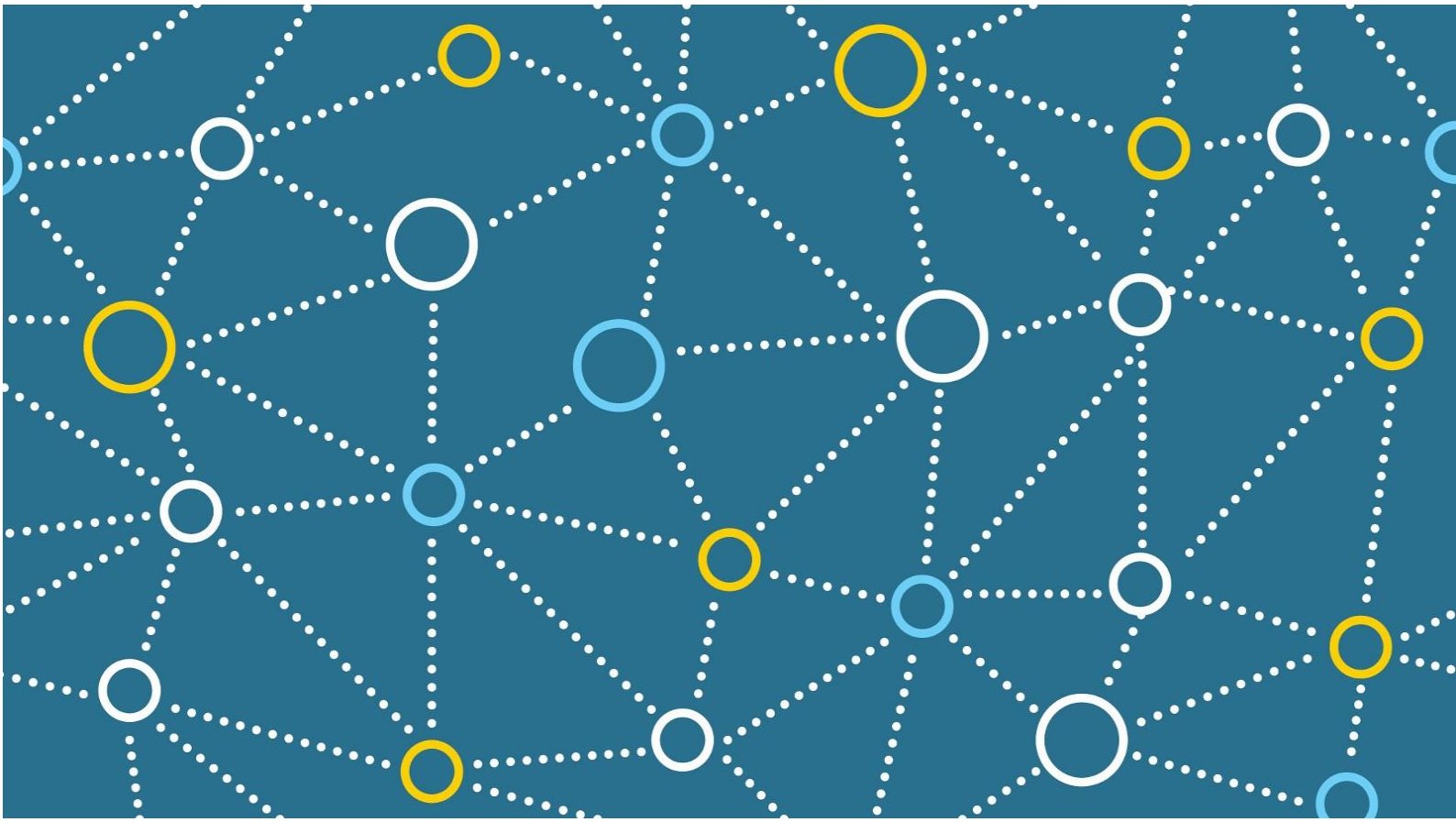


Yellowfin on Docker & Kubernetes

By Jake Rapinett

Version 1.2.2

Table of Contents

1. Introduction

This document serves as a technical guide to provide the best practices for deploying Yellowfin in Docker and Kubernetes environments.

This white paper will detail different Yellowfin setups that can be deployed, their use cases and provide deployment files that can be used as a baseline for deploying Yellowfin into your Docker and/or Kubernetes environments.

2. Yellowfin Docker resources

This section covers the resources that Yellowfin will be providing alongside this white paper, that will assist with deploying Yellowfin in Docker and Kubernetes environments.

2.1 Yellowfin All-In-One Image

The (Yellowfin All-In-One image) contains both the Yellowfin application as well as the Yellowfin repository database, which in this case is an embedded PostgreSQL database.

The recommended use cases for this image are:

- Short-term POCs and Demos.
- Testing Yellowfin functionality in a sandboxed environment.

This image will not persist data outside of the Docker container, and all content will be lost when the container is destroyed.

It is not recommended to use this image for any production deployments.

This Docker image also has the limitation that it cannot be used in a clustered environment.

2.1.1 Yellowfin All-In-One Image configuration options

The All-In-One image has the following environment variable available for configuration:

Configuration Item	Description	Example
Application Memory	Specify the number of megabytes of memory to be assigned to the Yellowfin application. If unset, Yellowfin will use the Java default (usually 25% of System RAM)	-e APP_MEMORY=4096

2.2 Yellowfin App-Only Image

The (Yellowfin App Only image) contains only the Yellowfin application and must be connected to an existing repository database.

Yellowfin instances deployed using this image can be deployed as discrete instances or as part of a Yellowfin Cluster. This image is suitable for both production and non-production environments, as data is persisted on the external Yellowfin repository, so Yellowfin data will be retained, if containers are destroyed.

The recommended use cases for this image are:

- Long-term instances of Yellowfin, where data persistence is important.

- Clustered Yellowfin deployments.

