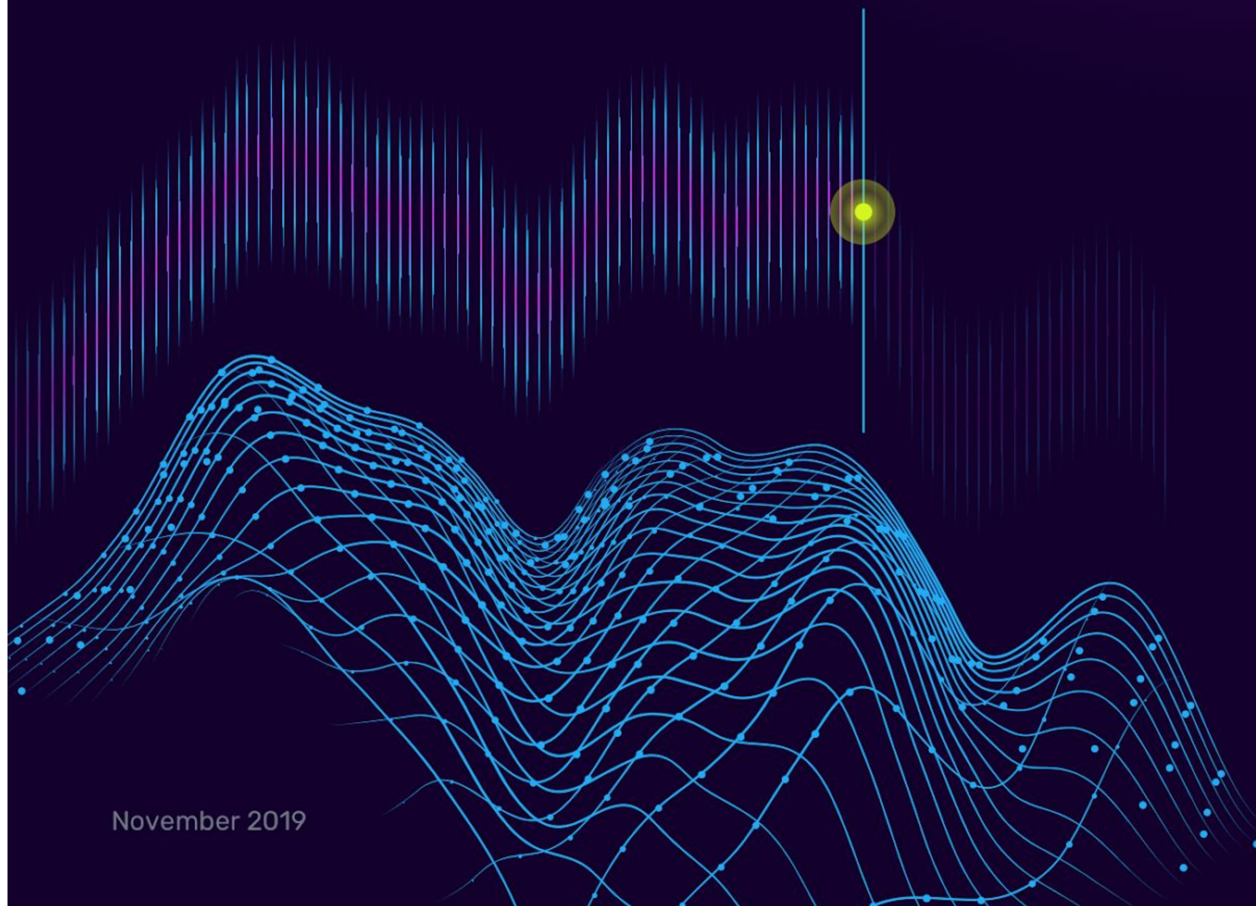
Monitor Full-Stack Kubernetes Metrics and Events at Scale with InfluxDB

TECHNICAL PAPER



November 2019

Kubernetes is being adopted as the backbone of modern IT applications. Therefore, monitoring Kubernetes, itself, has become increasingly important as applications are fragmented into microservices running within ephemeral containers. Part of the job of monitoring Kubernetes can be done with Prometheus collecting metrics, but that is not enough for a comprehensive monitoring. For instance, for a comprehensive monitoring approach, you would need to also be able to observe events, while keeping visibility into exposed metrics, and all from one view. Moving further towards a data-driven culture, your solution must scale in order to serve the vast variations of monitoring requirements from ITOps, SREs, developers, release engineers, security teams and business stakeholders.

Push and pull mechanisms, support for multiple data types, high precision, real-time stream ingestion and analytics, high availability, and long-term storage all matter when diving deeper into inspecting your Kubernetes application environment to address organization-wide observation needs. Most production environments don't have a single approach for application deployment and monitoring. Therefore, you should consider solutions that can handle variances, custom implementations, and business uniqueness, while facilitating the need for evolution over time.

