Lucidworks

Fusion 5 Survival Guide

Draft v0.2

The purpose of this guide is to help you install, configure, and run Fusion 5 in production with high-availability on Kubernetes. It is assumed that you're already familiar with the basic installation and getting started concepts covered in: https://github.com/lucidworks/fusion-cloud-native.

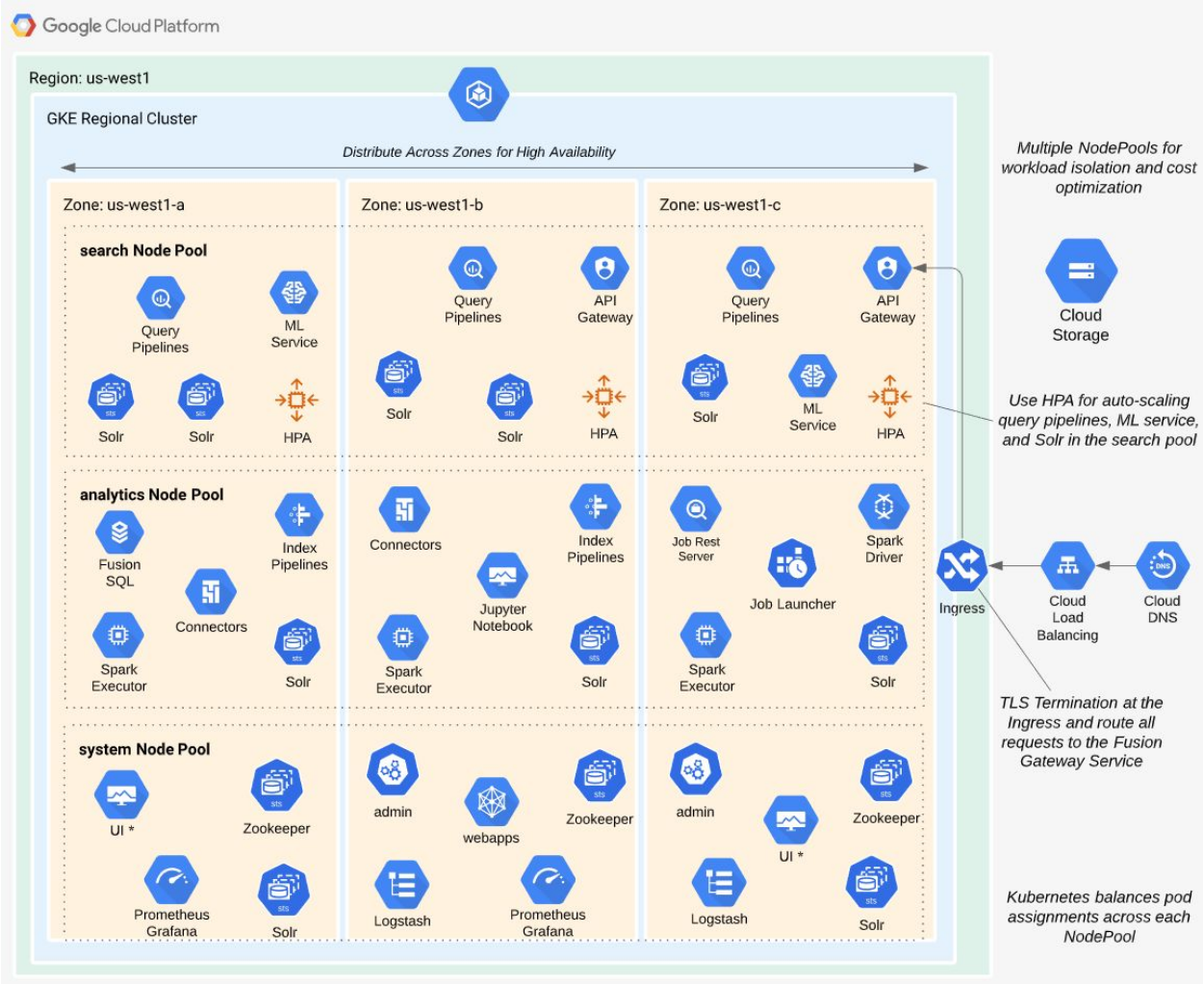
- Foundational Concepts 2
  - Which Kubernetes? 3
  - Multiple Zones for High Availability 4
  - Overview of Fusion Microservices 4
  - Ingress, TLS Termination, API Gateway 7
  - Stateless Sessions with JWT 8
  - Workload Isolation with Multiple Node Pools 9
  - High Performance Query Processing with Auto-scaling 10
- Planning Your Fusion Install 12
  - Prerequisites 12
    - Install Helm v3 12
    - Kubernetes Namespace 12
    - Docker Registry 13
  - Custom Values YAML 13
  - Monitoring and Alerting with Prometheus and Grafana 14
  - Solr Sizing 16
  - Multiple Node Pools 18
  - Solr Auto-scaling Policy 19
  - Pod Network Policy 20
  - Install Fusion 5 on Kubernetes 20
- Day Two Operations 21
  - Helm Upgrade Script 21
  - Logging 22
  - Grafana Dashboards 22
  - Pod Affinity Rules 24
  - Multiple Replicas and Horizontal Pod Auto-scaling 25
    - Use Gatling to Run Query Performance / Load Tests 25
    - Register Warming Queries 26
  - Resource Limits 26

Spark Ops 27
Cluster Mode 27
Spark config defaults 27
Spark Job Resource Allocation 28
Number of Instances and Cores Allocated 28
Memory Allocation 28
Configuring credentials in the Kubernetes cluster 28
Configuring GCS credentials for Spark jobs 29
Configuring S3 credentials for Spark jobs 29
Configuring Azure Data Lake credentials for Spark jobs 30
Configuring credentials per job 31
GCS 31
S3 31
How to get logs for a Spark job 31
Pod cleanup 32
Spark History Server 32
Installing Spark History Server 32
Recommended Configuration 32
Other Configurations 33
Azure 33
AWS 34
Configuring Spark 34
Accessing The Spark History Server 35
Upgrades with Helm v3 35

Foundational Concepts

Figure 1 depicts a Fusion 5 cluster running in a single Kubernetes namespace for high availability.

Figure 1: Fusion 5 Running in GKE with Multiple Node Pools for Workload Isolation



Let's cover some of the most important concepts of the Fusion cluster depicted in Figure 1, including:

- Which Kubernetes?
- Multi-zonal Cluster for HA
- Overview of Fusion Microservices
- Ingress, TLS Termination, Fusion API Gateway
- Stateless Sessions with JWT
- Workload Isolation with Node Pools
- High Performance Query Processing with Auto-scaling

Which Kubernetes?

Although the Figure 1 depicts running in Google Kubernetes Engine (GKE), Fusion 5 runs on any modern version of Kuberentes (v 1.12 or newer). In fact, very little code in Fusion has any awareness of Kubernetes or Docker. The K8s ecosystem is moving very fast and infrastructure

providers are quickly offering some flavor of Kubernetes[1]. However, the beauty of Kubenetes is that applications like Fusion do not need to care about the underlying infrastructure. Although Lucidworks provides setup scripts for GKE, AKS, and EKS to help users get started with Fusion 5, do not take this to mean Fusion only runs on those flavors of Kubernetes. In general, as long as K8s is running, Fusion 5 can run on it.

Multiple Zones for High Availability

Fusion relies heavily on Zookeeper to maintain quorum, which implies you need at least 3 ZK pods in an ensemble. In the Fusion Helm chart, Zookeeper is deployed as a StatefulSet. With GKE, you can launch a regional cluster that distributes nodes across 3 availability zones[2]. With a multi-zone setup, your cluster can withstand the loss of one zone without experiencing downtime; you may experience degraded performance from losing ⅓ of your total compute capacity assuming your pods are distributed evenly across zones. The multi-zonal cluster also ensures higher availability for the Kubernetes control plane services.

Lucidworks provides affinity rules to ensure multiple pods per service get distributed across multiple zones, see Pod Affinity Rules later in this document.

When running in a multi-zone cluster, each Solr node has a **solr_zone** system property set to the zone it is running in, e.g. `-Dsolr_zone=us-west1-a`. We'll cover how to use the **solr_zone** property to distribute replicas across zones in the Solr Auto-scaling Policy section below.

Overview of Fusion Microservices

Table 1 lists the Fusion microservices deployed by our Helm chart. Recognize that Fusion is a complex distributed application composed of many stateful and stateless services designed to support demanding search-oriented workloads at high scale.

Table 1: List of Fusion Microservices

Microservice	Protocol	Deployment or StatefulSet	Node Pool Assignment	Autoscaling Supported	Description
admin	REST/HTTP	Deployment	system	Not required. Minimum of 1 but 2 pods are recommended for HA	Exposes endpoints for admin tasks, such as creating applications and running jobs.