2.2.1 Yellowfin App-Only Image configuration options

The App-Only image has the following environment variables available for configuration:

Configuration Item	Description	Example
JDBC Driver Name, JDBC_CLASS_NAME	Configure the JDBC Driver Class for connecting to the Yellowfin Repository (Required)	-e JDBC_CLASS_NAME=org.postgresql.Driver
Repository URL, JDBC_CONN_URL	Specify the Connection URL to the Repository Database (Required)	-e JDBC_CONN_URL=jdbc:postgresql://host:5432/yf
Repository Username, JDBC_CONN_USER	Specify the Database User required to access the Repository Database (Required)	-e JDBC_CONN_USER=dba
Repository Password, JDBC_CONN_PASS	Specify the Database Password required to access the Repository Database. This can be encrypted. (Required)	-e JDBC_CONN_PASS=secret
Application Memory, APP_MEMORY	Specify the number of megabytes of memory to be assigned to the Yellowfin application. If unset, Yellowfin will use the Java default (usually 25% of System RAM)	-e APP_MEMORY=4096
DB Password Encrypted, JDBC_CONN_ENCRYPTED	Specify whether the Database Password is encrypted (true/false)	-e JDBC_CONN_ENCRYPTED=true
Connection Pool Size, JDBC_MAX_COUNT	Specify the maximum size of the Repository Database connection pool. (Default: 25)	-e JDBC_MAX_COUNT=25
Default Welcome Page, WELCOME_PAGE	Specify the default index page.	-e WELCOME_PAGE=custom_index.jsp
Internal Application HTTP Port	Specify the internal HTTP port. (Default: 8080)	-e APP_SERVER_PORT=9090
Internal Shutdown Port	Specify the internal shutdown port. (Default: 8083)	-e TCP_PORT=9093
Proxy Port, PROXY_PORT	External Proxy Port	-e PROXY_PORT=443
Proxy Scheme, PROXY_SCHEME	External Proxy Scheme (http/https)	-e PROXY_SCHEME=https
Proxy Host, PROXY_HOST	External Proxy Host or IP address	-e PROXY_HOST=reporting.company.com
External Cluster Address, CLUSTER_ADDRESS	External Cluster Address for Cluster Messaging. Usually the host or IP address of the Docker Host, or for Docker Swarm and Kubernetes, the DNS name of the container.	-e CLUSTER_ADDRESS=10.10.10.23
External Cluster Port, CLUSTER_PORT	A Unique TCP port for this container to receive Cluster Messages from other nodes	-e CLUSTER_ADDRESS=7801

Internal Cluster Network Adapter, CLUSTER_INTERFACE	Specify the docker interface to bind Cluster Messages to. Defaults to eth0, but this may need to be changed for Kubernetes and Docker Swarm	-e CLUSTER_INTERFACE=match-interface:eth1
Background Processing Task Types, NODE_BACKGROUND_TASKS	Comma separated list of which background Task Types can be run on this node. NODE_PARALLEL_TASKS must also be updated if this item is specified. If unspecified, all Task Types will be enabled.	-e NODE_BACKGROUND_TASKS= FILTER_CACHE,ETL_PROCESS_TASK
Background Task Processing Jobs, NODE_PARALLEL_TASKS	Comma separated list of the number of concurrent tasks for each Task Type that can be run on this node. The number of elements passed here must match the number of Task Types passed by NODE_BACKGROUND_TASKS	-e NODE_PARALLEL_TASKS=5,4
Additional Libraries URL, LIBRARY_ZIP	URL to a Zip file that contains additional libraries to be extracted into lib folder of Yellowfin. This can be used to add additional JDBC drivers or custom plugins to Yellowfin. Make sure that the path is not included with zip entries in the archive.	-e LIBRARY_ZIP=http://lib-host/libraries.zip

2.3 Yellowfin's GitHub Repository

The Yellowfin GitHub Repository provides the above Yellowfin Docker images as downloadable Dockerfiles, as well as a copy of the deployment file examples in this white paper.

2.3.1 Building the Yellowfin Dockerfiles into Docker Images

To build the Dockerfiles in the Yellowfin GitHub Repository into usable Docker images, run the following commands in a terminal/command line session that is in the same folder as one of the Dockerfiles, substituting in values that match the environment Yellowfin will be deployed in.

1) Build the Docker Image - uses the Dockerfile in the local directory
   `docker build -t yellowfin-app-only .`
2) Tag the image with a version
   `docker tag yellowfin-app-only:latest My_Docker_Registry/yellowfin-app-only:latest`
3) Push the image to a Docker Registry
   `docker push My_Docker_Registry/yellowfin-app-only:latest`

2.4 Yellowfin on Docker Hub

The Yellowfin Docker Hub account hosts both of the Docker Images outlined in this white paper. The Docker Images that are in the Yellowfin App-Only Image Repository and Yellowfin All-In-One Image Repository are versioned to match official Yellowfin release builds.

To pull the latest Yellowfin All-In-One Docker Image, run the following command:

```
docker pull yellowfinbi/yellowfin-all-in-one:latest
```

To pull the latest Yellowfin App-Only Docker Image, run the following command:

```
docker pull yellowfinbi/yellowfin-app-only:latest
```

To use a particular version of Yellowfin, check that it is available in the relevant Image Repository via searching for it in the “Tags” section of the repository, then calling the Docker pull command that matches that version. Here is an example command for pulling a Yellowfin 9.5.1 All-In-One Image:

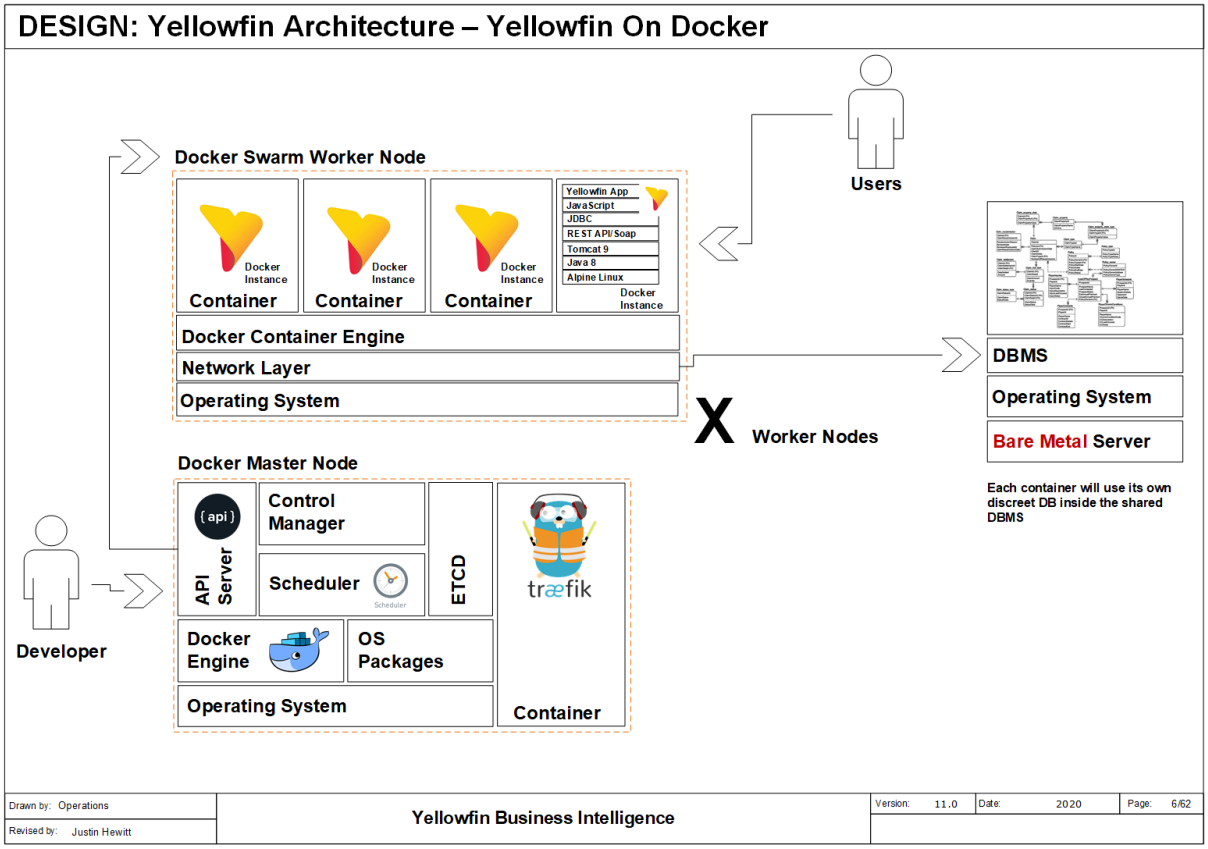
```
docker pull yellowfinbi/yellowfin-all-in-one:9.5.1
```

