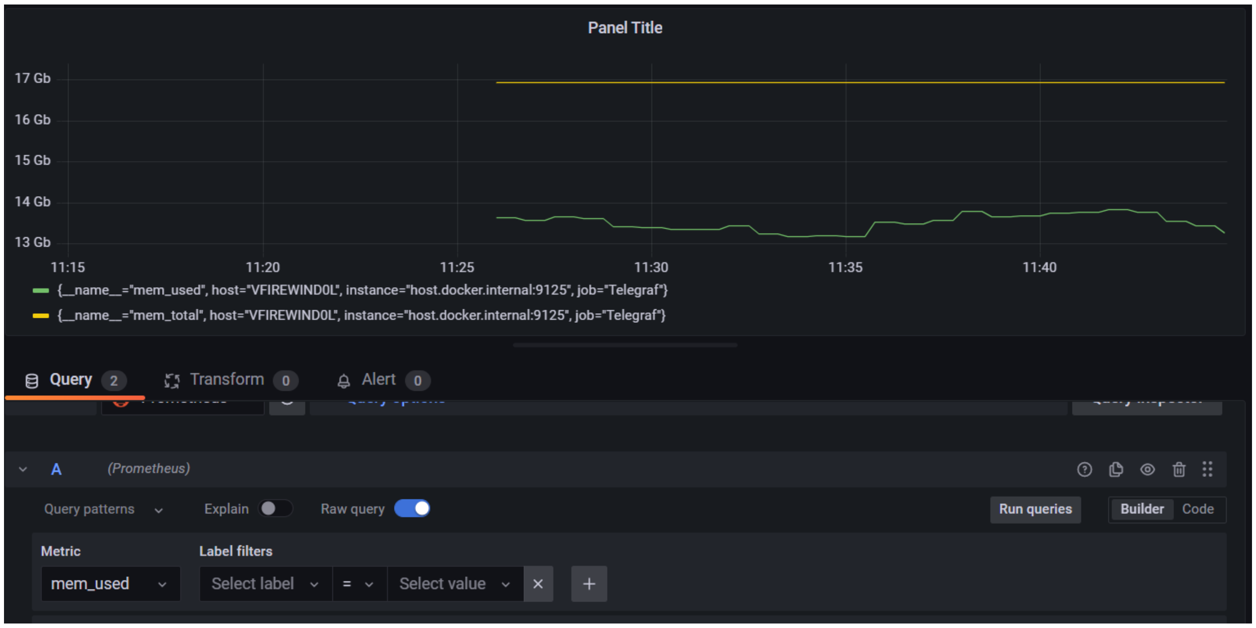
Dashboards are just collections of panels which report on all of the various metrics of performance and stability that exist for single components of a system. This is because there can be quite a few metrics worth watching for each individual component. Most of the third-party tools come with their own dashboards, but the ThingWorx component is one which for now, requires some thought and creativity.



: Maximum number of items accumulated before flushing stream buffer	1000
: Number of failed batch writes to stream	0
Number of stream entries that have been performed	7082929
Number of stream entries currently queued	3453
: Number of batch writes that have been performed	6796
: Rate stream queue is checked (milliseconds)	5
: Maximum number of stream writes processed in one block	2500
Number of processing threads	5
Number of stream entries that failed to be added to the queue	0
: Maximum wait time before flushing stream buffer (milliseconds)	10000
Number of stream entries that have been queued	7086382
Maximum number of stream entries to queue	500000

Consider your use case carefully and look over the various subsystems contained within ThingWorx. Each part of an application is localized to specific subsystems, and some are more business critical than others. What will go at the top of your dashboard? Add rows, add panels per row, and see what the many choices are for watching your system.