[1] https://www.infoworld.com/article/3265059/10-kubernetes-distributions-leading-the-container-revolution.html
[2] If you're not running in GKE, check with your K8s provider on how to deploy a cluster across multiple zones.

admin-ui	Web	Deployment	system	Not required; only 1 pod should be sufficient for most clusters	Serves static Web assets for the admin UI.
auth-ui	Web	Deployment	system	Not required; only 1 pod should be sufficient for most clusters	Serves static Web assets for the login form.
connectors-classic	REST/HTTP	StatefulSet	analytics or system	Yes (CPU or custom metric)	REST service for supporting non-RPC connector plugins.
connectors-rest	REST/HTTP	Deployment	analytics or system	Not required; only 1 pod should be sufficient for most clusters	Routes REST API requests to connectors-classic and connectors-rpc.
connectors-rpc	gRPC	Deployment	analytics or system	Yes (CPU or custom metric)	gRPC service for managing SDK-based connector plugins.
devops-ui	Web	Deployment	system	Not required; only 1 pod should be sufficient for most clusters	Serves static Web assets for the DevOps UI.
indexing	REST/HTTP	Deployment	search or analytics depending on write-volume	Yes (CPU or custom metric)	Processes indexing requests.

insights	Web	Deployment	system	Not required; only 1 pod should be sufficient for most clusters	Serves the App Insights UI
job-launcher	REST/HTTP	Deployment	analytics	Not required; only 1 pod should be sufficient for most clusters	Configures and aunches the Spark driver pod for running Spark jobs
job-rest-server	REST/HTTP	Deployment	analytics	Not required; only 1 pod should be sufficient for most clusters	Performs admin tasks for creating and running Spark jobs.
jupyter	HTTP	Deployment	analytics	Not required; only 1 pod should be sufficient for most clusters	Jupyter notebook for ad hoc analytics and visualization.
logstash	HTTP	StatefulSet	system	Not required. Minimum of 1 but 2 pods are recommended for HA	Collects logs from the other microservices and either indexes into system_logs or ships them to an external service like Elastic
ml-model-service	REST/HTTP and gRPC	Deployment	search	Yes (CPU or custom metric)	Exposes gRPC endpoints for generating predictions from ML models.
proxy / api-gateway	HTTP	Deployment	search	Not required. Minimum of 1 but 2 pods are recommended for HA	Performs authentication, authorization, and traffic routing.
query	REST/HTTP	Deployment	search	Yes (CPU or custom metric)	Processes query requests.

rules-ui	Web	Deployment	system	Not required; only 1 pod should be sufficient for most clusters	Serves static Web assets for the Rules UI.
solr	HTTP	StatefulSet	At least 3 nodes in search, 2 in analytics, and 2 in system	Yes (CPU or custom metric)	Search engine.
spark-driver	n/a	single pod per job	analytics or dedicated Node Pool for Spark jobs	1 per job	Launched by the job-launcher to run a Spark job
spark-executor	n/a	one or more pods launched by the Spark driver for executing job tasks	analytics or dedicated Node Pool for Spark jobs	depends on job configuration; controlled by the spark.executor.instances setting	Executes tasks for a Spark job
sql-service	REST/HTTP and JDBC	Deployment	analytics	Not required; only 1 pod should be sufficient for most clusters	Performs admin tasks for creating and managing SQL catalog assets. Exposes a JDBC endpoint for the SQL service.
webapps	REST/HTTP	Deployment	system	Not required; only 1 pod should be sufficient for most clusters	Serves App Studio-based Web apps.
zookeeper	TCP	StatefulSet	system	No, you need to run 1,3, or 5 Zookeeper pods to ensure a quorum; HPA should not be used for scaling ZK	Stores centralized configuration and performs distributed coordination tasks.

Ingress, TLS Termination, API Gateway

All external access to Fusion services should be routed through the Fusion proxy service, which serves as an API gateway and provides authentication and authorization. The most common

approach is to setup a Kubernetes Ingress that routes requests to Fusion services to the proxy service as shown in the example ingress definition below. Moreover, it is also common to do TLS termination at the Ingress so that all traffic to/from the K8s cluster is encrypted but internal requests happen over unencrypted HTTP.

```
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    annotations:
      ...
    labels:
      ...
    name: <RELEASE>-api-gateway
    namespace: <NAMESPACE>
  spec:
    rules:
    - host: <HOSTNAME>
      http:
        paths:
        - backend:
            serviceName: proxy
            servicePort: 6764
          path: /*
    tls:
    - hosts:
      - <HOSTNAME>
      secretName: <RELEASE>-api-gateway-tls-secret
  status:
    loadBalancer:
      ingress:
      - ip: <SOME_IP>
```

If running on GKE or AKS, the setup scripts in the fusion-cloud-native repo provide the option to create the Ingress and TLS cert (using Let's Encrypt). Otherwise, refer your specific K8s provider's documentation on creating an Ingress and TLS certificate.

Stateless Sessions with JWT

The Fusion API gateway requires incoming requests to be authenticated. The gateway supports a number of authentication mechanisms, including SAML, OIDC, basic auth, and Kerberos. Once authenticated, the gateway issues a JWT and returns it in the "id" cookie. Client applications will get the best performance by using the "id" cookie (or JWT Authorization header) instead of using Basic Auth for every query request because hashing a password is CPU intensive and slow by design (we use bcrypt), whereas verifying a JWT is fast and safe to

cache. We show an example of this in the Query Load Tests with Gatling section below, including how to refresh the JWT before it expires.

All Fusion services require requests to include a JWT to identify the caller.

Workload Isolation with Multiple Node Pools

You can run all Fusion services on a single Node Pool and Kubernetes will do its best to balance resource utilization across the nodes. However, Lucidworks recommends defining multiple Node Pools to separate services into "workload partitions" based on the type of traffic a service receives. Specifically, the Fusion Helm chart supports three optional partitions: **search**, **analytics**, and **system**. Workload isolation with Node Pools allows you to optimize resource utilization across the cluster to achieve better scalability, balance, and minimize infrastructure costs. It also helps with monitoring as you have better control over the traffic handled by each node in the cluster.

As depicted in Figure 1, the **search** partition hosts the API gateway (aka proxy), query pipelines, ML model service, and a Solr StatefulSet that hosts collections that support high volume, low-latency reads, such as your primary search collection and the signals_aggr collection which serves signal boosting lookups during query execution. The search partition is where you want to invest in better hardware, such as using nodes with SSDs for better query performance; typically, SSDs would not be needed for analytics nodes. The services deployed in the search partition often have Horizontal Pod Autoscalers (HPA) configured. We'll cover how to configure the HPA for search-oriented services in the HPA Auto-scaling section below.

When using multiple node pools to isolate / partition workloads, the Fusion Helm chart defines multiple StatefulSets for Solr. Each Solr StatefulSet uses the same Zookeeper connect string so are considered to be in the same Solr cluster; the partitioning of collections based on workload and zone is done with a Solr auto-scaling policy. The auto-scaling policy also ensures replicas get placed evenly between multiple availability zones (typically 3 for HA) so that your Fusion cluster can withstand the loss of one AZ and remain operational.

The **analytics** partition hosts the Spark driver & executor pods, Spark job management services (job-rest-service and job-launcher), index pipelines, and a Solr StatefulSet for hosting analytics-oriented collections, such as the signals collection. The signals collection typically experiences high write volume (to track user activity) and batch-oriented read requests from Spark jobs that do large table scans on the collection throughout the day. In addition, the analytics Solr pods may have different resource settings than the search Solr pods, i.e. you don't need as much memory for these as they're not serving facet queries and other memory intensive workloads in Solr.

Tip: When running in GKE, separating the Spark driver and executor pods into a dedicated Node Pool backed by preemptible nodes is a common pattern for reducing costs while

increasing the compute capacity for running Spark jobs. You can also do this on EKS with spot instances. We'll cover this approach in more detail in the Spark Ops section below.