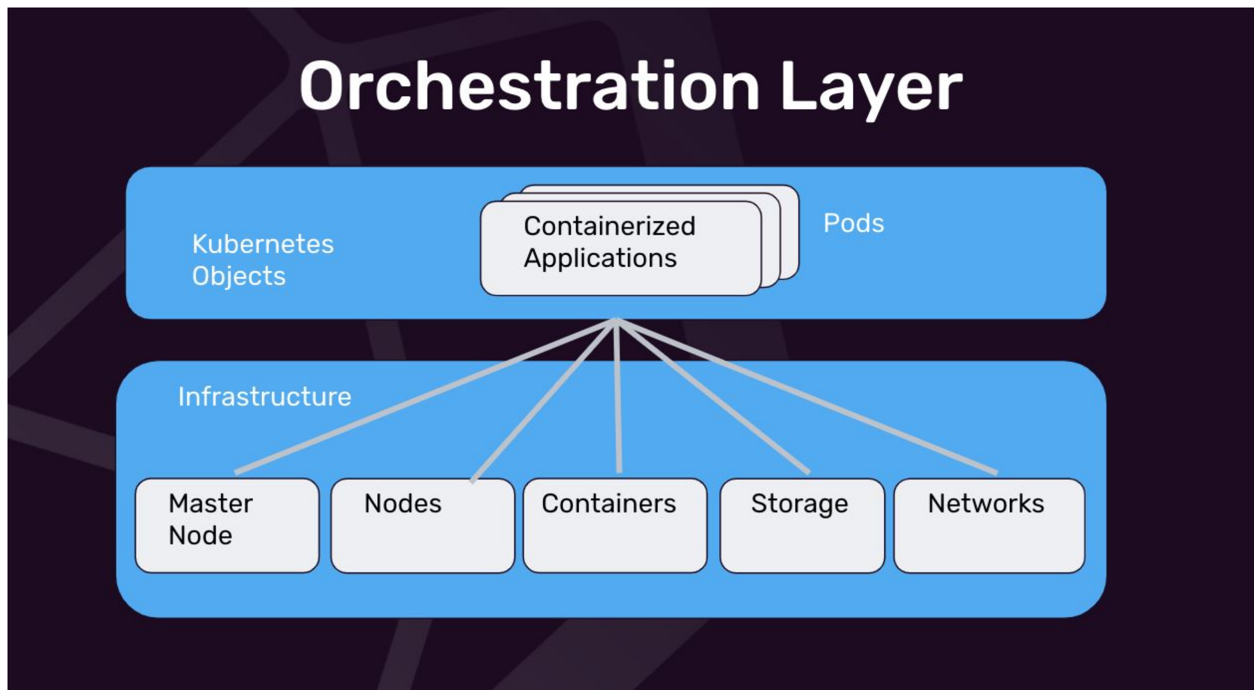
Here we discuss the importance of observing beyond nodes, containers and exposed Prometheus /metrics endpoints, to include Kubernetes state and custom application instrumentations. We'll discuss how to include events, such as logs, response and error codes, both for alerting and diagnosis. Also, what data collection methods could be used, what Kubernetes dashboards should be created, and why does InfluxDB time series platform — through real-time full-stack monitoring — provide the necessary platform foundation to address all monitoring and scalability needs.

Kubernetes overview

Kubernetes is a framework for running containerized workloads. It offers an open source platform which is primarily designed to automate deployment, upgrades/updates and scaling of application clusters. Describing Kubernetes using its own concepts, you can say that Kubernetes orchestrates clusters of containers that are encapsulated in pods and organized by services, labels and namespaces.

On a practical level, Kubernetes takes care of the scheduling and coordination of the resources for programs to run in a container cluster. Kubernetes is declarative, so you can tell how many replicas you want to run and leave the work to the orchestrator. It will dynamically schedule the necessary resources to run the replicas and continuously verify that they are actually running. Any deviation from the desired state will generate a scheduled task to correct it, bringing the cluster back to where it should be. As a result of this dynamic allocation of resources, utilization of infrastructure should experience more cost-effectiveness.

Portability is another important benefit of Kubernetes. Because containers carry all the system dependencies they need with them, they can be taken from one node to the other, or from one cloud provider to the other, without breaking the application. Additionally, Kubernetes pod encapsulation ensures that network and storage needs are assigned on demand — this works very well for stateless applications, although Kubernetes also supports stateful deployment options (StatefulSet). Applications running on Kubernetes are portable, and in an ideal scenario, they could go from the developers' work machine to the operations center.



Kubernetes architecture

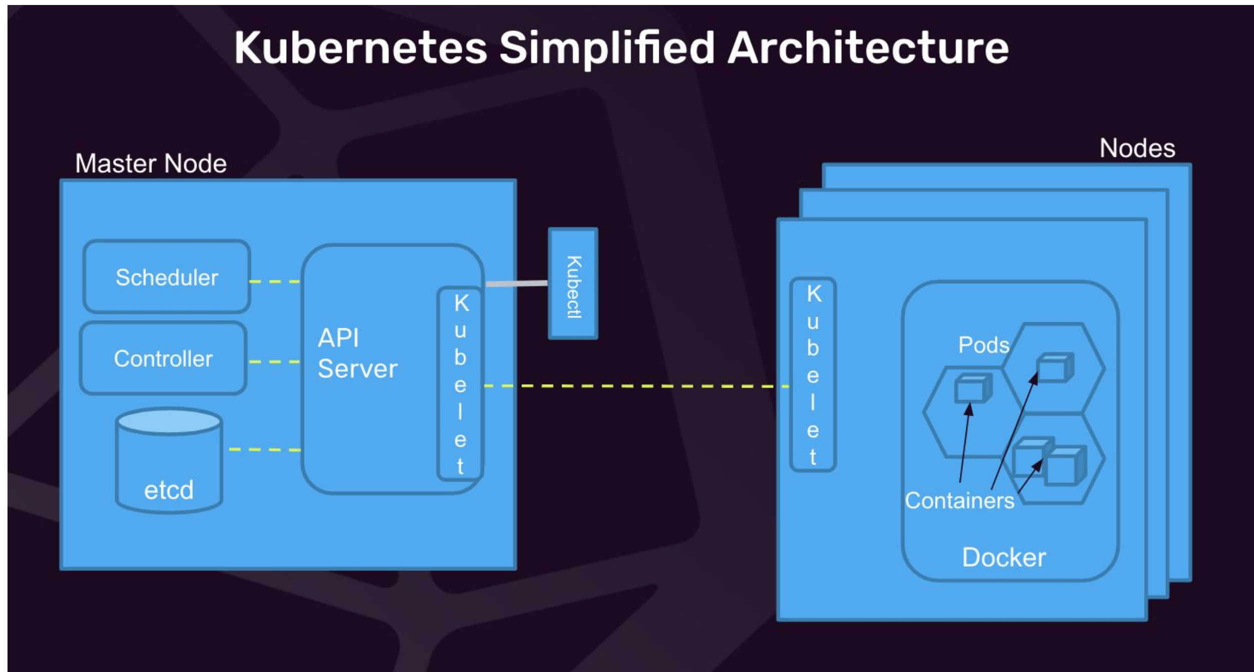
In a simplified view, Kubernetes is composed of a master node, worker nodes, and pod-encapsulated containers. The master node houses key elements of Kubernetes's architecture such as, control plane processes API server, scheduler and controller, and etcd database. On each node there are two control plane processes: kubelet and kube-proxy.