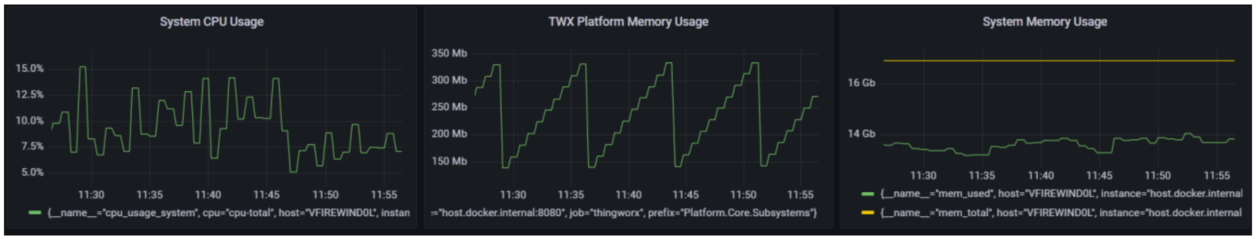
Don't forget that with Telegraf running, VM or machine usage metrics are also available for display on a dashboard. Things like overall CPU and Memory usage are critical to determining the health of a system, as we have demonstrated in our own reasoning in past benchmarks and scale tests. You can create a panel to monitor the mem_used versus mem_total, like so:



{__name__="mem_used", host="VFIREWINDOL", instance="host.docker.internal:9125", job="Telegraf"}
{__name__="mem_total", host="VFIREWINDOL", instance="host.docker.internal:9125", job="Telegraf"}

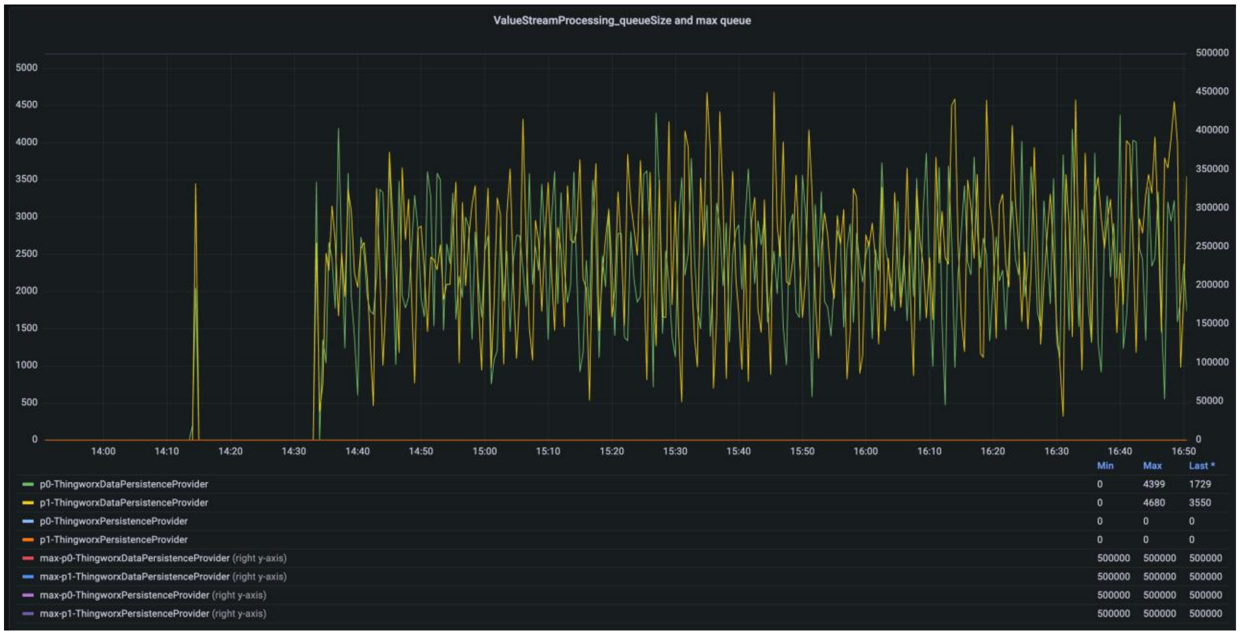
Raw query: on
Metric: mem_used
Label filters: Select label = Select value

Another metric from Telegraf worth adding is the CPU usage, which should be given "percentage" for the units and which needs a label filter of `cpu = cpu-total`. If we do some resizing and drag-and-dropping, then we now have the first row of a dashboard:



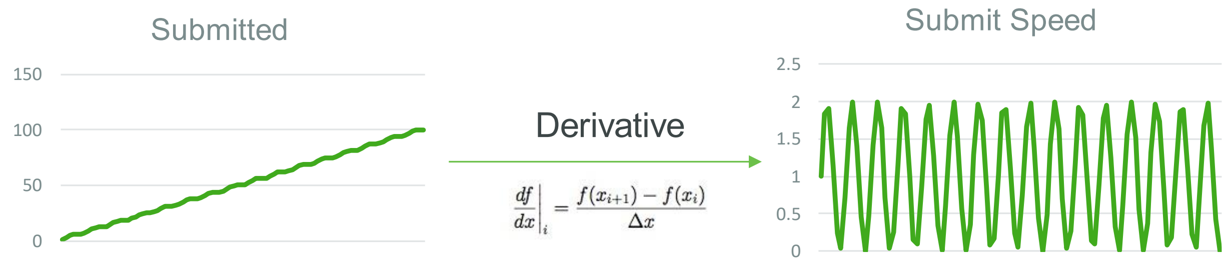
See how the Platform usage climbs steadily and is purged in a cycle? That is the Java Garbage Collection mechanism, and it's important to remember to leave room for spikes on top of those peaks. Data can also be calculated or processed in some way to make it more useful for determining system health and stability.

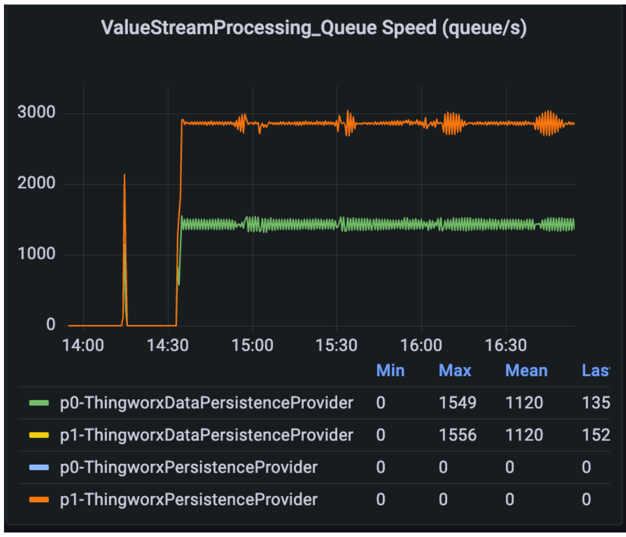
The data in the picture below uses the formula `submitted = completed + number queued + number failed`. It shows the current queue on the left Y-axis and the max queue on the right (since the two numbers usually are drastically different). It looks pretty, but it doesn't really tell us much about the system in this format, so let's do some math and find a representation that is a bit more helpful.



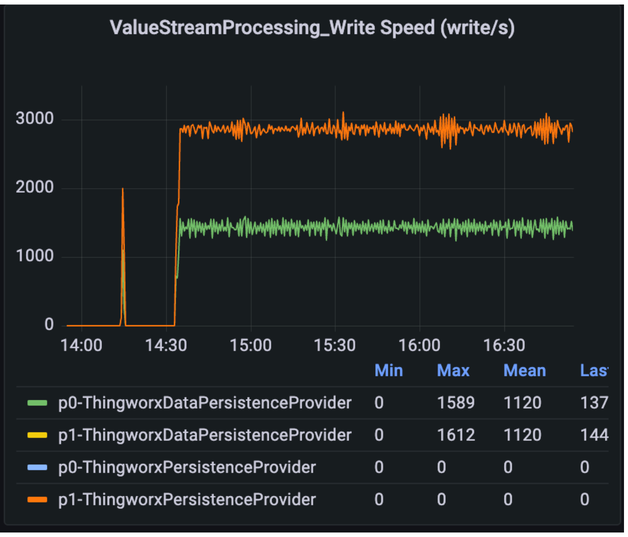
	Min	Max	Last *
p0-ThingworxDataPersistenceProvider	0	4399	1729
p1-ThingworxDataPersistenceProvider	0	4680	3550
p0-ThingworxPersistenceProvider	0	0	0
p1-ThingworxPersistenceProvider	0	0	0
max-p0-ThingworxDataPersistenceProvider (right y-axis)	500000	500000	500000
max-p1-ThingworxDataPersistenceProvider (right y-axis)	500000	500000	500000
max-p0-ThingworxPersistenceProvider (right y-axis)	500000	500000	500000
max-p1-ThingworxPersistenceProvider (right y-axis)	500000	500000	500000

Performing a “non-negative derivative” calculation over the submitted and the completed queue counts over time allows for us to look at the status of the queue *as a velocity*. When the “complete” speed appears behind the “submitted” speed for too long at a time, then that means the queue is filling up and will eventually result in data loss.