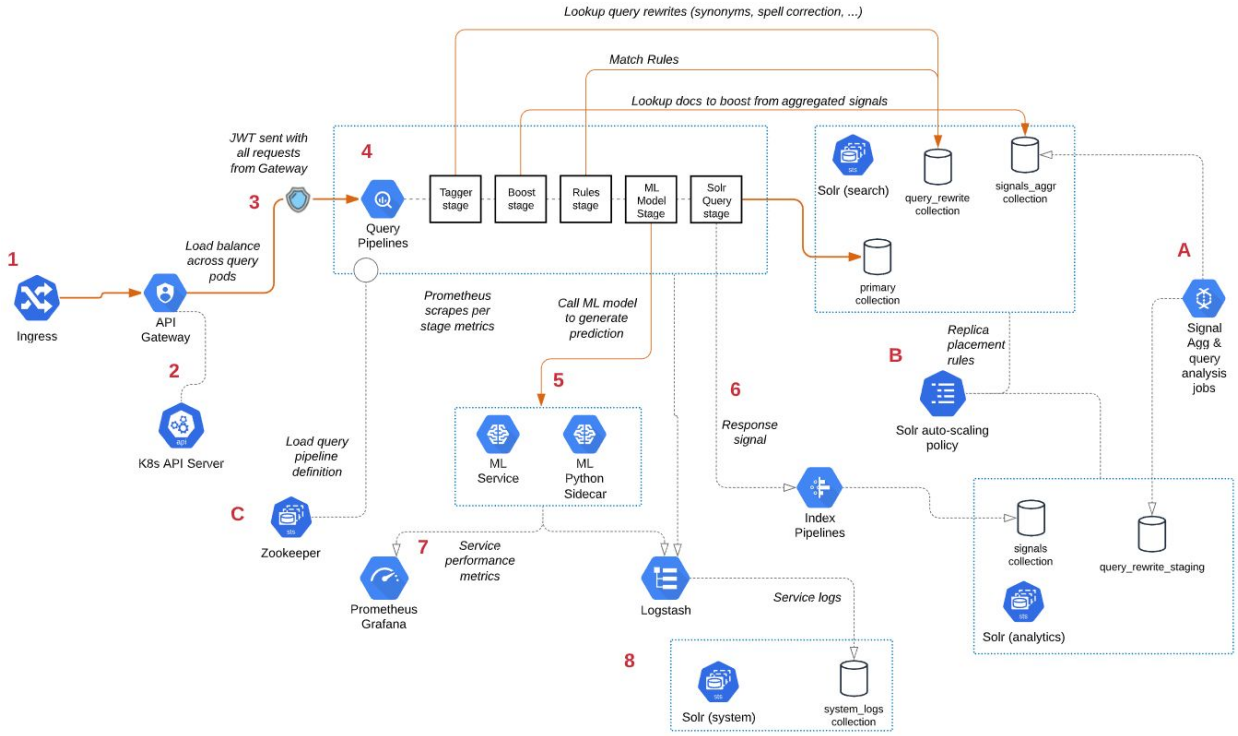
The **system** partition hosts all other Fusion services, such as the various stateless UI services (e.g. rules-ui), Prometheus/Grafana, as well as Solr pods hosting system collections like system_blobs. Lucidworks recommends running your Zookeeper ensemble in the system partition.

The analytics, search, and system partitions are simply a recommended starting point--you can extend upon this model to refine your pod allocation by adding more Node Pools as needed. For instance, running Spark jobs on a dedicated pool of preemptible nodes is a pattern we've had great success with in our own K8s clusters at Lucidworks.

High Performance Query Processing with Auto-scaling

To further illustrate key concepts about the Fusion 5 architecture, let's walk through how query execution works and the various microservices involved. There are two primary take-aways from this section. First, there are a number of microservices involved in query execution, which illustrates the value and importance of having a robust orchestration layer like Kubernetes. Second, Fusion comes well-configured out of the box so you don't have to worry about configuring all the details depicted in Figure 2.

Figure 2: Fusion Query Execution



At point **A** (far right), background Spark jobs aggregate signals to power the signal boosting stage and analyze signals for query rewriting (head/tail, synonym detection, and so on). At point **B**, Fusion uses a Solr auto-scaling policy in conjunction with K8s node pools to govern replica placement for various Fusion collections. For instance, to support high performance query traffic, we typically place the primary collection together with sidecar collections for query rewriting, signal boosting, and rules matching. Solr pods supporting high volume, low-latency reads are backed by a HPA linked to CPU or custom metrics in Prometheus. Fusion services store configuration, such as query pipeline definitions, in Zookeeper (point **C** lower left).

At point **1**, (far left), a query request comes into the cluster via a Kubernetes Ingress. The Ingress is configured to route requests to the Fusion API Gateway service. The gateway performs authentication and authorization to ensure the user has the correct permissions to execute the query. The Fusion API Gateway load-balances requests across multiple query pipeline services using native Kubernetes service discovery (point **2**).

The gateway issues a JWT to be sent to downstream services (point **3** in the diagram); this diagram is from the perspective of a request. An internal JWT holds identifying information about a user including their roles and permissions to allow Fusion services to perform fine-grained authorization. The JWT is returned as a Set-Cookie header to improve performance of subsequent requests. Alternatively, API requests can use the `/oauth2/token` endpoint in the Gateway to get the JWT using OAuth2 semantics.

At point **4**, the query service executes the pipeline stages to enrich the query before sending it to the primary collection. Typically, this involves a number of lookups to sidecar collections, such as the `<app>_query_rewrite` collection to perform spell correction, synonym expansion, and rules matching. Your query pipeline may also call out to the Fusion ML Model service to generate predictions, such as to determine query intent. The ML Model service may also use an HPA tied to CPU to scale out as needed to support desired QPS (point **5** in the diagram).

After executing the query the primary collection, Fusion generates a **response** signal to track query request parameters and Solr response metrics, such as numFound and qTime (point **6**). Raw signals are stored in the **signals** collection, which typically runs in the analytics partition in order to support high-volume writes.

Behind the scenes, every Fusion microservice exposes detailed metrics. Prometheus scrapes the metrics using pod annotations. The query microservice exposes per stage metrics to help understand query performance (point **7**). Moreover, every Fusion service ships logs to Logstash, which can be configured to index log messages into the system_logs collection in Solr or to an external service like Elastic (point **8**).

Armed with a basic understanding of the Fusion 5 architecture and how the various microservices interact, let's move on to planning your installation.

Planning Your Fusion Install

At this point, we assume you have a Kubernetes cluster running and have identified the namespace where you want to install Fusion. If you do not have a running cluster, you can use one of our setup scripts from the fusion-cloud-native repo to create one for you. However, we recommend you define the cluster using your own process as the clusters created by our scripts are intended for demo / getting started purposes.

NOTE: We will still use the setup_f5_*.sh scripts to perform the installation and upgrade tasks for Fusion, but will assume you've already created a cluster using the approved process for your organization.

Prerequisites

Please make sure you've installed the cloud provider specific command-line tools and have a working **kubectl** as described here: https://github.com/lucidworks/fusion-cloud-native. If you have not done so already, clone that repo. If you already cloned the repo, pull the master branch to get the latest changes.

Install Helm v3

Lucidworks recommends upgrading to Helm v3 for installing and upgrading Fusion on Kubernetes. Helm v3 no longer uses the server-side Tiller component.

On a Mac:
```
brew upgrade kubernetes-helm
```

For other OS, download from https://github.com/helm/helm/releases

Verify: `helm version --short`
```
v3.0.0+ge29ce2a
```

Kubernetes Namespace

Fusion 5 service discovery requires all services for the same release be deployed in the same namespace. Moreover, you should only run one instance of Fusion in a namespace. If you need multiple instances of Fusion running in the same Kubernetes cluster, then you need to deploy them in separate namespaces. If your organization requires CPU / Memory quotas for namespaces, you can start with a minimum of 12 CPU and 45GB of RAM (e.g. 3 x n1-standard-4 on GKE), but you will need to increase the quotas once you start load testing Fusion with production workloads and real datasets. Fusion requires at least 3 Zookeeper and 2 Solr nodes to achieve high-availability.

Before proceeding with the installation, collect the following information about your K8s environment:

- CLUSTER: Cluster name (passed to our setup scripts using the -c arg)
- NAMESPACE: K8s namespace where your Fusion cluster is running
- RELEASE: The unique Helm release name to use for Fusion, e.g. "f5"

Docker Registry

The Fusion Helm chart points to public Docker images on DockerHub. Your organization may not allow K8s to pull images directly from DockerHub or may require additional security scanning before loading images into production clusters. Work with your K8s / Docker admin team to determine how to get the Fusion images loaded into a registry that is accessible to your cluster. You can update the image for each service using the custom values yaml file discussed in the next section.

Custom Values YAML

Throughout this document, you will make changes to a custom values yaml file to refine your configuration of Fusion. You should keep this file in version control as you will need to make changes to it over time as you fine tune your Fusion installation; it will also be needed to perform upgrades.