The API server works as a central communication hub, with all interactions done either via direct API calls (HTTP/gRPC), indirectly from the Kubernetes command-line client (kubectl), or via a web-based UI (Kubernetes Dashboard addon). Kubelet is the service used by the API Server to communicate with worker nodes.

Kubernetes clusters state is represented by Kubernetes Objects (specs and status) in a persistent database, etcd. Therefore, Kubernetes Objects are persistent entities in the Kubernetes system which uses these entities to describe: what containerized applications are deployed and running (and on which nodes); resources available to those applications; and policies around how to orchestrate them (such as restart policies, upgrades and fault-tolerance). The Control Plane processes run continuous control loops to manage those objects' states and respond to changes detracting the cluster from what has been declared.

In short, Kubernetes Objects are records of intent enforced via a controller logic, ensuring that deployed clusters run in the desired state. Examples of Kubernetes Objects are: Pod, Namespace, Deployment,

DaemonSet, ReplicaSet, StatefulSet, Cronjob, etc. For a complete list, you can use the following command `kubectl api-resources`.



Native metrics pipelines provide only snapshots; for monitoring, a time series platform is required.

Kubernetes orchestration acts based on monitored cluster resources in compliance with the "records of intent" (Kubernetes Objects). Therefore, monitoring is an integral part of Kubernetes's operations. Nonetheless, Kubernetes only consumes metrics that are needed to do what it does: decide when, where and what to spin up or destroy. How it has done in the past and how it will do in the future is not part of the resource assignment and scheduling logic. Kubernetes's "world" is ephemeral in essence. It takes nothing for granted; a continuous control loop keeps compliance with the desired status while assigning resources based on the latest snapshot of the infrastructure pool.

In summary, Kubernetes is obedient, agile and efficient, but because it only cares about the current "state of things". It doesn't warn, learn or predict in reference to the past or future. For that, other platforms for monitoring and analytics are needed.

Core metrics and monitoring pipelines

The growth of Kubernetes adoption is accompanied by the pressing need for holistic monitoring of containerized application environments, not only orchestration to continuously converge to the

declared state. From the observation of resource usage and workloads running inside containers to the communication networks/mesh services connecting them. All must work to maintain end-to-end performance and contribute positively to critical business indicators.

However, from a monitoring perspective, Kubernetes is quite narrow-focused. It collects data to make a point decision and moves on to the next sample interval for another point decision. The previous data is not maintained. Only the most up-to-date value is kept because Kubernetes' main goal is to comply with the declared desired state of Kubernetes Objects recorded in `etcd`. As a consequence, there is no native monitoring of records over time or behavior trends included in Kubernetes.

Fortunately, the Kubernetes community is ecosystem-friendly and didn't overlook the need for holistic monitoring. They understood that there is much about monitoring modern applications environments that goes beyond Kubernetes' orchestration scope, and therefore, would be better handled by other platforms.

Kubernetes contributes by generating plenty of metrics and logs. Although it stays focused on its core functionalities, consuming only fresh data used by control loops, it took the approach to provide mechanisms to facilitate expanded monitoring via other solutions. For instance, Kubernetes adopted a standard way to expose metrics, Metrics API which uses Prometheus /metrics endpoint format implemented in all its components.

From a metrics collection perspective, Kubernetes differentiates core metrics from monitoring metrics, and provides different pipelines.

- **Core metrics** serves metrics that Kubernetes understands and consumes to perform the functions of its internal components. For example, core metrics are used for scheduling which encompasses resource estimation, initial resources/vertical autoscaling, cluster autoscaling, and horizontal pod autoscaling.
- **Native monitoring metrics** is used to monitor Kubernetes applications by observing containers, pods, services, and the status of the Kubernetes Objects and orchestration of the overall cluster. Kubernetes provides detailed information about an application's resource usage at each of these levels via Resource Pipeline and information about the status of Kubernetes Objects via Kube State Metrics Pipeline. See below mechanisms in which data is exposed for collection:
    - **Resource Metric Pipeline** provides the amount of resources currently used by a given node or a given pod, and a limited set of metrics related to cluster components such as the HorizontalPodAutoscaler controller, as well as the kubectl top utility.

      These metrics are collected by metrics-server ((lightweight short-term in-memory store) and are exposed via the Metrics API (metrics.k8s.io API). Metrics-server discovers all nodes on the cluster and queries each node's Kubelet for CPU and memory usage. Kubelet fetches the data from cAdvisor and aggregates on the metrics-server.

      In summary, metrics-server is a cluster-level aggregator component that periodically scrapes metrics from all Kubernetes nodes served by Kubelet through Summary API (exposed by cAdvisor). The metrics are aggregated, stored in memory and exposed in Metrics API format (Prometheus format). The metric-server stores the latest values only.