3. Yellowfin Container Architecture

This section aims to provide an outline on the architecture involved with running Yellowfin as a container in both Docker and Kubernetes environments.

3.1 Yellowfin on Docker Architecture

With Yellowfin on Docker, the below diagram will provide an outline of a Yellowfin cluster running in a Docker Swarm environment, which in our Docker deployment examples, has the most components.



Breaking down the above diagram:

- The Yellowfin component: Yellowfin containers that make up the Yellowfin cluster have been deployed over multiple Docker worker/slave nodes in a Docker Swarm cluster. Unlike Traefik, there is no restriction to which nodes the Yellowfin containers can be deployed to. Depending on the Yellowfin deployment type that was chosen, the Docker environment will have one or more Yellowfin instances running, with those instances connecting to either the same database (for a clustered Yellowfin deployment) or different database (discrete instance deployment).

- The Yellowfin Repository Database: Indicated by the DBMS section in the diagram, Yellowfin recommends not running the repository database in a Docker container for production workloads, and instead either utilizing a bare metal server, Virtual Machine or the cloud-provider equivalents, e.g. for AWS, an EC2 instance or using AWS RDS.
- Traefik: Outlined as part of the Yellowfin Cluster on Docker Swarm example, Traefik is a container-aware reverse proxy that runs on the manager node/s in a Docker Swarm environment - due to it needing access to the Docker Swarm API - and handles the load balancing and sticky sessions for the Yellowfin containers.

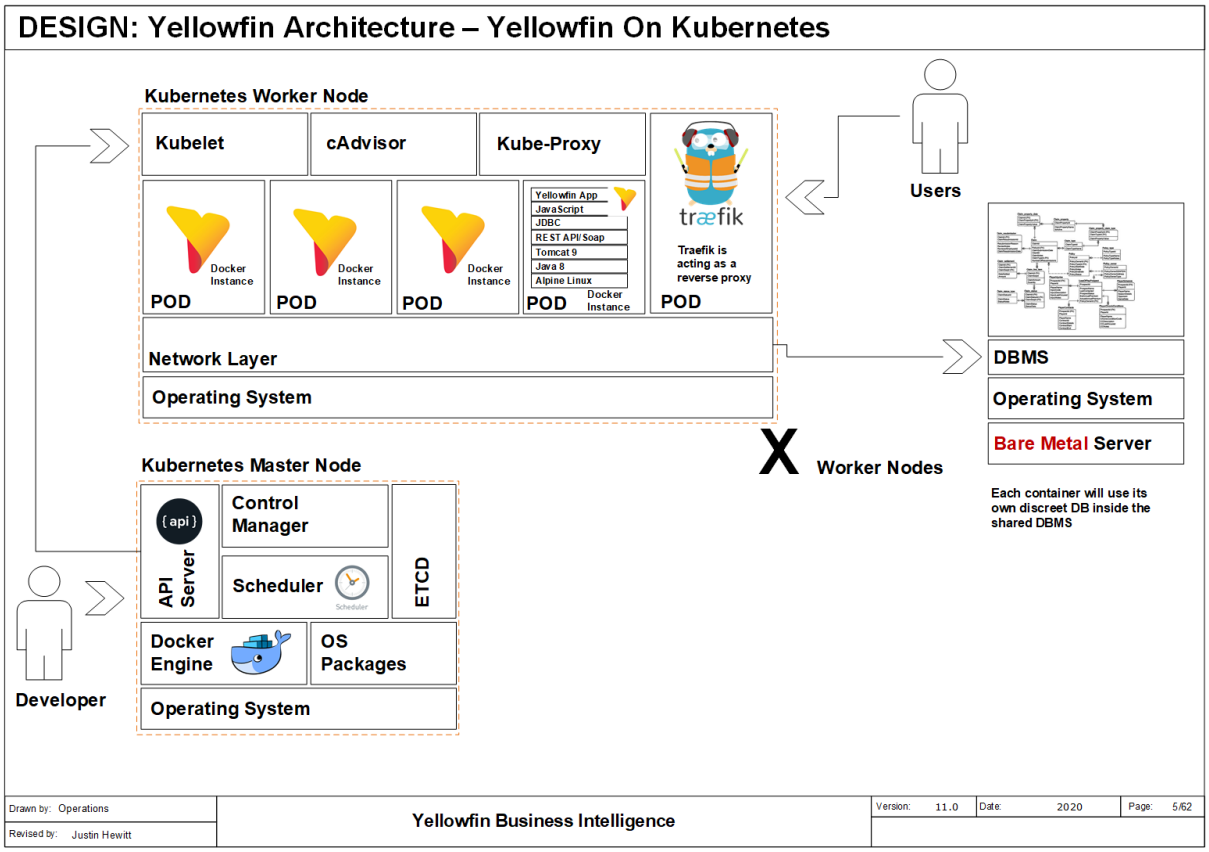
For the single instance deployments of Yellowfin on Docker Swarm, Traefik is an optional component, as with only one Yellowfin instance running, a reverse proxy is not required. With the multiple discrete Yellowfin instances deployment on Docker Swarm, deploying Traefik is optional – if it is deployed in this situation, it can handle the routing of requests to the discrete Yellowfin deployments, instead of exposing ports on the Docker Swarm cluster that route directly to the Yellowfin instances.

For a single Docker server, the above architecture is similar, just that the Docker Master and Worker/Slave nodes are one server. When running in this environment, whilst we have not provided an example of it, Traefik can be deployed to route requests to Yellowfin containers – See Traefik's documentation on how to achieve this..
With a clustered Yellowfin deployment on a single Docker server, if the Docker environment can have external load balancer that supports sticky sessions provisioned, then that can take the place of Traefik in this diagram.