The fusion-cloud-native repo contains a template for creating the custom values yaml file:

```
customize_fusion_values.yaml.example
```

Our setup scripts (**`setup_f5_*.sh`**) use the **`customize_fusion_values.yaml.example`** file as a template to create a custom values yaml for your cluster if it does not already exist. Consequently, you'll see some replaceable parameters in the file, such as `{NODE_POOL}`. However, since we're going to make a number of customizations in this file throughout this section, so let's create a working copy using the **`customize_fusion_values.sh`** utility script provided in the repo. You should name your custom values yaml file using the following convention:

```
<provider>_<cluster>_<release>_fusion_values.yaml
```

where `<provider>` is the K8s platform you're running on, such as gke, `<cluster>` is the name of your cluster, and `<release>` is the name you give to your Fusion release, such as f5. For instance, if you're running on GKE, your cluster is named "search", and you're using the "f5" release label, you would pass **`gke_search_f5_fusion_values.yaml`** to the script as shown below:

```
./customize_fusion_values.sh gke_search_f5_fusion_values.yaml \
  --num-solr 3 \
  --solr-disk-gb 100 \
  --prometheus true \
```

```
  --release f5
```

Tip: pass the `--help` parameter to see script usage details

After creating the custom values yaml file, take a moment to go through it to familiarize yourself with its structure and contents. You'll notice a separate section for each of the Fusion services listed in Table 1. Let's take a look at the configuration for the query-pipeline service to illustrate some important concepts about the custom values yaml (extra spacing added for display purposes only):

`query-pipeline:`	Service specific setting overrides under the top level heading
`  enabled: true`	Every Fusion service has an implicit enabled flag that defaults to true, set to false to remove this service from your cluster
`  nodeSelector:` `    cloud.google.com/gke-nodepool: default-pool`	Node selector identifies the label find nodes to schedule pods on
`  javaToolOptions: "..."`	Used to pass JVM options to the service

`  pod:`
`    annotations:`
`      prometheus.io/port: "8787"`
`      prometheus.io/scrape: "true"`
	Pod annotations to allow Prometheus to scrape metrics from the service

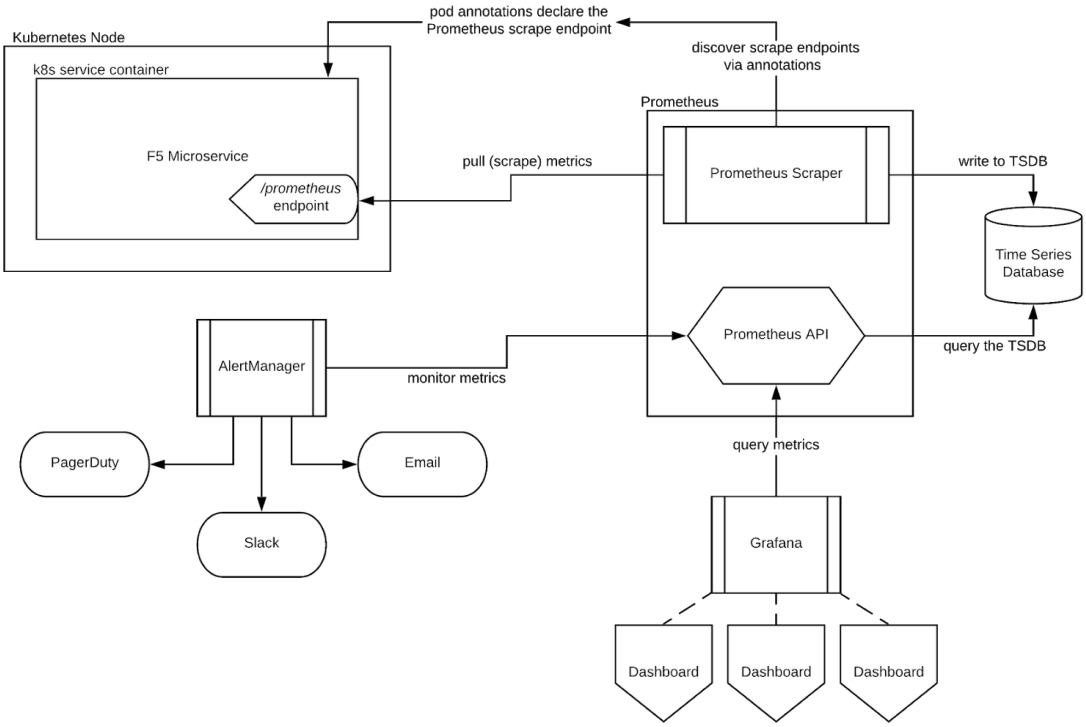
The **`--prometheus true`** option shown in the example above activates the Solr metrics exporter service and add pod annotations so that Prometheus can scrape metrics from Fusion services. Using this option also causes the script to create additional custom value yaml files for Prometheus and Grafana, e.g. `gke_search_f5_prom_values.yaml` and `gke_search_f5_graf_values.yaml`. You should also keep these files in source control. Of course if you don't want to use Prometheus, simply pass `--prometheus false` to the script.

Once we go through all of the configuration topics in this section, you'll have a well-configured custom values yaml file for your Fusion 5 installation. You'll then use this file during the Helm v3 installation at the end of this section.

Next let's see how to monitor the health and performance of Fusion services using Prometheus and Grafana.

Monitoring and Alerting with Prometheus and Grafana

Lucidworks recommends using Prometheus and Grafana for monitoring the performance and health of your Fusion cluster. The following diagram depicts how metrics work in a Fusion cluster.



Notice in the diagram that Prometheus pulls (aka "scrapes") metrics from Fusion services. Prometheus identifies which services to pull metrics from using pod annotations. For instance, to enable metrics for the Fusion **query-pipeline** service, you add the following pod annotations for the query service in the custom values yaml file (this is already be done for you in the `customize_fusion_values.yaml.example` template).

```
query-pipeline:
  ...
  pod:
    annotations:
      prometheus.io/port: "8787"
      prometheus.io/scrape: "true"
```

If you used the **`--prometheus true`** option when running the `customize_fusion_values.sh` script in the previous section, then you will have custom value yaml files for Prometheus and Grafana. You can use these files when installing Prometheus and Grafana; the settings should work for most Fusion clusters but you should review the contents of each to ensure the settings are suited for your needs, such as how long to keep metrics (default is 48h). For more information on how to configure these services, see:

- Prometheus: https://github.com/helm/charts/tree/master/stable/prometheus

- Grafana: https://github.com/helm/charts/tree/master/stable/grafana

We'll cover how to install the default Grafana dashboards provided in the fusion-cloud-native repo later in the document.

After the initial installation of Prometheus / Grafana, you'll need to use helm directly for making changes to those services as the setup_f5_*.sh scripts do not perform upgrades on Prometheus / Grafana. For instance, if you change a setting in the prom values yaml, you would do:

```
helm upgrade ${RELEASE}-prom stable/prometheus --namespace "${NAMESPACE}" \
  -f "$PROMETHEUS_VALUES" --version 9.0.0
```

Solr Sizing

When you're ready to build a production ready setup for Fusion 5, you need to customize the Fusion Helm chart to ensure Fusion is well-configured for production workloads.

You'll be able to scale the number of nodes for Solr up and down after building the cluster, but you need to establish the initial size of the nodes (memory and CPU) and size / type of disks you need.

Let's walk through an example config so you understand which parameters to change in the custom values yaml file.

```
solr:
  resources:
    limits:
      cpu: "7700m"
      memory: "26Gi"
    requests:
      cpu: "7000m"
      memory: "25Gi"
  logLevel: WARN
  nodeSelector:
    fusion_node_type: search
  exporter:
    enabled: true
    podAnnotations:
      prometheus.io/scrape: "true"
      prometheus.io/port: "9983"
      prometheus.io/path: "/metrics"
    nodeSelector:
      cloud.google.com/gke-nodepool: default-pool
  image:
    tag: 8.3.1
  updateStrategy:
    type: "RollingUpdate"
  javaMem: "-Xms11g -Xmx11g -Xmn4g
-XX:ParallelGCThreads=8"
  volumeClaimTemplates:
    storageSize: "100Gi"
```

Set resource limits for Solr to help K8s pod scheduling; these limits are not just for the Solr process in the pod, so allow ample memory for loading index files into the OS cache (mmap)

Run this Solr StatefulSet in the "search" node pool

Enable the Solr metrics exporter (for Prometheus) and schedule on the default node pool (system partition)

Configure memory settings for Solr

Size of the Solr disk

```
  replicaCount: 6

zookeeper:
  nodeSelector:
    cloud.google.com/gke-nodepool: default-pool
  replicaCount: 3
  persistence:
    size: 20Gi
  resources: {}
  env:
    ZK_HEAP_SIZE: 1G
    ZOO_AUTOPURGE_PURGEINTERVAL: 1
```

Number of Solr pods to run in this StatefulSet

Number of Zookeepers

To be clear, you can tune GC settings and number of replicas after the cluster is built. But changing the size of the persistent volumes is more complicated so you should try to pick a good size initially.