Submitted
Derivative
$\left.\frac{df}{dx}\right|_i = \frac{f(x_{i+1}) - f(x_i)}{\Delta x}$
Submit Speed



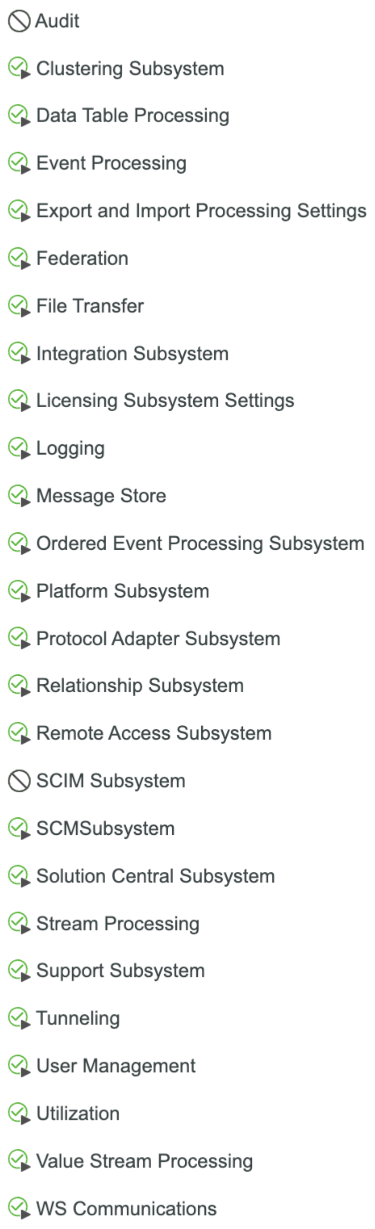
	Min	Max	Mean	Las
p0-ThingworxDataPersistenceProvider	0	1549	1120	135
p1-ThingworxDataPersistenceProvider	0	1556	1120	152
p0-ThingworxPersistenceProvider	0	0	0	0
p1-ThingworxPersistenceProvider	0	0	0	0



	Min	Max	Mean	Las
p0-ThingworxDataPersistenceProvider	0	1589	1120	137
p1-ThingworxDataPersistenceProvider	0	1612	1120	144
p0-ThingworxPersistenceProvider	0	0	0	0
p1-ThingworxPersistenceProvider	0	0	0	0

If we take this one step further and calculate the *average* of the submitted minus the completed *over time*, then we can actually **predict** approximately **when the queue will fill up**. This can then be displayed on a dashboard in Grafana, or used as the basis for an alert.

What to Monitor



- ⊘ Audit
- ✓ Clustering Subsystem
- ✓ Data Table Processing
- ✓ Event Processing
- ✓ Export and Import Processing Settings
- ✓ Federation
- ✓ File Transfer
- ✓ Integration Subsystem
- ✓ Licensing Subsystem Settings
- ✓ Logging
- ✓ Message Store
- ✓ Ordered Event Processing Subsystem
- ✓ Platform Subsystem
- ✓ Protocol Adapter Subsystem
- ✓ Relationship Subsystem
- ✓ Remote Access Subsystem
- ⊘ SCIM Subsystem
- ✓ SCMSubsystem
- ✓ Solution Central Subsystem
- ✓ Stream Processing
- ✓ Support Subsystem
- ✓ Tunneling
- ✓ User Management
- ✓ Utilization
- ✓ Value Stream Processing
- ✓ WS Communications