For both environment types, the Yellowfin All-In-One image does not require the external Yellowfin Repository, as the image comes embedded with one.

3.2 Yellowfin on Kubernetes Architecture

With Yellowfin on Kubernetes, the below diagram will provide an outline of a Yellowfin cluster running in a Kubernetes environment, which in our Kubernetes deployment example, has the most components.



Breaking down the above diagram, which has very similar components to those in the Docker Swarm architectural diagram:

- The Yellowfin component: Yellowfin pods that form the Yellowfin cluster. These pods may be placed on multiple Kubernetes worker nodes depending on available resources in the Kubernetes cluster. Depending on the Yellowfin deployment type that was chosen, the Kubernetes environment will have one or more Yellowfin instances running, with those instances connecting to either the same database (for a clustered Yellowfin deployment) or different database (discrete instance deployment).
- The Yellowfin Repository Database: Indicated by the DBMS section in the diagram, Yellowfin recommends not running the repository database in a Kubernetes pod for production workloads, and instead either utilizing a bare metal server, Virtual Machine or the cloud-provider equivalents, e.g. for AWS, an EC2 instance or using AWS RDS.

- Traefik: Outlined in our Kubernetes deployment examples, Traefik is a container-aware reverse proxy that runs Kubernetes environment and handles the load balancing and sticky sessions for the Yellowfin instances.

Scaling down the above diagram to the single instance deployment of Yellowfin on Kubernetes, the main difference will be there will only be a single Yellowfin pod present.

With the All-In-One Yellowfin Image, there will be no requirement for an external Yellowfin Repository database, as the image includes an embedded repository database.

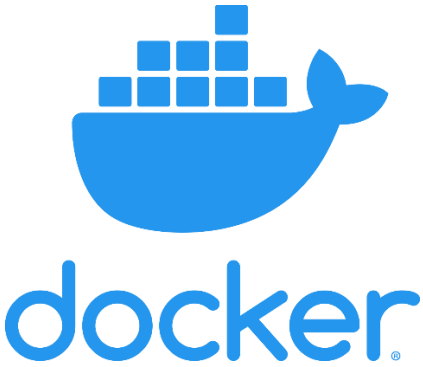
In a multiple discrete instances deployment of Yellowfin, each Yellowfin instance will be communicating with a different Yellowfin Repository database.

4. Yellowfin on Docker – Deployment options

The following section will be covering how to deploy Yellowfin on a single Docker server using Docker Compose, and how to deploy Yellowfin in a Docker Swarm environment using a Docker Stack file.

Pre-requisites: A running Docker server with Docker-Compose installed or a Docker Swarm cluster.

Beyond scope of this document but recommended: Running Yellowfin with HTTPS enabled.



4.1 Deploying Yellowfin on a standalone Docker Server (non-Swarm)

For deploying Yellowfin on a single Docker server, Yellowfin recommends using Docker Compose to run the included deployment files to deploy Yellowfin.

4.1.1 Deploying a sandbox Yellowfin instance – The All-In-One image

The simplest Yellowfin deployment, to deploy a self-contained instance of Yellowfin, save the below YAML markup to a file. To follow our example below, save the file as "*yellowfin-all-in-one.yml*"

```
version: '3'
services:
  yellowfin-all-in-one:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to the host's port 8080
   environment:
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-all-in-one:<RELEASE_VERSION_GOES_HERE>"
```

Then run the following command in a terminal to deploy Yellowfin and execute it in the background.

```
docker-compose up -d -f yellowfin-all-in-one.yml
```

The result of the above deployment file example will be a Yellowfin All-In-One instance running on the Docker host's port 8080, with 4GB of RAM allocated to Yellowfin.

4.1.2 Deploying a single instance of Yellowfin – App-Only image

For this deployment, a Yellowfin Repository database will need to be already created and synced with same version of Yellowfin that will be used in the Yellowfin container.

Save the following YAML markup to a file, which in this example will be saved to the file "yellowfin-single-instance.yml"

```
version: '3'
services:
  yellowfin-standalone-single-instance:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to the host's port 8080
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_HERE # Database connection string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin
```

With the above example, substitute in the database connection settings, which can be found in the existing Yellowfin installation, inside the Yellowfin web.xml file.

As an example, here is an example of those values setup to connect to a local PostgreSQL instance.

```
environment:
  # Required environment variables
  - JDBC_CLASS_NAME=org.postgresql.Driver      # Database driver class name
  - JDBC_CONN_URL=jdbc:postgresql://localhost/docker_standalone_yellowfin_single_instance      # Database connection string
  - JDBC_CONN_USER=postgres      # Username to use when accessing the database
  - JDBC_CONN_PASS=bXFOoj5gnB1oRB1kZq5      # Password for the database user
  - JDBC_CONN_ENCRYPTED=true      # Flag for indicating if the database user's password supplied is encrypted or not.
  - APP_MEMORY=4096            # The amount of memory in megabytes to assign to the Yellowfin Application.
```

Then run the following command in a terminal to deploy Yellowfin and execute it in the background.

```
docker-compose up -d -f yellowfin-single-instance.yml
```

The result of the above deployment file example will be a Yellowfin App-Only instance running on the Docker host's port 8080, with 4GB of RAM allocated to Yellowfin.

4.1.3 Deploying multiple discrete instances of Yellowfin – App-Only image

As with the single-instance approach, for each discrete instance of Yellowfin that is going to be deployed, each instance will need to have initialized a separate Yellowfin Repository database, which needs to be synced with same version of Yellowfin as the one that will be used in the Yellowfin container that is connecting to it.

Save the following YAML markup to a file, which in this example will be saved to the file "yellowfin-multiple-instances.yml"

```
version: '3'
services:
  yellowfin-multi-instance-prod:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to the host's port 8080
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_1_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_1_HERE # Database connection
string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_1_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_1_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=8192 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin

  yellowfin-multi-instance-dev:
   ports:
      - "8090:8080" # Maps Yellowfin running on port 8090 to the host's port 8080
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_2_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_2_HERE # Database connection
string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_2_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_2_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin
```

With the above example, as with the single Yellowfin instance example, substitute in the database connection settings, which can be found in the existing Yellowfin installations, inside the Yellowfin web.xml file.

Then run the following command in a terminal to deploy Yellowfin and execute it in the background.

```
docker-compose up -d -f yellowfin-multiple-instances.yml
```

The result of the above deployment file example will be 2 Yellowfin App-Only instances running on the Docker host, one on port 8080, which will be allocated 8GB of RAM, and the other on port 8090, which will be allocated 4GB of RAM.

4.1.4 Deploying a Yellowfin Cluster – App-Only image

For this deployment, a Yellowfin Repository database will need to be already created and synced with same version of Yellowfin that will be used in the Yellowfin containers.