Configure Storage Class for Solr Pods (Optional)

If you wish to run with a storage class other than the default you can create a storage class for your Solr pods before you install. For example to create regional disks in GCP you can create a file called storageClass.yaml with the following contents:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
 name: solr-gke-storage-regional
provisioner: kubernetes.io/gce-pd
parameters:
 type: pd-standard
 replication-type: regional-pd
 zones: us-west1-b, us-west1-c
```

and then provision into your cluster by calling:

```
kubectl apply -f storageClass.yaml
```

to then have Solr use the storage class by adding the following to the custom values yaml:

```
solr:
  volumeClaimTemplates:
    storageClassName: solr-gke-storage-regional
    storageSize: 250Gi
```

NOTE: We're not advocating that you must use regional disks for Solr storage, as that would be redundant with Solr replication. We're just using this as an example of how to configure a custom storage class for Solr disks if you see the need. For instance, you could use regional disks without Solr replication for write-heavy type collections.

Multiple Node Pools

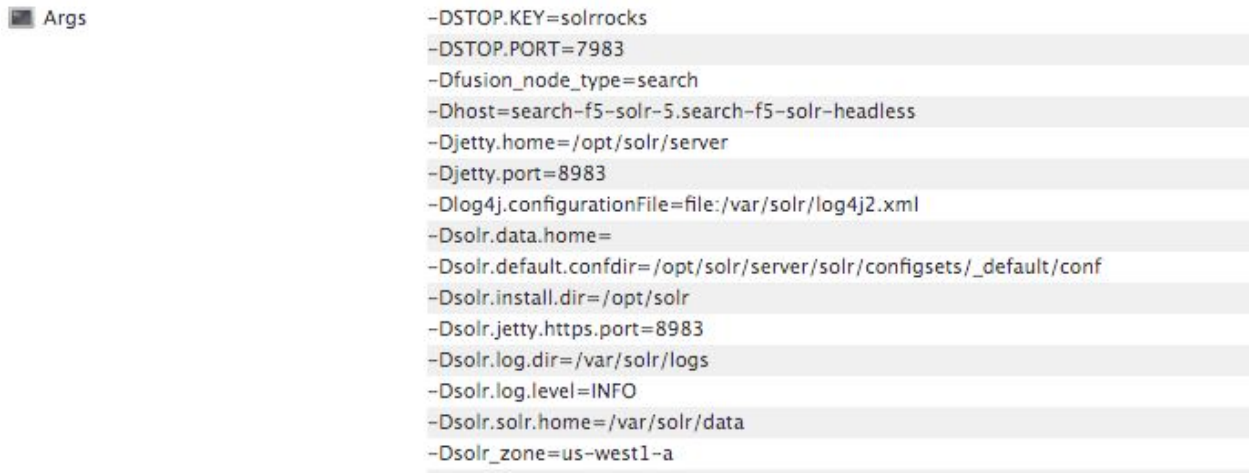
As discussed in the Workload Isolation with Multiple Node Pools section above, Lucidworks recommends isolating search workloads from analytics workloads using multiple node pools. You'll need to define multiple node pools for your cluster as our scripts do not do this for you; we do provide an example script for GKE, see: `create_gke_cluster_node_pools.sh`.

In the custom values yaml file, you can add additional Solr StatefulSets by adding their names to the list under the `nodePools` property. If any property for that statefulset needs to be changed from the default set of values, then it can be set directly on the object representing the node pool, any properties that are omitted are defaulted to the base value. See the following example:

```
solr:
  ..
  replicaCount: 3
  nodePools:
    - name: ""
    - name: "analytics"
      javaMem: "-Xmx4g"
      replicaCount: 3
      volumeClaimTemplates:
        storageSize: "100Gi"
      nodeSelector:
        fusion_node_type: analytics
      resources:
        requests:
          cpu: 2
          memory: 12Gi
        limits:
          cpu: 3
          memory: 12Gi
    - name: "search"
      javaMem: "-Xms11g -Xmx11g -Xmn5g -XX:ParallelGCThreads=8"
      replicaCount: 12
      volumeClaimTemplates:
        storageSize: "50Gi"
      nodeSelector:
        fusion_node_type: search
      resources:
        limits:
          cpu: "7700m"
          memory: "26Gi"
        requests:
          cpu: "7000m"
          memory: "25Gi"
```

In the above example the analytics partition will have 3 replicas (Solr pods), but the search nodepool would have 12 replicas. Each nodepool would automatically be assigned the property of `-Dfusion_node_type=<search/system/analytics>` which matches the `name` of the nodePool.

The Solr pods will have a **fusion_node_type** System property set on them as shown below:



You can use the **fusion_node_type** property in Solr auto-scaling policies to govern replica placement during collection creation.

Solr Auto-scaling Policy

You can configure a custom Solr auto-scaling policy in the custom values yaml file under the fusion-admin section as shown below:

```
fusion-admin:
  ...
  solrAutocalingPolicyJson:
    {
      "set-cluster-policy": [
        {"node": "#ANY", "shard": "#EACH", "replica":"<2"},
        {"replica": "#EQUAL", "sysprop.solr_zone": "#EACH", "strict" : false}
      ]
    }
```

You can use an auto-scaling policy to govern how the shards and replicas for Fusion system and application-specific collections are laid out.

If your cluster defines the search, analytics, and system node pools, then we recommend using the **policy.json** provided in the fusion-cloud-native repo as a starting point. The Fusion Admin

service will apply the policy from the custom values yaml file to Solr before creating system collections during initialization.

Pod Network Policy

A Kubernetes network policy governs how groups of pods are allowed to communicate with each other and other network endpoints. With Fusion, it's expected that all incoming traffic flows through the API Gateway service. Moreover, all Fusion services in the same namespace expect an internal JWT to be included in the request, which is supplied by the Gateway. Consequently, Fusion services enforce a basic level of API security and do not need an additional network policy to protect them from other pods in the cluster. However, some organizations will still want to configure a network policy. Lucidworks will provide a starting policy yaml file with Fusion 5.1.

Install Fusion 5 on Kubernetes

At this point, you're ready to install Fusion 5 using your custom values yaml file. Use the appropriate setup script for your platform, such as `setup_f5_aks.sh` for AKS. If you're not running on AKS, EKS, or GKE, then simply use the setup_f5_k8s.sh script. Pass the **--help** option to see script usage details. For instance, the following command will install Fusion along with Prometheus and Grafana into the **default** namespace in a cluster named **search** running in the **us-west1** region with the **f5** release label:

```
./setup_f5_gke.sh -c search -p gcp-project -z us-west1 \
  -r f5 -n default --prometheus install \
  -t -h <ingress-hostname>
```

Once the install completes, refer to the Verifying the Fusion Installation steps to verify your Fusion installation is running correctly.

*IMPORTANT: If you used our script to configure an Ingress for the API gateway service, (-t -h options), then you should move the contents of the **tls-values.yaml** file under the api-gateway section of your main custom values yaml file. This alleviates having to keep track of both configuration files when upgrading. For instance, if you passed -t -h test1.lucidworkssales.com to the setup script, then you would copy the contents of tls-values.yaml to your main custom values yaml file under the api-gateway section as shown below:*

```
api-gateway:
  service:
    type: "NodePort"
  ingress:
    enabled: true
    host: "test1.lucidworkssales.com"
    tls:
      enabled: true
    annotations:
```

```
    "networking.gke.io/managed-certificates": "f502rc6-managed-certificate"
    "kubernetes.io/ingress.class": "gce"
nodeSelector:
  cloud.google.com/gke-nodepool: default-pool
pod:
  annotations:
    prometheus.io/port: "6764"
    prometheus.io/scrape: "true"
```

Day Two Operations

This section describes an overview of common daily maintenance tasks.

Helm Upgrade Script

Our `setup_f5_*.sh` scripts are nice for getting started and for checking your environment has all the necessary command-line tools before attempting an installation. However, once you have a working cluster, we recommend creating a simple upgrade script that hardcodes the various parameters and alleviates you needing to remember which parameters to pass to the script. This is especially helpful when working with multiple K8s clusters. Rename the example script provided in the fusion-cloud-native repo using our custom value yaml naming convention:

```
cp upgrade_fusion.sh.example <provider>_<cluster>_<release>_upgrade_fusion.sh
```

Edit the script to hardcode the various parameters for your cluster, e.g.

```
PROVIDER=gke
CLUSTER_NAME=search
RELEASE=f5
NAMESPACE=default
CHART_VERSION=5.0.2
```

The script assumes your **kubeconfig** is pointing to the correct cluster and fails fast if not. Thus, you'll have to be sure to select the correct kubeconfig before running the script. The script also assumes the use of Helm v3. Be sure to append all custom values yaml files to the **MY_VALUES** variable in the upgrade script, e.g.