- **Kube-state-metrics** service listens to the Kubernetes API server and generates metrics about the state of the objects. It is not focused on the system health or resources metrics, but rather on the status of Kubernetes Objects, such as deployments, nodes, pods, etc. Kube-state-metrics holds an entire snapshot of Kubernetes state in memory and continuously generates new metrics based on it. Kube-state-metrics are also exposed via Prometheus format ("/metrics" endpoints). See below a list of monitored objects:

    - CronJob
    - DaemonSet
    - Deployment
    - Job
    - LimitRange
    - Node
    - PersistentVolume
    - PersistentVolumeClaim
    - Pod
    - ReplicaSet
    - ReplicationController
    - ResourceQuota
    - Service
    - StatefulSet
    - Namespace
    - Horizontal Pod Autoscaler
    - Endpoint
    - Secret
    - ConfigMap

- **Third-party monitoring pipeline** allows solutions to augment monitoring scope beyond what is covered with Kubernetes pipelines to include:
    - **Additional monitoring targets:** such as master node, Kubernetes processes components (scheduler, controller, etcd, kubelet, kube-proxy, coredns, other addons...) and container workloads
    - **Additional methods:** besides Prometheus pull method, also support for pushed data events (app messages, logs).
    - **Additional types of data:** besides numeric metrics supported by Prometheus format (float64), includes also integer, string and boolean types particularly important for monitoring services, Error code, true/false output.
    - **Higher precision:** For certain types of applications (in finance and gaming markets, or for billing and forensics, for instance), higher precision, nanoseconds, could be required.

A third-party monitoring pipeline can be implemented in multiple ways in Kubernetes environments:

- **Monitoring agent and/or log collection daemon** on each Kubernetes cluster node, deployed as a DaemonSet to collect node resource and system data. The agent can also use the host network to reach containers inside the same node for pull or push monitoring.
- **Application sidecar agent**, deployed paired with the containerized application in a pod encapsulation. They share volume and network namespace. It is especially useful for custom instrumentation data (application needs unique plugins/monitoring agent configuration) and to isolate the impact of high volume metrics scraping and event pushing to the pod resources only, not the

node or entire cluster. A sidecar deployment would be also less impactful on network resources.
    - **Central agent** to scrape and collect cluster-wide metrics. If used as an aggregator, it can also receive various monitoring data from DaemonSets to batch it before sending the data to the monitoring data store. However, the choice to aggregate data or send it directly to the central store will depend on the monitoring volume and how much impact it has on the system and network resources. In general, it would be better to deploy local agents for more intense communications (a constant flow of pushed events or fine-grained polling intervals) and an aggregator agent to pipeline the data to the central repository.

A fundamental piece to support an effective monitoring pipeline is the high-availability central repository for visualization of all monitored data in dashboards, as well as retention of historic data for as long as it is needed.

InfluxData monitoring platform

InfluxData monitoring platform has InfluxDB purpose-built time series database as a critical element in its architecture. Monitoring Prometheus endpoints in Kubernetes is straightforward: scrape from API server the list of exposed "/metrics" endpoints, pull these exposed metrics, and send them to a time series database for storage, analysis and visualization. The time series database of choice could be Prometheus itself, or another TSDB that supports the Prometheus framework, like InfluxDB. However, InfluxDB is particularly interesting if scalability, high availability and ingestion of multiple collection methods (pull and push) as well as types of monitored data are required (metrics, events and logs).

Full stack real-time monitoring to stay out-of-trouble with great end-user experience

The reality is that /metrics endpoints are not the only monitoring that is required. Applications expose data in multiple ways and formats. Furthermore, the nature of the data to be captured also varies. For instance, a subtle event is best monitored if the event data is pushed at the time of event occurrence to a platform that supports real-time monitoring.

Prometheus pull model is not effective, or at a minimum cost-efficient, for monitoring events. On one hand, in order to reduce the risk of missing a spike or drop, the polling interval would have to be very fine — claiming a heavy toll on system resources. On the other hand, large intervals most likely will not catch subtle variations. A monitoring solution that can support both polling of regular metric data, as well as pushed events, is fundamental for holistic real-time monitoring.

Storage cost-effectiveness will protect your budget

Additionally, in order to keep up with the fragmentation and complexity of container environments, a lot of monitoring data must be ingested, analyzed, stored and presented in comprehensive dashboards and alerted in an actionable form. You can think of at least an order of magnitude increase in monitoring data volume with containerized micro-serviced applications. Therefore, monitoring can consume your resources and budget quite easily. There is a cost associated with every record collected and another cost to every record stored. And that is done continuously and cumulatively. That is where the performance of a purpose-built time series database platform makes a significant difference —

both as an enabler of monitoring at high ingest rates with real-time dashboarding, as well as a cost-effective store at scale.

One platform that scales to address all your needs

Facing the challenge to optimize both aspects of your monitoring implementation, in terms of performance and scalability, InfluxDB stands out as the preferred choice for a time series database platform, to which all monitoring data exposed by Kubernetes components and containerized applications can be sent and monitored from one central point. Nothing should go without leaving a trace of its impact, not even Kubernetes' short-lived pods. Accountability is fundamental for diagnosis and prevention.

In order to be suitable to be a central repository, InfluxDB scales horizontally in HA clusters. It supports sharding and clustering; therefore, it accommodates high-volume data ingestion from multiple locations and cloud deployments.