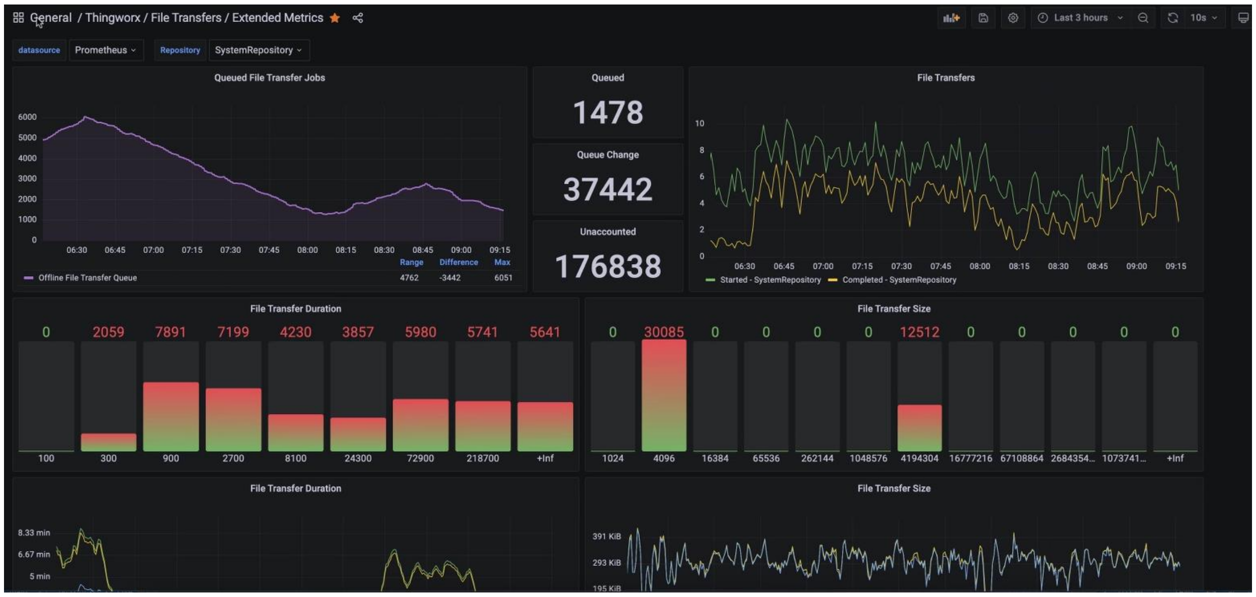
In addition to monitoring the system which ThingWorx runs upon, ThingWorx itself can easily be monitored down to the subsystems level by Prometheus due to the Metrics endpoint. Many applications have support built into the way they format the data for scraping, including the JVM (which exposes Prometheus-formatted metrics with the JMX Exporter) and the OS (which can use the Node Exporter or Telegraf for the same purpose). For these more generic components, there are popular community dashboards which can be downloaded and used in Grafana for data analysis and review.

For ThingWorx, there's different kinds of data to track: subsystem data (see the list on the right) and non-subsystem data. There's queue based versus non-queue data. These different metrics can collectively characterize the overall health of the application, depending on the use case.

For instance, if this is a system with very many connected devices, one metric which may be important to track is the number of total devices defined on the Foundation server vs. the number of devices which are currently connected. If there are relays involved, then many devices suddenly going offline can mean a relay has failed. Another example is if the system sizing depends on an assumption that there will only ever be a fraction of the total number of devices connected at a time. Use cases like these could be monitored easily by keeping track of the total vs. the number of connected devices.

Other common indicators of a healthy ThingWorx application might include the value stream and stream queues. These queues should fluctuate over time as the data is ingested and processed, but they should never be *growing in size*. If the stream queues are growing, then that means the data is writing to the queues faster than the queues can write to the database. Eventually, when the system runs out of resources to keep track of the queues, data will be lost. Having the stream information displayed in a chart can make it very easy to spot an upward trend in resource usage early on, which can catch a blockage or bottleneck that needs attention before it starts to affect the larger system in catastrophic ways later.

Memory usage information from the various subsystems might be something worth tracking, as well as the event queue. These can indicate that the business logic is functioning with room to handle spikes, and that the server has enough memory to service all three dimensions of an IoT application: the ingestion, the business logic and thing-based alerting, and the user experience and UI. If file transfers are a key part of the use case, then the number of concurrent transfers, the average speed of them, the size of the files, all of this kind of stuff can be tracked and charted in Grafana by making use of the ThingWorx metrics which automatically show up there once you import the Prometheus data source. A mature dashboard used for a production environment might look a little like this:



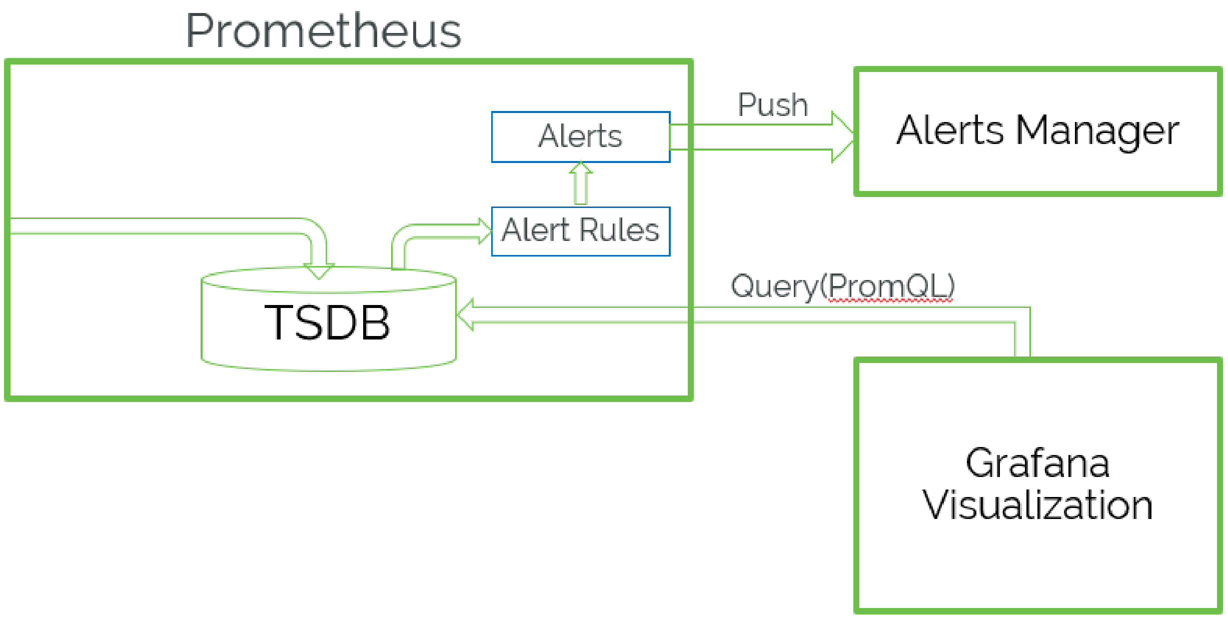
For further reading about subsystem monitoring, check out the Help Center.

How to Alert

The alerting mechanism built-in to Prometheus is incredibly easy to configure, so it might be tempting to generate tons of alert rules. However, remember that **the more noise a system makes, the harder it is for those monitoring that system to know when action is really required**. Playbooks which document how to respond to alerts, who to contact, how to act, and all the information necessary to handle an alert, should be created as an ongoing part of the DevOps process.

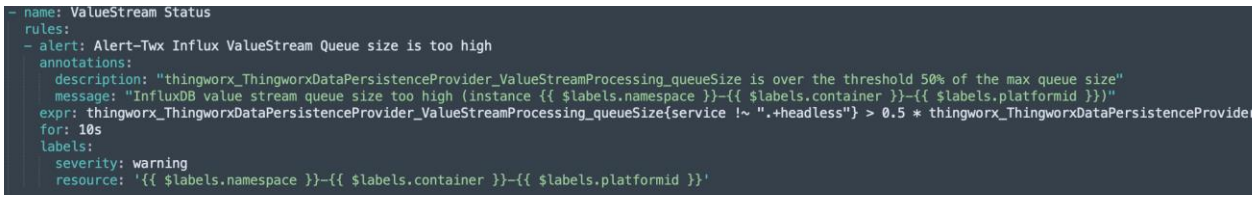
Alerts should fire with the right severity in the subject line, as well as all of the information about the issue that is currently known, presented in a concise way, so that whoever receives the alert starts thinking about the root cause sooner and recovers the system faster. Those who receive the alert should have the ability to facilitate its resolution, and know who is expected to react to any alerts which come in.

In the ThingWorx monitoring stack, Prometheus handles the alert rules and the generation of alerts, but alert filtering and delivery is managed in an external alerts manager.