```
MY_VALUES="${MY_VALUES} --values gke_search_f5_fusion_affinity.yaml"
```

Of course being a good ops person, you'll also check the script into version control alongside your custom values yaml files.

Logging

Fusion services come pre-configured to write log messages to stdout (common pattern in K8s) and send log messages to Logstash (deployed in our Helm chart). Logstash is configured to index log messages into the Fusion **system_logs** collection. The Log Viewer integrated into the Fusion Admin UI provides basic log analytics for the logs collected in system_logs. However, we fully expect our customers / prospects to already have a preferred log analytics stack, such as ELK, Splunk, DataDog, or similar. The Log Viewer is only there for demo purposes and the unlikely case of a customer not having a preferred logging stack already.

For integration with a customers existing log infrastructure, they have two basic choices:

1. Disable Logstash in our Helm chart (set **logstashEnabled: false** for all services in the custom values yaml) and then have their logging infrastructure scrape logs from the stdout from each pod. This is a very common pattern in Kubernetes and most modern log analytics solutions have good integration with Kubernetes. In most cases, the customers ops team will help guide you on how they want this to work (typically with a log shipper process deployed as a DaemonSet on each node), there's not much you'll have to do.

2. Use Logstash and configure it to send logs to some output sink supported by Logstash. This would be a good option for customers / prospects that use ELK but don't have an existing integration with Kubernetes. Again, most would use something like Filebeat as a DaemonSet to do this vs. relying on our Logstash. This will require some configuration changes to the Logstash in our Helm chart using custom values yaml as is done for most Fusion configuration changes.

Elasticsearch

Elasticsearch is a common tool used to store and query logs and can be easily configured as an output for Logstash. Just include the **elasticsearchHost** to our Helm chart and set it to **host:port**. In most cases, the port number will be 9200.

```
logstash:
   elasticsearchHost: <elasticsearch-host>:9200
```

Grafana Dashboards

If you're running Prometheus and Grafana for monitoring the performance and health of Fusion services, then you'll want to install Lucidworks Grafana dashboards. We provide the following dashboards in the fusion-cloud-native repo:

Dashboard JSON	Description

dashboard_query_pipeline.json	Query Pipeline Metrics including HTTP request metrics and both per-pipeline and per pipeline stage execution times
dashboard_indexing_pipipeline.json	Indexing Service Metrics including HTTP request metrics and per-pipeline metrics execution times
dashboard_jvm_metrics.json	General JVM metrics for instrumented spring-boot services
dashboard_solr_core.json	Solr exporter per-core metric dashboard, including request details

If you used the `--prometheus install` option when installing Fusion, then you need to get the initial Grafana password from a K8s secret by doing:

```
kubectl get secret --namespace "${NAMESPACE}" f5-graf-grafana \
  -o jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

With Grafana, you can either setup a temporary port-forward to a Grafana pod or expose Grafana on an external IP using a K8s LoadBalancer. To define a LoadBalancer, do (replace ${RELEASE} with your Helm release label):

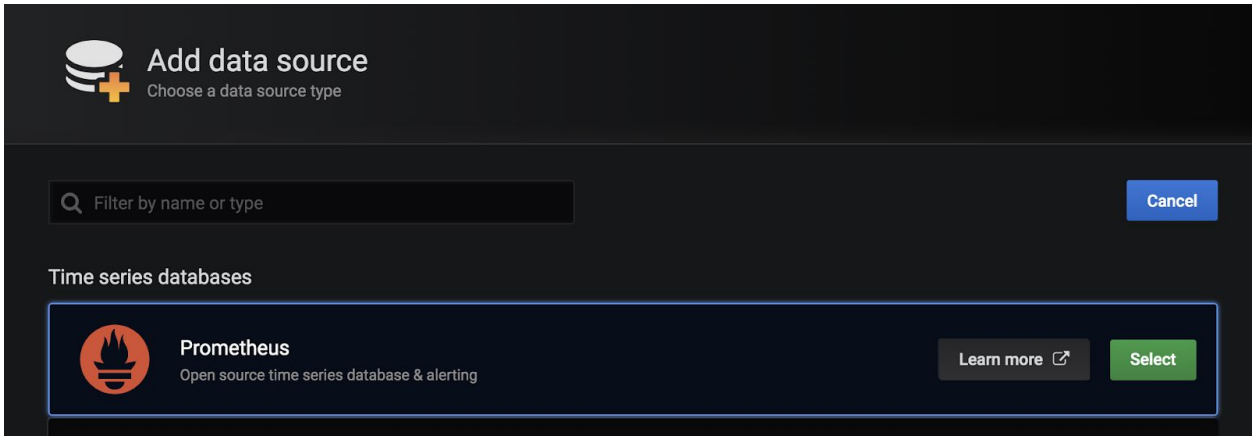
```
kubectl expose deployment ${RELEASE}-graf-grafana --type=LoadBalancer --name=grafana
```

You can use `kubectl get services --namespace <namespace>` to determine when the load balancer is setup and its IP address. Direct your browser to **http://<GrafanaIP>:3000** and enter the username **admin@localhost** and the password that was returned in the previous step.

This will log you into the application. It is recommended that you create another administrative user with a more desirable password.

One of the first things you will want to do is to configure the Prometheus data source in Grafana. Go to the gear icon on the left and then to Data Sources.

Click **Add Data Source** and then click on **Prometheus** as the data source type. It will bring you to a page where it will ask for HTTP URL for the Prometheus server.

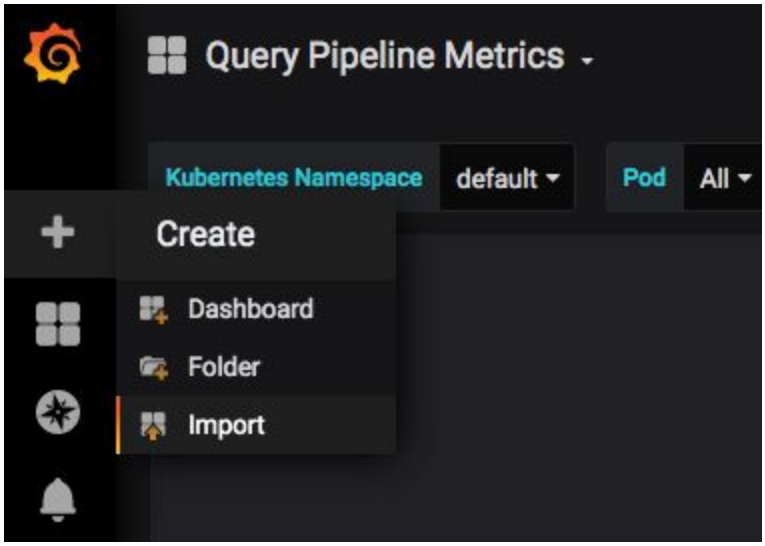


Enter

```
http://<RELEASE>-prom-prometheus-server
```

Configure any additional fields as desired (but defaults are fine), then click Save and Test, which should succeed.

Import the dashboards from fusion-cloud-native:



Pod Affinity Rules

The **example-values/affinity.yaml** file sets affinity rules for all of the Fusion components where the cluster is spread over multiple availability zones. All components have the same affinity setup which follows the following logic:

- When scheduling, prefer to put a pod on a node that is in an availability zone that doesn't already have a running instance of this component

- Require that pods are all deployed on a host that doesn't have a running instance of the component that is being scheduled.

This means that the loss of a host will bring down at most one component but there is the trade off that the cluster will be at least as large as the number of replicas in the largest deployment. If a large number of a certain type of component should be run then it may make sense to relax the **required** to a **preferred** policy on hostname, i.e. change:

```
requiredDuringSchedulingIgnoredDuringExecution:
```

to

```
preferredDuringSchedulingIgnoredDuringExecution:
```

for the **kubernetes.io/hostname** policies. We recommend copying the example file and renaming it using our convention: <provider>_<cluster>_<release>_fusion_affinity.yaml

Append the following to your upgrade script (discussed previously ~ Helm Upgrade Script):

```
MY_VALUES="${MY_VALUES} --values gke_search_f5_fusion_affinity.yaml"
```

Multiple Replicas and Horizontal Pod Auto-scaling

You can configure multiple replicas and horizontal pod autoscalers (tied to CPU usage) using the **example-values/replicas.yaml** file. Once you're ready to apply the replica and HPA settings from the replicas.yaml file, we recommend copying the example file and renaming it using our convention: **`<provider>_<cluster>_<release>_fusion_replicas.yaml`**

Append the following to your upgrade script (discussed previously ~ Helm Upgrade Script):

```
MY_VALUES="${MY_VALUES} --values gke_search_f5_fusion_replicas.yaml"
```

Use Gatling to Run Query Performance / Load Tests

Lucidworks recommends running query performance tests to establish a baseline number of pods for the proxy, query pipeline, and Solr services. You can use the gatling-qps project provided in the fusion-cloud-native repo as a starting point for building a query load test. Gatling.io is a load test framework that provides a powerful Scala-based DSL for constructing performance test scenarios. See **FusionQueryTraffic.scala** in the repo as a starting point for building query performance tests for Fusion 5.