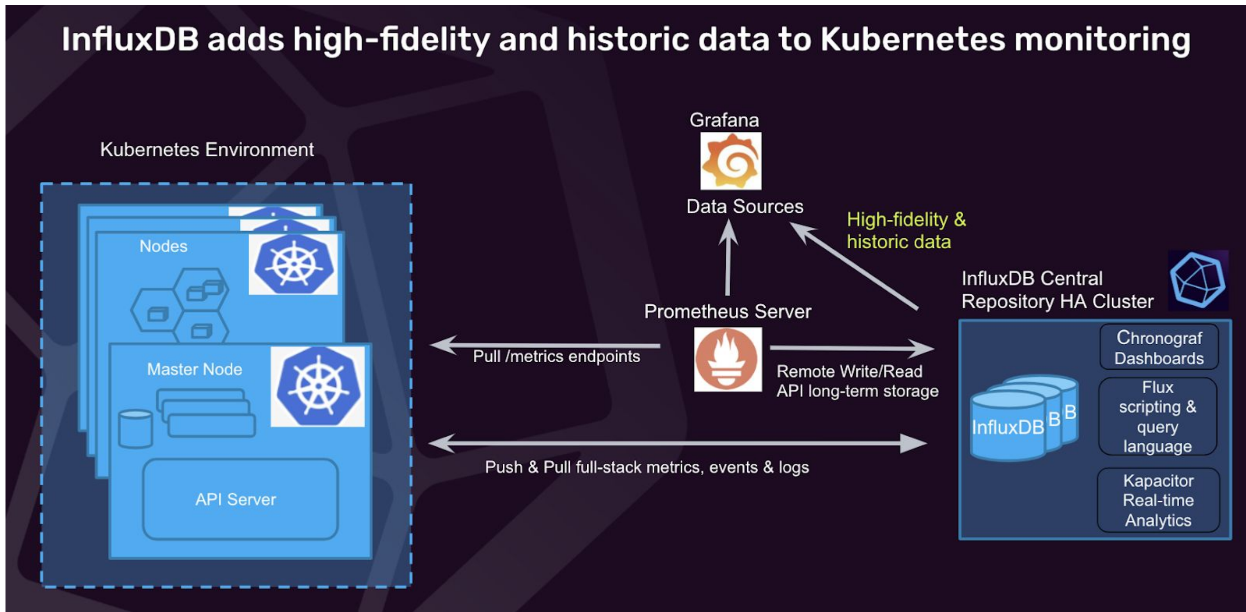
Other major InfluxDB differentiators, among monitoring solutions, are:

- Support for high-fidelity data (metrics, events and logs);
- Real-time streaming processing
- Cross-analytics between measurements through the InfluxDB data scripting and query language Flux.

These capabilities make the InfluxDB platform a modern choice for organizations looking to remain at the forefront of intelligent alerting, trend analysis for anomaly detection, and automation based on AIOps (among other frameworks that feed time series data).

Adding to the benefits is the fact that InfluxDB, like Kubernetes and Prometheus, is also an open source project. InfluxDB is used as a time series database and data source for many other solutions and open source projects, such as Grafana (advanced dashboarding). And indeed, many organizations may implement a combination of monitoring solutions to address specific aspects of their environment, and still keep InfluxDB as the long-term storage for collected timestamped data.

See diagram for Kubernetes with Prometheus monitoring and InfluxDB for long-term high-fidelity data:



Hands-on Kubernetes monitoring

In the process of migrating applications to Kubernetes, you must think beyond its benefits to also account for what else a successful migration to Kubernetes requires. Kubernetes certainly does an extraordinary job automating the replacement of trouble makers, bringing back absentees, getting rid of excess, among other orchestration functions. Nonetheless, in the process of continuously working to keep the clusters in the desired state, Kubernetes may be masking dangerous trends. For instance, its effectiveness in fixing undesired situations makes continued and intermittent faulty behavior invisible. Furthermore, Kubernetes monitoring solutions need to go through the same diligent consideration on key aspects of production environments, ensuring that they are suitable for:

- **Scalability:** horizontally scalable to handle the required data granularity and long-term storage from the whole Kubernetes cluster, but also scalable to handle collection and injection at very high rates (scraping or pushing) of an ever-increasing number of /metrics endpoints without collapsing.
- **High availability:** possibility to deploy in fully replicated instances, thereby removing single-point-of-failure situations
- **Long-term retention of monitored data:** durability of data stored for as long as needed
- **Multiple data types support:** numeric and non-numeric, such as integers, floats, strings, booleans. Important for ingestion of desired data without pre-processing to reformat it.
- **SLI, SLO and SLA support:** support for monitoring of both environment raw metrics as well as composing business KPIs.
- **Dashboard visualization:** visualization of data in audience-appropriate dashboard formats (histograms, percentiles, etc.)
- **Intelligent and scalable alerting engine:** support static as well as adaptive alerting
- **Real-time detection and quick diagnosis of issues:** support stream processing as well as pushed events. Also, provide a good means for debugging and root-cause analysis.
- **Security:** support for secured and authenticated monitoring

In terms of monitoring goals, you can think of:

- **Kubernetes core services and cluster metadata (kubelet API and kube inventory via kube-apiserver):** in general terms, health and status of a Kubernetes cluster
    - Make sure that Kubernetes components are functional and performant: API server, Metrics server, etcd, scheduler, controller, kubelet, kube-proxy, CoreDSN and other add-ons in use.
    - Make sure that Kubernetes platform is effective: monitor orchestration events such as pod destruction & creation, autoscaling, service and namespace creation, etc.
    - Examples of what to monitor:
        - Deployments status — desired, current, up-to-date, available, age
        - StatefulSets status
        - CronJobs execution stats
        - Kubernetes components health checks
        - Kubernetes Events
        - API Server requests
        - Etcd stats
- **Node-level, pod-level, and container-level metrics:** standard telemetry on Kubernetes hosts and services
    - Make sure that resources are available and at adequate capacity utilization: master node, nodes, pods and containers.
    - Examples of what to monitor:
        - Resource utilization per node (CPU, memory, disk, network bandwidth)
        - Node status (ready, not ready, etc.)
        - Number of pods running per node
        - Container health — status, restarts
        - Container resource utilization (CPU, memory, disk, network bandwidth)
        - Pods health — instances ready, status, restarts, eviction
        - Pod resource utilization (CPU and memory)
        - Mounted volumes stats
- **Application metrics from applications "contained" in pods:** monitor the container workload.
    - Make sure that applications running in containers are running without issues or performance degradation: monitor exposed metrics, pushed events and logging files.
    - What to monitor depends on the application. See below some examples:
        - Health / up/down check
        - HTTP requests (queue, latency, response code, etc.)
        - Outgoing connections (e.g. database connections)
        - Number of threads
- **High-fidelity data:** this is collected related to full application stack as well as Kubernetes cluster metadata
    - Make sure that no trend of harmful events goes undetected benefiting from Kubernetes automation: correlate metrics with Kubernetes objects state events.
    - Make sure that issues are quickly diagnosed and remediated: collect logs data that can be correlated with performance degradation.