Save the following YAML markup to a file, which in this example will be saved to the file "yellowfin-cluster.yml"

```
version: '3'
services:
  yellowfin-cluster-node-1:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to the host's port 8080
      #- "7801:7800" # Maps the Yellowfin cluster port to an external port on the host
(Optional)
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_HERE # Database connection string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
      - CLUSTER_ADDRESS=yellowfin-cluster-node-1 # Address to use for clustering -
recommended to use Docker networking to connect the containers
      - CLUSTER_PORT=7800 # TCP Port to use for cluster networking
      -
NODE_BACKGROUND_TASKS=REPORT_BROADCAST_BROADCASTTASK,REPORT_BROADCAST_MIREPORTTASK,FILTER_C
ACHE,SOURCE_FILTER_REFRESH,SOURCE_FILTER_UPDATE_REMINDER,THIRD_PARTY_AUTORUN,ORGREF_CODE_RE
FRESH,ETL_PROCESS_TASK,SIGNALS_DCR_TASK,SIGNALS_ANALYSIS_TASK,SIGNALS_CLEANUP_TASK,COMPOSIT
E_VIEW_REFRESH,SIGNALS_CORRELATION_TASK # Comma separated list of which background Task
Types can be run on this node.
      - NODE_PARALLEL_TASKS=4,4,4,4,4,4,4,4,4,4,4,4,4 # Comma separated list of the number
of concurrent tasks for each Task Type that can be run on
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin

  yellowfin-cluster-node-2:
   ports:
      - "8081:8080" # Maps Yellowfin running on port 8081 to the host's port 8080
      #- "7802:7800" # Maps the Yellowfin cluster port to an external port on the host
(Optional)
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_HERE # Database connection string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
      - CLUSTER_ADDRESS=yellowfin-cluster-node-2 # Address to use for clustering -
recommended to use Docker networking to connect the containers
      - CLUSTER_PORT=7800 # TCP Port to use for cluster networking
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin
      -
NODE_BACKGROUND_TASKS=REPORT_BROADCAST_BROADCASTTASK,REPORT_BROADCAST_MIREPORTTASK,FILTER_C
```

```
ACHE,SOURCE_FILTER_REFRESH,SOURCE_FILTER_UPDATE_REMINDER,THIRD_PARTY_AUTORUN,ORGREF_CODE_RE
FRESH,ETL_PROCESS_TASK,SIGNALS_DCR_TASK,SIGNALS_ANALYSIS_TASK,SIGNALS_CLEANUP_TASK,COMPOSIT
E_VIEW_REFRESH,SIGNALS_CORRELATION_TASK # Comma separated list of which background Task
Types can be run on this node.
      - NODE_PARALLEL_TASKS=4,4,4,4,4,4,4,4,4,4,4,4,4 # Comma separated list of the number
of concurrent tasks for each Task Type that can be run on

  yellowfin-cluster-node-3:
   ports:
      - "8082:8080" # Maps Yellowfin running on port 8082 to the host's port 8080
      #- "7803:7800" # Maps the Yellowfin cluster port to an external port on the host
(Optional)
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_HERE # Database connection string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
      - CLUSTER_ADDRESS=yellowfin-cluster-node-3 # Address to use for clustering -
recommended to use Docker networking to connect the containers
      - CLUSTER_PORT=7800 # TCP Port to use for cluster networking
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin
```

With the above example, as with the other deployment examples, substitute in the database connection settings, which can be found in the existing Yellowfin installations, inside the Yellowfin web.xml file.

Then run the following command in a terminal to deploy Yellowfin and execute it in the background.

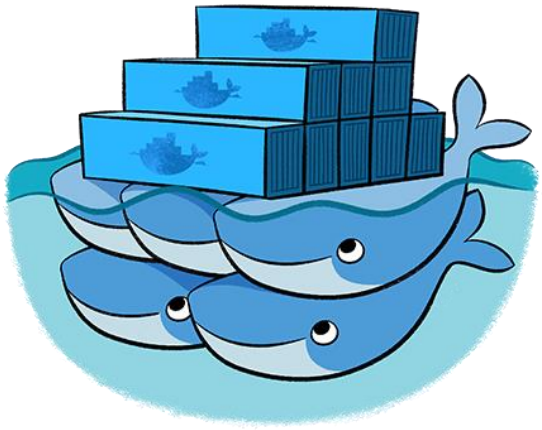
```
docker-compose up -d -f yellowfin-cluster.yml
```

The result of the above deployment file example will be a 3 node Yellowfin Cluster, each allocated 4GB of RAM and running on ports 8080, 8081 and 8082 on the Docker host.

Please note that as each Yellowfin instance is running on a separate port, Yellowfin recommends putting (a load balancer or reverse proxy that has sticky session support in front of the instances, so that users are directed to one of Yellowfin instances for the duration of their session.

4.2 Deploying Yellowfin in a Docker Swarm environment

For deploying Yellowfin in a Docker Swarm environment, Yellowfin recommend using Docker Stacks to run the included deployment files to deploy Yellowfin.



4.2.1 Deploying a sandbox Yellowfin instance – The All-In-One image

The simplest Yellowfin deployment, to deploy a self-contained instance of Yellowfin, save the below YAML markup to a file. To follow along with our example, save the file as "*yellowfin-all-in-one.yml*"

```
version: '3'
services:
  yellowfin-all-in-one:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to Docker Swarm port 8080
   environment:
      - APP_MEMORY=8192 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-all-in-one:<RELEASE_VERSION_GOES_HERE>"
```

Then run the following command in a terminal to deploy Yellowfin.

```
docker stack deploy --compose-file yellowfin-all-in-one.yml yellowfin
```

The result of the above deployment file example will be a Yellowfin All-In-One instance running on port 8080 on the Docker Swarm cluster, with 8GB of RAM allocated to Yellowfin.
As long as the Docker Swarm routing mesh is working between nodes, you will be able to access that Yellowfin instance from any of the Docker nodes.

4.2.2 Deploying a single instance of Yellowfin – App-Only image

For this deployment, a Yellowfin Repository database will need to be already created and synced with same version of Yellowfin that will be used in the Yellowfin container.

Save the following YAML markup to a file, which in this example will be saved to the file "yellowfin-single-instance.yml"

```
version: '3'
services:
  yellowfin-instance:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to Docker Swarm port 8080
   deploy:
      replicas: 1
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_HERE # Database connection string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin
```

With the above example, substitute in the database connection settings, which can be found in the existing Yellowfin installation, inside the Yellowfin web.xml file.

Now run the following command in a terminal to deploy Yellowfin.

```
docker stack deploy --compose-file yellowfin-single-instance.yml yellowfin
```

The result of the above deployment file example will be a Yellowfin App-Only instance running on port 8080 on the Docker Swarm cluster, with 4GB of RAM allocated to Yellowfin.