Register Warming Queries

To avoid any potential delays when a new query pod joins the cluster, such as in reaction to an HPA auto-scaling trigger, we recommend registering a small set of queries to "warm up" the query pipeline service before it gets added to the Kubernetes service. In the query-pipeline section of the custom values yaml, configure your warming queries using the structure shown in the example below:

```
warmingQueryJson:
  {
  "pipelines": [
    {
      "pipeline": "<PIPELINE>",
      "collection": "<COLLECTION>",
      "params": {
        "q": ["*:*"]
      }
    },{
      "method" : "POST",
      "pipeline": "<ANOTHER_PIPELINE>",
      "collection": "<ANOTHER_COLL>",
      "params": {
        "q": ["*:*"]
      }
    }
  ],
  "profiles": [
    {
      "profile": "<PROFILE>",
      "params": {
        "q": ["*:*"]
      }
    }
  ]
  }
```

NOTE: The indentation for the opening / closing braces is important for embedding JSON in YAML

Resource Limits

Lucidworks recommends installing Fusion without resource limits initially as they can over-complicate the initial setup of your cluster. Resource requests / limits directly impact the number of nodes needed to deploy Fusion. Once your installation is up-and-running with a critical mass of data, then you can start to fine-tune resource limits for Fusion services. Lucidworks provides a template to help you get started with setting the appropriate resource limits, see: **example-values/resources.yaml**. However, you may need to refine the settings for your environment and workloads, especially for Solr.

Spark Ops

In Fusion 5.x, Spark operates in native Kubernetes mode instead of standalone mode (like in Fusion 4.x). The sections below describe Spark operations in Fusion 5.0.

Cluster Mode

Fusion 5.0 ships with Spark 2.4.3 and operates in "cluster" mode on top of Kubernetes. In cluster mode, each Spark driver runs in a separate pod and hence resources can be managed per job. Each executor also runs in its own pod.

Spark config defaults

The table below shows the default configurations for Spark. These settings are configured in the job-launcher config map, accessible using `kubectl get configmaps <release-name>-job-launcher`. Some of these settings are also configurable via Helm.

Spark Resource Configurations

Spark Configuration	Default value	Helm Variable
spark.driver.memory	3g	
spark.executor.instances	2	executorInstances
spark.executor.memory	3g	
spark.executor.cores	6	
spark.kubernetes.executor.request.cores	3	

Spark Kubernetes Configurations

Spark Configuration	Default value	Helm Variable
spark.kubernetes.container.image.pullPolicy	Always	image.imagePullPolicy
spark.kubernetes.container.image.pullSecrets	[artifactory]	image.imagePullSecrets
spark.kubernetes.authenticate.driver.serviceAccountName	<name>-job-launcher-spark	
spark.kubernetes.driver.container.image	fusion-dev-docker.ci-artifactory.lucidworks.com	image.repository

spark.kubernetes.executor.container.image	fusion-dev-docker.ci-artifactory.lucidworks.com	image.repository

Spark Job Resource Allocation

Number of Instances and Cores Allocated

In order to set the number of cores allocated for a job, add the following parameter keys and values in the Spark Settings field within the "advanced" job properties in the Fusion UI or the sparkConfig object if defining a job via the Fusion API. If `spark.kubernetes.executor.request.cores` is not set (default config), then Spark will set the number of CPUs for the executor pod to be the same number as spark.executor.cores. In that case, if spark.executor.cores is 3, then Spark will allocate 3 CPUs for the executor pod and will run 3 tasks in parallel. To under-allocate the CPU for the executor pod and still run multiple tasks in parallel, set spark.kubernetes.executor.request.cores to a lower value than spark.executor.cores. The ratio for `spark.kubernetes.executor.request.cores` to spark.executor.cores depends on the type of job: either CPU-bound or I/O-bound. Allocate more memory to the executor if more tasks are running in parallel on a single executor pod.

Parameter Key	Parameter Example Value
spark.executor.instances	3
spark.kubernetes.executor.request.cores	3
spark.executor.cores	6
spark.driver.cores	1

If these settings are left unspecified, then the job launches with a driver using one core and 3GB of memory plus two executors, each using one core with 1GB of memory.

Memory Allocation

The amount of memory allocated to the driver and executors is controlled on a per-job basis using the spark.executor.memory and spark.driver.memory parameters in the Spark Settings section of the job definition in the Fusion UI or within the sparkConfig object in the JSON definition of the job.

Parameter Key	Parameter Example Value
spark.executor.memory	6g
spark.driver.memory	2g

Configuring credentials in the Kubernetes cluster

AWS/GCS credentials can be configured per job or per cluster.

Configuring GCS credentials for Spark jobs

1. Create a secret containing the credentials JSON file.
- See https://cloud.google.com/iam/docs/creating-managing-service-account-keys on how to create service account JSON files.
- `kubectl create secret generic solr-dev-gc-serviceaccount-key --from-file=/Users/kiranchitturi/creds/solr-dev-gc-serviceaccount-key.json`
2. Create an extra config map in Kubernetes setting the required properties for GCP.
    a. Create a properties file with GCP properties
    - $ cat gcp-launcher.properties

```
spark.kubernetes.driverEnv.GOOGLE_APPLICATION_CREDENTIALS =
/mnt/gcp-secrets/solr-dev-gc-serviceaccount-key.json
spark.kubernetes.driver.secrets.solr-dev-gc-serviceaccount-key =
/mnt/gcp-secrets
spark.kubernetes.executor.secrets.solr-dev-gc-serviceaccount-key =
/mnt/gcp-secrets
spark.executorEnv.GOOGLE_APPLICATION_CREDENTIALS =
/mnt/gcp-secrets/solr-dev-gc-serviceaccount-key.json
spark.hadoop.google.cloud.auth.service.account.json.keyfile =
/mnt/gcp-secrets/solr-dev-gc-serviceaccount-key.json
```

    b. Create a config map based on the properties file

```
kubectl create configmap gcp-launcher
--from-file=gcp-launcher.properties
```

3. Add the gcp-launcher config map to values.yaml under job-launcher.

```
configSources: gcp-launcher
```

Configuring S3 credentials for Spark jobs

AWS credentials can't be set via a single file. So, we have to set two environment variables referring to the key and secret.

1. Create a secret pointing to the creds:

```
kubectl create secret generic aws-secret --from-literal=key='<access
key>' --from-literal=secret='<secret key>'
```

2. Create an extra config map in Kubernetes setting the required properties for AWS:
    a. Create a properties file with AWS properties

```
cat aws-launcher.properties
spark.kubernetes.driver.secretKeyRef.AWS_ACCESS_KEY_ID=aws-secret
:key
spark.kubernetes.driver.secretKeyRef.AWS_SECRET_ACCESS_KEY=aws-se
cret:secret
spark.kubernetes.executor.secretKeyRef.AWS_ACCESS_KEY_ID=aws-secr
```

```
et:key
spark.kubernetes.executor.secretKeyRef.AWS_SECRET_ACCESS_KEY=aws-
secret:secret
```

b. Create a config map based on the properties file:

```
kubectl create configmap aws-launcher
--from-file=aws-launcher.properties
```

3. Add the aws-launcher config map to values.yaml under job-launcher:

```
configSources: aws-launcher
```

Configuring Azure Data Lake credentials for Spark jobs

Configuring Azure through environment variables or configMaps does not seem to be possible at the moment. You need to manually upload the core-site.xml file into the job-launcher pod at /app/spark-dist/conf.

Currently only Data Lake Gen 1 is supported.

Here's what the core-site.xml file should look like:

```
<property>
  <name>dfs.adls.oauth2.access.token.provider.type</name>
  <value>ClientCredential</value>
</property>
<property>
    <name>dfs.adls.oauth2.refresh.url</name>
    <value> Insert Your OAuth 2.0 Endpoint URL Value Here </value>
</property>
<property>
    <name>dfs.adls.oauth2.client.id</name>
    <value> Insert Your Application ID Here </value>
</property>
<property>
    <name>dfs.adls.oauth2.credential</name>
    <value>Insert the Secret Key Value Here </value>
</property>
<property>
    <name>fs.adl.impl</name>
    <value>org.apache.hadoop.fs.adl.AdlFileSystem</value>
</property>
<property>
    <name>fs.AbstractFileSystem.adl.impl</name>
    <value>org.apache.hadoop.fs.adl.Adl</value>
```

```
</property>
```

Configuring credentials per job

1. Create a Kubernetes secret with the GCP/AWS credentials.
2. Add the Spark configuration to configure the secrets for the Spark driver/executor.