What makes Kubernetes automation reliable is a comprehensive monitoring approach that keeps visibility over all layers of the application environment to observe behavior over time, and in case of

issues, provide quick access to data for expedited diagnosis and time to recovery. From that perspective, a more effective solution would integrate monitoring data from infrastructure, network monitoring, application performance and logs in one central platform, where a holistic view of the Kubernetes environment can be observed.

InfluxDB Kubernetes monitoring

InfluxData's time series platform InfluxDB supports pull and push of metrics, events and logs from multiple sources, including directly from Kubernetes nodes, pods, master node and from all exposed Prometheus /metrics endpoints. Additionally, it provides real-time stream analytics for efficient ingestion and effective alerting.

Collecting monitoring data with Telegraf

Telegraf is InfluxData's open source, plugin-based collecting agent with more than 200 plugins covering most common systems and applications found in IT infrastructure, networks and applications. It can be configured in listening and scraping modes, therefore supporting push and pull methods. This makes Telegraf a versatile agent.

Some plugins are specially crafted for Kubernetes monitoring:

- Telegraf Kubernetes Input Plugin: `[[inputs.kubernetes]]`collects data from Kubelet API (Node, Pod and Container Monitoring)
- Telegraf Kubernetes Inventory plugin: `[[inputs.kube_inventory]]`collects kube state metrics (nodes, namespaces, deployments, replica sets, pods, etc.):
- Telegraf Prometheus Input Plugin: `[[inputs.prometheus]]` Reads metrics from one or many Prometheus clients, scrape metrics from /metrics endpoints (from an array of URLs or K8s services, scrape Kubernetes pods for Prometheus annotations

Here are some baseline plugins that complement or supplement the monitoring breadth:

- Telegraf Internal plugin: `[[inputs.internal]]` collects its own systems metrics (cpu, disk, mem, etc.) and also aggregates agent stats on all Telegraf plugins (input and output). By doing so, Telegraf provides visibility in the monitoring pipeline, such as:
    - Gather_errors
    - Metrics_dropped
    - Metrics_gathered
    - Metrics_written
- Telegraf Docker plugin: `[[inputs.docker]]` collects docker container run-time events. Can be used as an alternative to cAdvisor /summary API
- Telegraf System plugin: The System input plugin gathers general stats on system load, uptime, and the number of users logged in. It is similar to the UNIX uptime command.

Here are some commonly used plugins for listening to event and log messages, as well as retrieving/parsing data from files:

- Telegraf HTTP Listener: `[[inputs.http_listener]]` listens for messages (in InfluxDB line-protocol) sent via HTTP POST. The intent of the plugin is to allow Telegraf to serve as a

proxy/router for the /write endpoint of the InfluxDB HTTP API. Can be configured for HTTPS (TLS Certs and Keys, and MTLS)

- Telegraf Syslog plugin: `[[inputs.syslog]]` listens for syslog messages transmitted over UDP or TCP or TLS,
- Telegraf StatsD plugin: `[[inputs.statsd]]` a special type of plugin which runs a backgrounded statsd listener service while Telegraf is running. StatsD messages are formatted as described in the original Etsy statsd implementation.
- Telegraf Kafka Consumer plugin:`[[inputs.kafka_consumer]]` reads from Kafka and creates metrics using one of the supported input data formats*****.
- Telegraf Tail plugin: `[[inputs.tail]]` "tails" a logfile and parses each log message. The plugin expects messages in one of the Telegraf Input Data Formats*****.
- Telegraf File plugin: `[[inputs.file]]`updates a list of files every interval and parses the contents using the selected input data format. Files will always be read in their entirety; if you wish to tail/follow a file use the tail input plugin instead.
- Telegraf Multifile plugin: `[[inputs.multifile]]` allows Telegraf to combine data from multiple files into a single metric, creating one field or tag per file. This is often useful creating custom metrics from the /sys or /proc filesystems.
- Telegraf Logparser plugin: `[[inputs.logparser]]` streams and parses the given logfiles. Currently, it has the capability of parsing "grok" patterns from logfiles, which also supports regex patterns.
- Telegraf Socket Listener plugin: `[[inputs.socket_listener]]` service input plugin that listens for messages from streaming (tcp, unix) or datagram (udp, unixgram) protocols. The plugin expects messages in the Telegraf Input Data Formats* .