4.2.3 Deploying multiple discrete instances of Yellowfin – App-Only image

As with the single-instance approach, for each discrete instance of Yellowfin that is going to be deployed, a separate Yellowfin repository database will need to be initialized, which needs to be synced with same version of Yellowfin that will be used in the Yellowfin container that is connecting to it.

Save the following YAML markup to a file, which in this example will be saved to the file “yellowfin-multiple-instances.yml”

```
version: '3'
services:
  yellowfin-multi-instance-prod:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to Docker Swarm port 8080
   deploy:
      replicas: 1
      placement:
        constraints: [node.role == manager] # (Optional) Places the production instance on
a manager node
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_1_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_1_HERE # Database connection
string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_1_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_1_HERE # Password for the database user
```

```
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=8192 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin

  yellowfin-multi-instance-dev:
   ports:
      - "8090:8080" # Maps Yellowfin running on port 8080 to Docker Swarm port 8090
   deploy:
      replicas: 1
      placement:
        constraints: [node.role == worker] # (Optional) Places the dev instance on a worker
node
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_2_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_2_HERE # Database connection
string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_2_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_2_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
   image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin
```

With the above example, as with the single Yellowfin instance example, substitute in the database connection settings, which can be found in the existing Yellowfin installations, inside the Yellowfin web.xml file.

In the above example there is the optional constraints specification under the deployment sections.

This constraint can be removed or modified, as it is there for reference to show how to place the “Production” instance of Yellowfin on a Docker Swarm Manager node, and the “Development” instance on a Docker Swarm Worker node. Please evaluate this deployment specification based on the resources available to the Docker Swarm Manager and Worker nodes.

Now run the following command in a terminal to deploy Yellowfin.

```
docker stack deploy --compose-file yellowfin-multiple-instances.yml yellowfin
```

The result of the above deployment file example will be 2 Yellowfin App-Only instances running on the Docker Swarm cluster, one on port 8080, which will be allocated 8GB of RAM, and the other on port 8090, which will be allocated 4GB of RAM.

4.2.4 Deploying a Yellowfin Cluster – App-Only image

As with the other Yellowfin deployments documented above, a Yellowfin Repository database will need to be initialized for the cluster to connect to, which needs to be synced with same version of Yellowfin that will be used in this cluster deployment.

Compared to deploying on non-Swarm Docker though, Yellowfin recommends that a reverse proxy or load balancer that has sticky session capabilities and is container-aware is deployed in front of the Yellowfin containers.
This is due to the Docker Swarm routing mesh only supporting sticky sessions when using the Enterprise edition of Docker, and when compared to the non-Swarm deployment, all of the Yellowfin containers will be exposed on the same port. For those reasons, Yellowfin highly recommend the usage of Traefik, which our deployment file example will be using to handle load balancing between Yellowfin instances and provide sticky session support via a browser cookie.

Save the following YAML markup to a file, which in this example will be saved to the file "yellowfin-cluster.yml"

```
version: '3'
services:
  yellowfin-cluster:
   ports:
      - "8080:8080" # Maps Yellowfin running on port 8080 to Docker Swarm port 8080
      #- "7801:7800" # Maps the Yellowfin cluster port to an external port on the host
(Optional)
   hostname: "yellowfin-node-{{.Task.Slot}}" # Optional, sets the hostname to to the
provided value.
   environment:
      # Required environment variables
      - JDBC_CLASS_NAME=INSERT_DATABASE_TYPE_HERE # Database driver class name
      - JDBC_CONN_URL=jdbc:INSERT_JDBC_CONNECTION_STRING_HERE # Database connection string
      - JDBC_CONN_USER=INSERT_DATABASE_USER_HERE # Username to use when accessing the
database
      - JDBC_CONN_PASS=INSERT_JDBC_PASSWORD_HERE # Password for the database user
      - JDBC_CONN_ENCRYPTED=true # Flag for indicating if the database user's password
supplied is encrypted or not.
      - APP_MEMORY=4096 # The amount of memory in megabytes to assign to the Yellowfin
Application.
      #- CLUSTER_ADDRESS="yf-node-{{.Task.Slot}}" # Address to use for clustering -
recommended to use Docker networking to connect the containers. If left commented, it will
use the IP of the containers instead.
      - CLUSTER_PORT=7800 # TCP Port to use for cluster networking, used to connect between
containers
      -
NODE_BACKGROUND_TASKS=REPORT_BROADCAST_BROADCASTTASK,REPORT_BROADCAST_MIREPORTTASK,FILTER_C
ACHE,SOURCE_FILTER_REFRESH,SOURCE_FILTER_UPDATE_REMINDER,THIRD_PARTY_AUTORUN,ORGREF_CODE_RE
FRESH,ETL_PROCESS_TASK,SIGNALS_DCR_TASK,SIGNALS_ANALYSIS_TASK,SIGNALS_CLEANUP_TASK,COMPOSIT
E_VIEW_REFRESH,SIGNALS_CORRELATION_TASK # Comma separated list of which background Task
Types can be run on this node.
      - NODE_PARALLEL_TASKS=4,4,4,4,4,4,4,4,4,4,4,4,4 # Comma separated list of the number
of concurrent tasks for each Task Type that can be run on

   deploy:
     replicas: 2 # Number of Yellowfin instances to deploy
     labels:
        - "traefik.enable=true" # Tell Traefik to route to these instances, works with
exposedbydefault param in Traefik
        - "traefik.docker.network=yellowfin-cluster_yf-cluster" # IMPORTANT: This is will
allow Traefik to route to the Yellowfin instances. Format:
<%NameOfSwarmStack%>_<%NameOfDockerNetwork%>
        - "traefik.http.routers.yellowfin.rule=Host(`INSERT_DNS_HOSTNAME`)" #IMPORTANT: The
URL/DNS Name that you want Traefik to use for routing to your Yellowfin instances. Eg: `
yellowfin.example.com`
        - "traefik.http.routers.yellowfin.entrypoints=web" # Utilizes Traefik's web
entrypoint
        - "traefik.http.services.yellowfin.loadBalancer.server.port=8080" # Traefik to
route to the Yellowfin application port
        - "traefik.http.services.yellowfin.loadBalancer.sticky.cookie" # Enables sticky
sessions support
   networks:
```

```
        yf-cluster: # The network to add the Yellowfin instances to
    image: "yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>" # Path to your app-
only image of Yellowfin

  traefik:
    image: traefik # Pulls Traefik from Docker Hub
    command:
      - "--providers.docker.endpoint=unix:///var/run/docker.sock" #Gives Traefik access to
the Docker API
      - "--providers.docker.swarmMode=true" # Tells Traefik we're using Docker Swarm
      - "--providers.docker.exposedbydefault=true" # Exposed by default will auto-route to
any Docker containers/services that have the right labels.
      - "--providers.docker.network=yf-cluster" # The network Traefik should be monitoring.
Must match the one Yellowfin will be deployed in.
      - "--providers.docker.watch=true" # Watch for Docker Events
      - "--entrypoints.web.address=:80" # Run Traefik on port 80
      - "--api=true" # Enables the Traefik API
    ports:
      - 80:80 # Runs Traefik on port 80 (HTTP)
      #- 8090:8080 # Optional - Runs Traefik on port 8090 (if wanting to use the dashboard)
    volumes:
      # So that Traefik can listen to Docker events
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - yf-cluster # The network to add Traefik to.
    deploy:
      placement:
        constraints:
          - node.role == manager # IMPORTANT, only allows Traefik to deploy to a manager
node. This is recommended by Traefik.
networks:
  yf-cluster: # Creates a network using the overlay driver
    driver: overlay
```

With the above example, substitute in the database connection settings, which can be found in the existing Yellowfin installation, inside the Yellowfin web.xml file, a DNS name for Traefik to listen to for Yellowfin and also need to substitute in the value of what this Docker Stack will be named into the relevant Traefik labels.