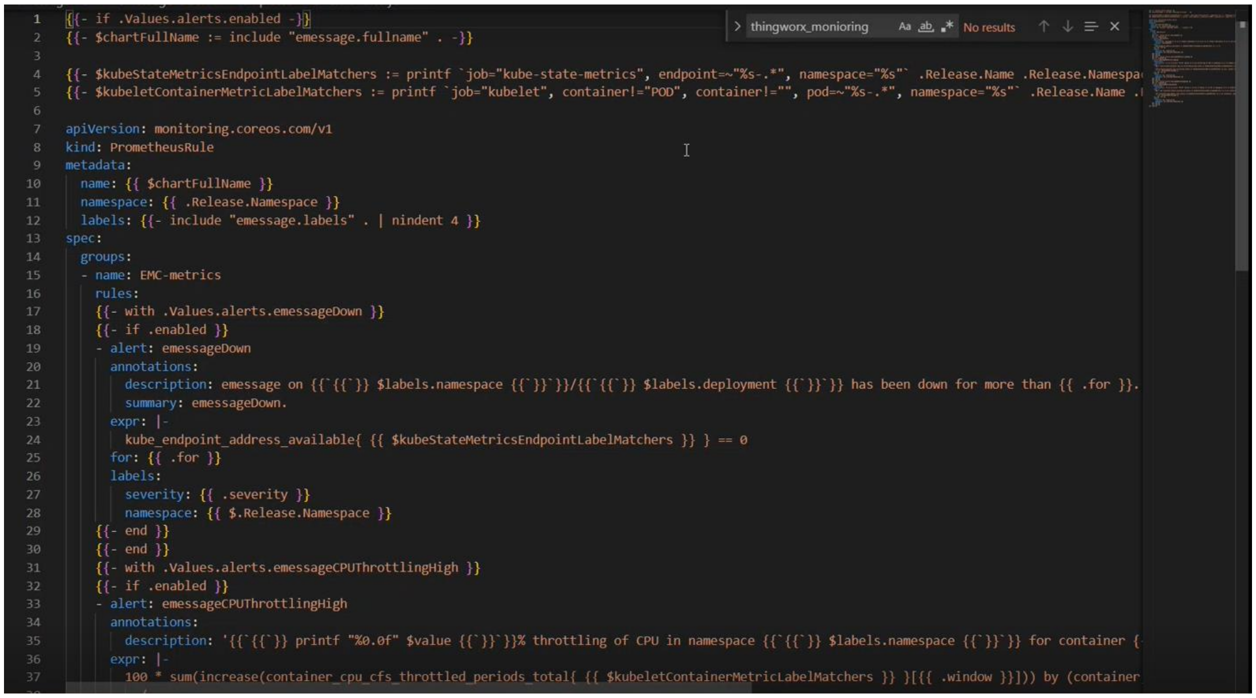


Generally, you want your alerts to follow a curve. If the current queue size exceeds 50% of the maximum, perhaps that isn't a huge deal, if the application catches up quickly. How long are spikes in queue processing expected to last? Perhaps if the queue size is over half-full for 10 seconds, 30 seconds, then that means the queue is falling behind and not catching up. Ok, so this might be a warning level alert. When does this become an error? Well, let's say the queue exceeds 90% of the max queue size. This might want to alert the moment it hits the mark. Now, farther along the curve, it may not take as long before data gets lost.

As the severity of the situation increases, the threshold for alerting should increase as well. That way when errors do alert, it is a sure thing that they require a response immediately. The alerts are then pushed into the "Alerts Manager" for delivery based on your management rules. The Alerts Manager may decide to withhold warnings altogether, or send them to a much smaller mailing list, whatever filtering helps to ensure the right people receive the right alerts, right when they need them.