***** Telegraf input data formats:

- InfluxDB Line Protocol
- Collectd
- CSV
- Dropwizard
- Graphite
- Grok
- JSON
- Logfmt
- Nagios
- Value, ie: 45 or "booyah"
- Wavefront

There are also implementations that push data straight to InfluxDB

- Fluentd InfluxDB plugin: `[[fluent-plugin-influxdb]]` buffered output plugin for fluentd and influxDB (when using fluentd as a log collector and InfluxDB as the central repository)

Telegraf monitoring ecosystem

IT Ops can fully benefit from a vast list of Telegraf plugins (200+) and client libraries (Python, Java, JavaScript/Node.js, GoLang, C# …) to meet their needs for a full-stack application environment and cloud monitoring solution. See below some other Telegraf plugins available for full-stack monitoring:

- **Network:** Ping plugin, HTTP Response plugin, SNMP plugin, JTI OpenConfig
- **System:** CPU plugin, Disk plugin, DiskIO plugin, Mem plugin, Net plugin, Netstat plugin
- **Application:** Apache HTTP Server plugin, Apache Tomcat plugin, Postgres plugin, MySQL,
- **Message Queue/Bus:** Kafka consumer plugin, AMQP
- **Cloud monitoring:** Google Cloud sub/pup plugin, Amazon Kinesis Consumer

InfluxData provides monitoring solutions for unified network monitoring solution comprising network availability, performance and traffic flow monitoring as well as for application performance monitoring solution (APM).

Telegraf deployment options

In Kubernetes cluster environments, Telegraf can be deployed as a DaemonSet in every node, as an application sidecar in pods or as a central collector.

Telegraf deployed as DaemonSet agent

For a native Kubernetes deployment, Telegraf can be installed as a DaemonSet in every Kubernetes node, in the same way that Prometheus Node_Exporter is. All agents deployed are configured from the same ConfigMap (same active plugins and configuration). The Daemonset will run a Telegraf Pod on every node in a cluster. Each of these nodes can be configured to pull Kubelet API stats (Kubernetes input plugin) and/or Kube-state-metrics (Prometheus input plugin). Telegraf can also collect all logging data in the node. Kube-state-metrics is run as its own Pod in one of the nodes in your cluster.

Custom instrumented metrics and events can also be sent to Telegraf DaemonSet. However, it is necessary to configure the application to send the instrumented monitoring data to the local host. And Telegraf http_listener plugin must be configured with a localhost address and port to be reachable from the container and receive pushed metrics.

Telegraf deployed as a central scraper

Telegraf can run on a cluster node in a non-DaemonSet deployment. It will be in its own pod. In this configuration, we call Telegraf a “scraper” since its main function is to discover and collect all exposed /metrics endpoints. Telegraf in a scraper deployment has its own configuration file (ConfigMap) with Prometheus plugin. The Prometheus plugin can be configured to scrape Pod annotations written into deployment configurations in Kubernetes. At regular intervals, Telegraf will consult the DNS service native to Kubernetes to grab metrics from the “/metrics” endpoints of services that match the configured Labels. This provides the same autodiscovery that Prometheus has offered. Telegraf can also pull regular telemetry and other service plugins.

Although a central scraper has advantages of reducing the number of monitoring agents instances, as the number of scraping endpoints increases, resulting from developers and operations engineers exposing metrics, such deployment topology will face scraping scalability issues and will add

maintenance burden to operations. Sidecar deployment can be used as an alternative to scale scraping while limiting the impact of application metrics exposure on the overall cluster scraping.

Telegraf deployed as a sidecar agent

As a principle, a good strategy for scraping shall neither increase the burden on operations nor impose a barrier to democratizing metrics to anyone who needs it. The answer to this apparent DevOps dilemma is actually quite simple and is in the heart of Kubernetes' containerized structure: you just need to contain (isolate) the impact of each exposed metrics. The scraper should be contained inside the pod of the service or workload, that it is going to scrape. The mechanism to implement this approach is called sidecar deployment.

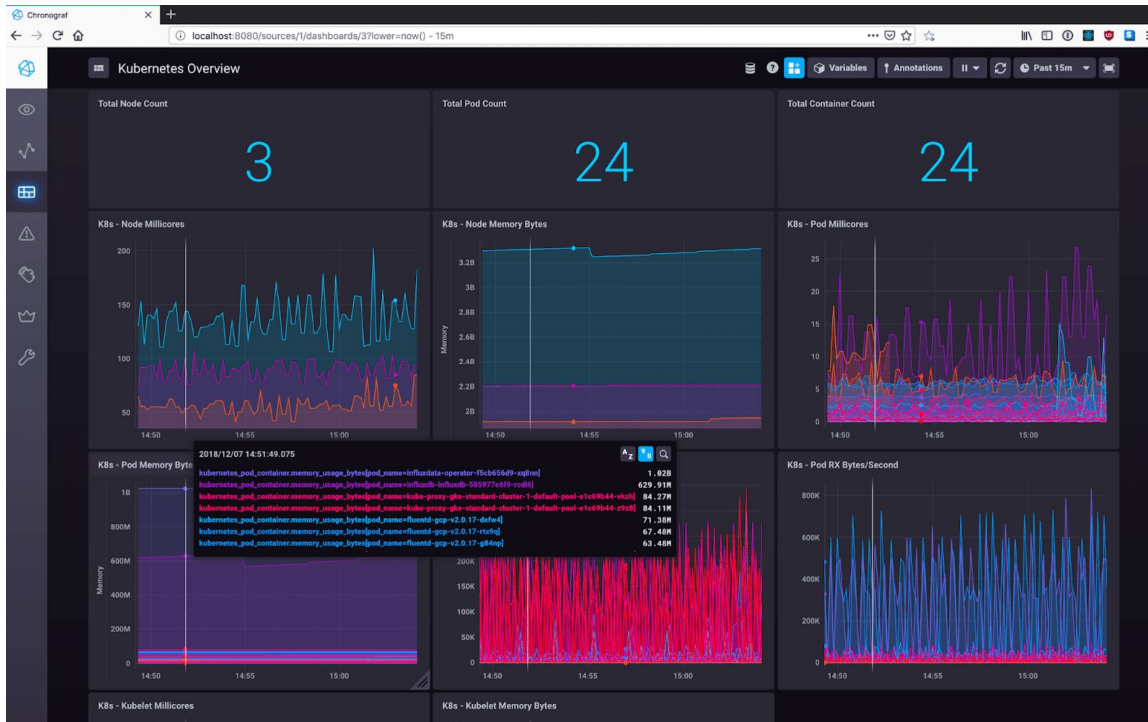
A central scraper has to pull the metrics data from every exposed endpoint within the polling interval, if a developer exposes a large number of endpoints for scraping, the scrapper may not be able to finish all the endpoints before a new polling interval starts. The burden to identify these situations and subdivide the monitoring into segments will only increase, requiring domain expertise and continuous interventions, i.e. "toil" work.