Now run the following command in a terminal to deploy Yellowfin – which will deploy the Docker stack with the name “yellowfin-cluster” – which has been referenced in the deployment file above.

```
docker stack deploy --compose-file yellowfin-cluster.yml yellowfin-cluster
```

The result of the above deployment file example will be a Yellowfin cluster with 2 nodes, each with 4GB of RAM each, being fronted by Traefik, which will be listening on port 80 and routing requests to Yellowfin when it receives a request that matches the DNS hostname that was added to the Yellowfin Deploy Labels.

5.1 Deploying Yellowfin in Kubernetes environment

For deploying, Yellowfin in a Kubernetes environment, the following examples have been deployed against a Kubernetes environment via Kubectl.

Pre-requisites: A running Kubernetes cluster.

Beyond scope of this document but recommended: Running Yellowfin with HTTPS enabled.

5.1.1 Deploying a sandbox Yellowfin instance – The All-In-One image

The simplest Yellowfin deployment, to deploy a self-contained instance of Yellowfin, save the below YAML markup to a file. To follow out example, save the file to "*yellowfin-all-in-one.yml*"

```
---
### Yellowfin All in one Service ###
apiVersion: v1
kind: Service
metadata:
  name: yellowfin-all-in-one
spec:
  ports:
  - name: "web"
    port: 8080
    targetPort: 8080
  selector:
    app: yellowfin-all-in-one
  type: LoadBalancer
status:
  loadBalancer: {}
---
### Yellowfin All in one Deployment ###
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: default
  labels:
    app: yellowfin-all-in-one
  name: yellowfin-all-in-one

spec:
  replicas: 1
  selector:
    matchLabels:
      app: yellowfin-all-in-one
  template:
    metadata:
      labels:
        app: yellowfin-all-in-one
    spec:
      containers:
        - env:
          - name: APP_MEMORY
```

```
          value: "6144"
        name: yellowfin-all-in-one
        image: yellowfinbi/yellowfin-all-in-one:<RELEASE_VERSION_GOES_HERE>
        ports:
          - name: web
            containerPort: 8080
```

With this particular example, we've instructed Kubernetes to deploy Yellowfin behind a load balancer using the "type: LoadBalancer" attribute in the service definition, which when used with a Kubernetes environment that is running in a cloud environment (like AWS, Azure or GCP), will instruct Kubernetes to interact with the cloud provider to provision a load balancer that will route traffic to the Yellowfin service.

For a Kubernetes environment that can't auto-provision load balancers (eg: a basic On-premise deployment), "Service.Spec.Type" can be switched to "NodePort" and the "Service.Spec.Status" section removed, so that Kubernetes will deploy Yellowfin and make it accessible across the Kubernetes cluster on a specific port.

See the official Kubernetes documentation on the "LoadBalancer" and "NodePort" service types for more information.

To deploy the example, run the following command in a terminal.

```
kubectl apply -f yellowfin-all-in-one.yml
```

This will deploy a single instance of the (Yellowfin All-In-One image) into the Kubernetes cluster, allocated 6GB of RAM, and depending on if the Service specification in the deployment file was modified, will either deploy a load balancer to send traffic to the instance via a port on the Kubernetes cluster (Load Balancer) or will only publish a port on the Kubernetes cluster that can be used to communicate with the Yellowfin instance (NodePort).

5.1.2 Deploying a single instance of Yellowfin – App-Only image

For this deployment, a Yellowfin Repository database will need to be already created and synced with same version of Yellowfin that will be used in the Yellowfin container. The following example file will be utilizing the reverse proxy Traefik to proxy requests to the Yellowfin instance, as well as handling load balancing and sticky session support.

This is so that if amount of Yellowfin instances is scaled up in the deployment, from a single instance to a clustered deployment, the Kubernetes ingress routing will already be in place.

If Traefik is not an option in the deployment, this document will also be providing an example of how to run instances of Yellowfin without Traefik on Kubernetes in section 5.1.3, which can be modified to your needs.

5.1.2.a Installing Traefik using Helm

To Install Traefik on Kubernetes, Yellowfin recommends installing it via the Kubernetes package manager, Helm, which will be detailed below on how to install via the terminal.

```
## Installing Traefik on Kubernetes using Helm ##

# Step 1 - Check that instance has a connection to the Kubernetes Cluster that you want to
manage.
kubectl get svc
# If you are managing multiple Kubernetes clusters and are seeing services for a different
cluster, follow the steps in this guide for switching to the right cluster:
https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-
clusters/

# Step 2 - Install Helm 3 via the quick start script provided by the Helm maintainers. Helm
is going to assist us with getting Traefik up and running on our Kubernetes cluster.
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 > get_helm.sh
chmod 700 get_helm.sh
./get_helm.sh

# Step 3 - Add Traefik's Helm chart repository
helm repo add traefik https://containous.github.io/traefik-helm-chart
helm repo update

# Step 4 - Install Traefik on your Kubernetes cluster
helm install traefik traefik/traefik
## When the helm deployment finishes, it will setup Traefik to be an available Ingress type
into your Kubernetes cluster.
```

When the above commands are finished, the result will be a Traefik service running in the Kubernetes cluster, using the Kubernetes service type “LoadBalancer”.

5.1.2.b Deploying Yellowfin with Traefik as a proxy.