GCS

1. Create a secret containing the credentials JSON file. (See https://cloud.google.com/iam/docs/creating-managing-service-account-keys on how to create service account JSON files)
   ```
   kubectl create secret generic solr-dev-gc-serviceaccount-key
   --from-file=/Users/kiranchitturi/creds/solr-dev-gc-serviceaccount-key.j
   son
   ```
2. Toggle the Advanced config in the job UI and add the following to the Spark configuration:
   ```
   spark.kubernetes.driver.secrets.solr-dev-gc-serviceaccount-key =
   /mnt/gcp-secrets
   spark.kubernetes.executor.secrets.solr-dev-gc-serviceaccount-key =
   /mnt/gcp-secrets
   spark.kubernetes.driverEnv.GOOGLE_APPLICATION_CREDENTIALS =
   /mnt/gcp-secrets/solr-dev-gc-serviceaccount-key.json
   spark.executorEnv.GOOGLE_APPLICATION_CREDENTIALS =
   /mnt/gcp-secrets/solr-dev-gc-serviceaccount-key.json
   spark.hadoop.google.cloud.auth.service.account.json.keyfile =
   /mnt/gcp-secrets/solr-dev-gc-serviceaccount-key.json
   ```

S3

AWS credentials can't be set via a single file. So, we have to set two environment variables referring to the key and secret.

1. Create a secret pointing to the creds
   ```
   kubectl create secret generic aws-secret --from-literal=key='<access
   key>' --from-literal=secret='<secret key>'
   ```
2. Toggle the Advanced config in the job UI and add the following to Spark configuration:
   ```
   spark.kubernetes.driver.secretKeyRef.AWS_ACCESS_KEY_ID=aws-secret:key
   spark.kubernetes.driver.secretKeyRef.AWS_SECRET_ACCESS_KEY=aws-secret:s
   ecret
   spark.kubernetes.executor.secretKeyRef.AWS_ACCESS_KEY_ID=aws-secret:key
   spark.kubernetes.executor.secretKeyRef.AWS_SECRET_ACCESS_KEY=aws-secret
   :secret
   ```

How to get logs for a Spark job

- To get the initial logs that contain information about the pod spinup, do
  ```
  curl -X GET -u admin:password123
  http://localhost:8764/api/apollo/spark/driver/log/{jobId}
  ```

- Get the pod ID by running
  `k get pods -l spark-role=driver -l jobConfigId=<job-id>`
- Logs from failed jobs can be obtained by using
  `kubectl logs [DRIVER-POD-NAME]`
- Logs from running containers can be tailed using the -f parameter:
  `kubectl logs -f [POD-NAME]`

Spark deletes failed and successful executor pods. Fusion provides a cleanup Kubenetes cron job that removes successfully completed driver pods every 15 minutes.

Pod cleanup

Driver pods are cleaned up using a Kubernetes cron job that runs every 15 minutes to clean up pods using this command:

```
kubectl delete pods --namespace default
--field-selector=status.phase=Succeeded -l spark-role=driver
```

This cron job is created automatically when the job-launcher microservice is installed in the Fusion cluster.

Spark History Server

While logs from the Spark driver and executor pods can be viewed using `kubectl logs [POD_NAME]`, executor pods are deleted at their end of their execution, and driver pods are deleted by Fusion on a default schedule of every hour. In order to store and view Spark logs in a more long-term fashion, you can install the Spark History Server into your Kubernetes cluster and configure Spark to write logs in a manner that will persist.

Installing Spark History Server

Spark History Server can be installed via its default Helm chart:

```
helm install stable/spark-history-server --values values.yaml
```

Recommended Configuration

Our recommended configuration for using the Spark History Server with Fusion is to store and read Spark logs in cloud storage. For installations on Google Kubernetes Engine, we suggest setting these keys in the values.yaml:

```
gcs:
        enableGCS: false
        secret: history-secrets
        key: [SECRET_KEY_NAME].json
        logDirectory: gs://[BUCKET_NAME]
service:
    type: ClusterIP
    port: 18080
```

We override the default service.type of LoadBalancer with ClusterIP to keep the History Server from being accessible to outside connections without port-forwarding.

You may need to set up your secret for full access to the cloud bucket:

```
$ export ACCOUNT_NAME=[SECRET_KEY_NAME]
$ export GCP_PROJECT_ID=[PROJECT_ID]

$ gcloud iam service-accounts create ${ACCOUNT_NAME} --display-name
"${ACCOUNT_NAME}"
$ gcloud iam service-accounts keys create "${ACCOUNT_NAME}.json"
--iam-account "${ACCOUNT_NAME}@${GCP_PROJECT_ID}.iam.gserviceaccount.com"
$ gcloud projects add-iam-policy-binding ${GCP_PROJECT_ID} --member
"serviceAccount:${ACCOUNT_NAME}@${GCP_PROJECT_ID}.iam.gserviceaccount.com"
--role roles/storage.admin
$ gsutil iam ch
serviceAccount:${ACCOUNT_NAME}@${GCP_PROJECT_ID}.iam.gserviceaccount.com:obje
ctAdmin gs://[BUCKET_NAME]
```

The service key sets up the history server on an internal IP within your cluster but does not create a LoadBalancer (which is the default setting in the Spark History Server Helm chart). This prevents the server from being exposed to outside access by default. We'll look at how to access the history server shortly.

Other Configurations

Azure

```
$ echo "your-storage-account-name" >> azure-storage-account-name
$ echo "your-container-name" >> azure-blob-container-name

# to auth with sas token (if wasbs.sasKeyMode=true, which is the default)
$ echo "your-azure-blob-sas-key" >> azure-blob-sas-key

# or to auth with storage account key
$ echo "your-azure-storage-account-key" >> azure-storage-account-key

$ kubectl create secret generic azure-secrets
--from-file=azure-storage-account-name --from-file=azure-blob-container-name
[--from-file=azure-blob-sas-key | --from-file=azure-storage-account-key]
```

For SAS token access, values.yaml should look like:

```
wasbs:
    enableWASBS: true
    secret: azure-secrets
    sasKeyName: azure-blob-sas-key
    storageAccountNameKeyName: azure-storage-account-name
    containerKeyName: azure-blob-container-name
    logDirectory: [BUCKET-NAME]
```

For non-SAS access:

```
wasbs:
    enableWASBS: true
    secret: azure-secrets
    sasKeyMode: false
    storageAccountKeyName: azure-storage-account-key
    storageAccountNameKeyName: azure-storage-account-name
    containerKeyName:  azure-blob-container-name
    logDirectory: [BUCKET-NAME]
```

AWS

The recommended approach for S3 access is to use AWS IAM roles, but you can also use a access/secret key pair as a Kubernetes secret:

```
$ aws iam list-access-keys --user-name your-user-name --output text | awk
'{print $2}' >> aws-access-key

$ echo "your-aws-secret-key" >> aws-secret-key

$ kubectl create secret generic aws-secrets --from-file=aws-access-key
--from-file=aws-secret-key
```

For IAM, your values.yaml will be:

```
s3:
    enableS3: true
    logDirectory: s3a://[BUCKET-NAME]
```

(note the Hadoop s3a:// link instead of s3://)

With a access/secret pair, you'll need to add the secret:

```
s3:
    enableS3: true
    enableIAM: false
    accessKeyName: aws-access-key
    secretKeyName: aws-secret-key
    logDirectory: s3a://[BUCKET-NAME]
```

Configuring Spark

After the History Server has been set up, then the Fusion job-launcher deployment ConfigMap's application.yaml key will need to be updated with these Spark settings so the driver and executors know where to write out their logs. In this example, we're redirecting to a GCS bucket:

```
spark:
    eventLog:
        enabled: true
```

```
        compress: true
        dir: gs://[BUCKET-NAME]
    hadoop:
        google:
            cloud:
                auth:
                    service:
                        account:
                            json:
                                keyfile: /etc/secrets/[SECRET_KEY_NAME].json]
    kubernetes:
        driver:
            secrets:
                history-secrets: /etc/secrets
        executor:
            secrets:
                history-secrets: /etc/secrets
```

Accessing The Spark History Server

As we have set up the History Server to only set up a ClusterIP, we will need to port forward the server using kubectl:

```
kubectl get pods # to find the Spark History Server pod
kubectl port-forward [POD_NAME] 18080:18080
```

You can now access the Spark History Server at http://localhost:18080.

Upgrades with Helm v3

One of the most powerful features provided by Kubernetes and a cloud-native microservices architecture is the ability to do a rolling update on a live cluster. Fusion 5 allows customers to upgrade from Fusion 5.0.2 to a later 5.x.y version on a live cluster with zero downtime or disruption of service.

When Kubernetes performs a rolling update to an individual microservice, there will be a mix of old and new services in the cluster concurrently (only briefly in most cases) and requests from other services will be routed to both versions. Consequently, Lucidworks ensures all changes we make to our service do not break the API interface exposed to other services in the same 5.x line of releases. We also ensure stored configuration remains compatible in the same 5.x release line.

Lucidworks releases minor updates to individual services frequently, so our customers can pull in those upgrades using Helm at their discretion.

To upgrade your cluster at any time, use the --upgrade option with our setup scripts in this repo.

The scripts in this repo automatically pull in the latest chart updates from our Helm repository and deploy any updates needed by doing a diff of your current installation and the latest release

from Lucidworks. To see what would be upgraded, you can pass the --dry-run option to the script.