On the other hand, as a sidecar, Telegraf scrapes only the application (workload) running on the containers inside the pod without overtaking a central polling cycle. Situations when the scraper cannot finish polling all endpoints lead to metrics not to be collected, creating "monitoring blind spots". Additionally, Telegraf is in the same network as the containers inside the pod and will be sharing storage resources. This setup provides a much faster, safer and efficient means to send data from the app to the collector. For instance, the application running in the container could write events inside a socket (Telegraf Socket Listener plugin) or a file (Telegraf Tail plugin).

See below an overall Kubernetes Monitoring diagram with InfluxDB and various deployment options for Telegraf deployment.

InfluxDB Kubernetes dashboards

The InfluxData time series platform comes with pre-canned Kubernetes dashboards for baseline monitoring. See below Chronograf pre-canned Kubernetes dashboard with an InfluxDB datasource for Kubernetes nodes, pods and container monitoring.



Chronograf pre-canned Kubernetes dashboard

How InfluxData facilitates working with Kubernetes

InfluxData has added Kubernetes-specific capabilities to make it easier for you to work with Kubernetes:

- **Helm charts for faster Node Deployment** - kube-influxdb is a collection of Helm charts for the InfluxData TICK Stack to monitor Kubernetes with InfluxData.
- **Native Kubernetes Operators** - A Kubernetes Operator manages InfluxDB instances. It is built using the Operator SDK, which is part of the Operator Framework and manages one or more InfluxDB instances deployed on Kubernetes. InfluxDB Kubernetes Operator allows for InfluxDB to be deployed as a Kubernetes object, facilitating backup/restore operations.
- **Telegraf Kubernetes input plugins** - Telegraf Kubernetes Input Plugin, Telegraf Kubernetes Inventory plugin, Telegraf Prometheus Input Plugin.
- **Telegraf plugin for service discovery of Prometheus /metrics** - Telegraf Prometheus Input Plugin discovers and gathers metrics from HTTP servers exposing metrics in Prometheus format.
- **Telegraf, InfluxDB, and Grafana Kubernetes apps** - Kubernetes apps are prepackaged applications that can be deployed to Google Kubernetes Engine in minutes. The Telegraf InfluxDB and Grafana package (for metrics collection, storage, & visualization) is popular for monitoring the performance of your applications, infrastructure, and even gathering and displaying real-time analytics within your applications.

- **High availability (HA) and scalability of monitored data** - Large volume of metrics and events can be preserved in InfluxDB storage clusters allowing long-term policy retention together with high data granularity and high series cardinality.
- **Integration with Prometheus monitoring** - Kubernetes native monitoring is based on Prometheus local storage. Nonetheless, scalability and durability are beyond of Prometheus' scope. Prometheus project has a set of interfaces that allow integrating with remote storage systems which can provide scalability and durability assurances with preserving existing Prometheus PromQL/Grafana dashboards. InfluxDB integration with Kubernetes Prometheus monitoring is supported in three ways:

    1. **Remote Write API:** Prometheus can write samples that it ingests to InfluxDB in a standardized format.
    2. **Remote Read API:** Prometheus can read (back) sample data from InfluxDB in a standardized format.
    3. **PromQL Transpiler:** Allows reusing existing Prometheus Grafana dashboards with InfluxDB

InfluxData Cloud Platform

Kubernetes synthetizes the cloud paradigm. It embraces the openness, agility and extensibility required to keep up with the demands of developers, operations engineers, managers and users of applications. So does InfluxData. You can start your free Kubernetes monitoring in the InfluxData cloud platform with a free tier and all the facilities listed above that provide the fastest time to awesome!

InfluxData's open, agile and extensible monitoring platform keeps you ahead of the game as you track SLIs, achieve SLOs and comply with SLAs:

- Keep track in real-time of performance metrics and events of short-lived, ephemeral workloads
- Facilitate diagnosis and root-cause identification of issues and performance degradation in high-complexity fragmented environments
- Keep performance at scale while cost-effectively supporting the exponential increase of monitoring data volume
- Provide complete visibility from resource utilization to application and business performance indicators
- Provide scalability and high availability of monitoring data for long or permanent retention

InfluxData — monitor more than just Kubernetes

What and how you collect and monitor should not be limited by the tools used. On the contrary, tools must create opportunities to expand visibility horizons into your infrastructure, networks and applications. InfluxData platform gives you total flexibility for instrumentation, analysis and visualization of your entire environment. It has, at its core, InfluxDB, which is purpose-built for time series to empower developers and enterprises to accomplish more — and with more efficiency.

With a motivated global developer community, InfluxData is continuously pushing the edge of time series processing and storage embracing the cloud demands and challenges for applications. InfluxData delivers a solution that meets the requirements of even the most sensitive and demanding sectors like finance and service providers. The combination of Kubernetes with the InfluxDB times series platform provides the monitoring foundation to support organizations on their journey to growth and modernization.

InfluxDB documentation, downloads & guides

Download InfluxDB
Get documentation
Customer case studies
Join the InfluxDB community

799 Market Street
San Francisco, CA 94103
(415) 295-1901
www.InfluxData.com
Twitter: @InfluxDB
Facebook: @InfluxDB