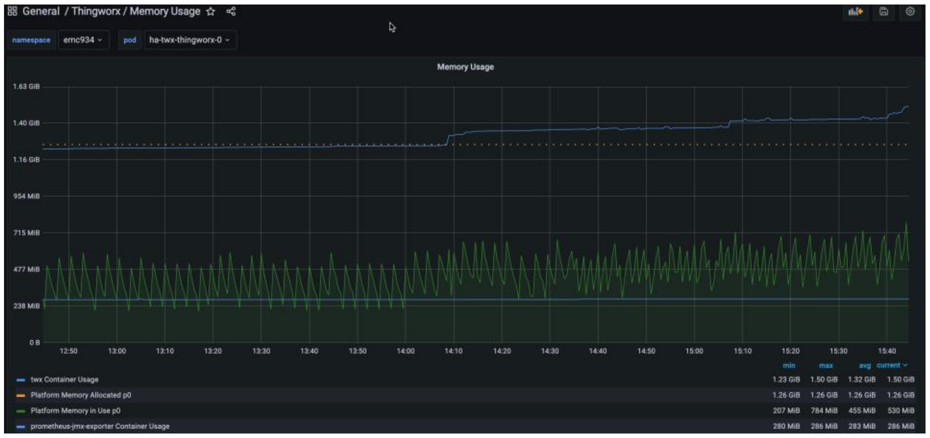
```
- name: ValueStream Status
  rules:
  - alert: Alert-Twx Influx ValueStream Queue size is too high
    annotations:
      description: "thingworx_ThingworxDataPersistenceProvider_ValueStreamProcessing_queueSize is over the threshold 50% of the max queue size"
      message: "InfluxDB value stream queue size too high (instance {{ $labels.namespace }}-{{ $labels.container }}-{{ $labels.platformid }})"
    expr: thingworx_ThingworxDataPersistenceProvider_ValueStreamProcessing_queueSize{service !~ ".+headless"} > 0.5 * thingworx_ThingworxDataPersistenceProvide
    for: 10s
    labels:
      severity: warning
      resource: '{{ $labels.namespace }}-{{ $labels.container }}-{{ $labels.platformid }}'
```



```
{{- if .Values.alerts.enabled -}}
{{- $chartFullName := include "emessage.fullname" . -}}

{{- $kubeStateMetricsEndpointLabelMatchers := printf `job="kube-state-metrics", endpoint=~"%s-.*", namespace="%s"` .Release.Name .Release.Namespa
{{- $kubeletContainerMetricLabelMatchers := printf `job="kubelet", container!="POD", container!="", pod=~"%s-.*", namespace="%s"` .Release.Name .

apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: {{ $chartFullName }}
  namespace: {{ .Release.Namespace }}
  labels: {{- include "emessage.labels" . | nindent 4 }}
spec:
  groups:
  - name: EMC-metrics
    rules:
    {{- with .Values.alerts.emessageDown }}
    {{- if .enabled }}
    - alert: emessageDown
      annotations:
        description: emessage on {{`{{`}} $labels.namespace {{`}}`}}/{{`{{`}} $labels.deployment {{`}}`}} has been down for more than {{ .for }}.
        summary: emessageDown.
      expr: |-
        kube_endpoint_address_available{ {{ $kubeStateMetricsEndpointLabelMatchers }} } == 0
      for: {{ .for }}
      labels:
        severity: {{ .severity }}
        namespace: {{ $.Release.Namespace }}
    {{- end }}
    {{- end }}
    {{- with .Values.alerts.emessageCPUThrottlingHigh }}
    {{- if .enabled }}
    - alert: emessageCPUThrottlingHigh
      annotations:
        description: '{{`{{`}} printf "%0.0f" $value {{`}}`}}% throttling of CPU in namespace {{`{{`}} $labels.namespace {{`}}`}} for container {
      expr: |-
        100 * sum(increase(container_cpu_cfs_throttled_periods_total{ {{ $kubeletContainerMetricLabelMatchers }} }[{{ .window }}])) by (container
```

In Conclusion, A Healthy Application…

Has stable memory usage that fluctuates predictably and doesn't grow over time. In a system experiencing mild issues, the memory starts to trend upward:



	min	max	avg	current
twx Container Usage	1.23 GiB	1.50 GiB	1.32 GiB	1.50 GiB
Platform Memory Allocated p0	1.26 GiB	1.26 GiB	1.26 GiB	1.26 GiB
Platform Memory in Use p0	207 MiB	784 MiB	455 MiB	530 MiB
prometheus-jmx-exporter Container Usage	280 MiB	286 MiB	283 MiB	286 MiB

If left unattended, systems like this may eventually experience outages. Finding the issue this early means there is even time to do some digging, debugging, taking of stack traces, and other such troubleshooting steps before the system must be restarted or recovered. That can really make the difference in identifying and resolving *before* there are real problems.

One metric which makes for good alerting is the total number of failed stream entries, which can indicate there's an issue writing to the database even before the queue has started to fill up. Other alerts may include warnings and errors based around percentages of memory used or queues filled, which depend on how long the queues take to fill up and how long the state has been at its increased usage.

Prometheus has all of the tools necessary to make this possible across a variety of infrastructures and use cases. Set it up on a local machine and poke around at what ThingWorx metrics are available to meet your monitoring needs.