Now with Traefik installed, to deploy Yellowfin on Kubernetes, save the following deployment file to “*yellowfin-single-instance.yml*”

```
---
### Yellowfin Standalone Service ###
apiVersion: v1
kind: Service
metadata:
  name: yellowfin-standalone

spec:
  ports:
    - protocol: TCP
      name: web
      port: 8080
  selector:
    app: yellowfin-standalone
---
### Yellowfin Standalone Deployment ###
kind: Deployment
apiVersion: apps/v1
metadata:
  namespace: default
  name: yellowfin-standalone
  labels:
    app: yellowfin-standalone

spec:
  replicas: 1
  selector:
    matchLabels:
      app: yellowfin-standalone
  template:
```

```
    metadata:
      labels:
        app: yellowfin-standalone
    spec:
      containers:
        - env:
          - name: APP_MEMORY
            value: "4096"
          - name: JDBC_CLASS_NAME
            value: INSERT_DATABASE_TYPE_HERE
          - name: JDBC_CONN_ENCRYPTED
            value: "true"
          - name: JDBC_CONN_PASS
            value: INSERT_JDBC_PASSWORD_HERE
          - name: JDBC_CONN_URL
            value: jdbc:INSERT_JDBC_CONNECTION_STRING_HERE
          - name: JDBC_CONN_USER
            value: INSERT_DATABASE_USER_HERE
          name: yellowfin-standalone
          image: yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>
          ports:
            - name: web
              containerPort: 8080

---
### Yellowfin Standalone Ingress ###
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: yellowfinstandaloneingressroute
  namespace: default
spec:
  entryPoints:
    - web
  routes:
  - match: Host(`INSERT_DNS_HOSTNAME`)
    kind: Rule
    services:
    - name: yellowfin-standalone
      port: 8080
      sticky:
        cookie:
          httpOnly: true
          name: stickyCookie
```

With the above example, substitute in the database connection settings, which can be found in the existing Yellowfin installation, inside the Yellowfin web.xml file and a DNS name for Traefik to listen to, for routing requests to the Yellowfin instance.

To deploy the example, run the following command in a terminal.

```
kubectl apply -f yellowfin-single-instance.yml
```

This will deploy a single instance of the Yellowfin App-only image into the Kubernetes cluster with 4GB of RAM to Yellowfin, with Traefik forwarding requests to the Yellowfin instance when requests land on port 80 (standard HTTP port) of the DNS Hostname/Address that was substituted into the file.

When deploying Yellowfin with Traefik fronting it, the Yellowfin Kubernetes service will default to the “ClusterIP” service type in Kubernetes, so it will not expose any ports to the external interface of the Kubernetes cluster.

5.1.3 Deploying multiple discrete instances Yellowfin – App-Only image

As with the single-instance approach, for each discrete instance of Yellowfin to be deployed, a separate Yellowfin Repository database will need to be initialized, which needs to be synced with same version of Yellowfin that will be used in the Yellowfin container that is connecting to it.

To help provide different options for deploying Yellowfin on Kubernetes in an on-premises and cloud environments, the following example will not use the Traefik service that was used in section 5.1.2 for the single Yellowfin instance deployment. As mentioned in the single instance deployment, the single instance deployment of Yellowfin can be deployed without Traefik by taking the below deployment file and modifying it to only deploy one service. If you do want to use Traefik to route to multiple Yellowfin instances, take the deployment file that was used in section 5.1.2 and expand on it to add multiple discrete instances of Yellowfin that each have a different DNS Hostname/URL and are connecting to different Yellowfin Repository databases.

To deploy multiple instances of Yellowfin on Kubernetes, save the following deployment file to "*yellowfin-multi-instance.yml*"

```
---
### Yellowfin Production Instance - Service ###
apiVersion: v1
kind: Service
metadata:
  name: yellowfin-multi-instance-prod
spec:
  ports:
  - name: "web"
    port: 8080
    targetPort: 8080
  selector:
    app: yellowfin-multi-instance-prod
  type: LoadBalancer
status:
  loadBalancer: {}
---
### Yellowfin Development Instance - Service ###
apiVersion: v1
kind: Service
metadata:
  name: yellowfin-multi-instance-dev
spec:
  ports:
  - name: "web"
    port: 8080
    targetPort: 8080
  selector:
    app: yellowfin-multi-instance-dev
  type: LoadBalancer
status:
  loadBalancer: {}
---
### Yellowfin Production Instance - Deployment ###
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: default
  labels:
    app: yellowfin-multi-instance-prod
  name: yellowfin-multi-instance-prod

spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: yellowfin-multi-instance-prod
  template:
    metadata:
      labels:
        app: yellowfin-multi-instance-prod
    spec:
      containers:
        - env:
          - name: APP_MEMORY
            value: "6144"
          - name: JDBC_CLASS_NAME
            value: INSERT_DATABASE_TYPE_1_HERE
          - name: JDBC_CONN_ENCRYPTED
            value: "true"
          - name: JDBC_CONN_PASS
            value: INSERT_JDBC_PASSWORD_1_HERE
          - name: JDBC_CONN_URL
            value: jdbc:INSERT_JDBC_CONNECTION_STRING_1_HERE
          - name: JDBC_CONN_USER
            value: INSERT_DATABASE_USER_1_HERE
          name: yellowfin-multi-instance-prod
          image: yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>
          ports:
            - name: web
              containerPort: 8080
---
### Yellowfin Development Instance - Deployment ###
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: default
  labels:
    app: yellowfin-multi-instance-dev
  name: yellowfin-multi-instance-dev

spec:
  replicas: 1
  selector:
    matchLabels:
      app: yellowfin-multi-instance-dev
  template:
    metadata:
      labels:
        app: yellowfin-multi-instance-dev
    spec:
      containers:
        - env:
          - name: APP_MEMORY
            value: "4096"
          - name: JDBC_CLASS_NAME
            value: INSERT_DATABASE_TYPE_2_HERE
          - name: JDBC_CONN_ENCRYPTED
            value: "true"
          - name: JDBC_CONN_PASS
            value: INSERT_JDBC_PASSWORD_2_HERE
          - name: JDBC_CONN_URL
            value: jdbc: INSERT_JDBC_CONNECTION_2_STRING_HERE
          - name: JDBC_CONN_USER
            value: INSERT_DATABASE_USER_2_HERE
          name: yellowfin-multi-instance-prod
          image: yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>
          ports:
            - name: web
              containerPort: 8080
```

With the above example, substitute in the database connection settings, which can be found in the existing Yellowfin installation.

To deploy the example, run the following command in a terminal.

```
kubectl apply -f yellowfin-multi-instance.yml
```

This example will deploy the 2 Yellowfin instances based on the App-Only image using the "LoadBalancer" service type in Kubernetes, so that each Yellowfin instance can be accessed on port 8080 of the provisioned load balancer.

The "Production" instance of Yellowfin will be allocated 6GB of RAM, whilst the "Development" instance will be allocated 4GB of RAM. In a Kubernetes environment where a load balancer cannot be provisioned during the deployment (e.g. a basic on-premises environment), the services specification can be modified to switch the ServiceType to "NodePort", as referenced earlier in section 5.1.1.

5.1.4 Deploying a Yellowfin cluster – App-Only image

For this deployment, it is going to be nearly identical to deploying a single instance of Yellowfin in Kubernetes using the App-only image (section 5.1.2). Before starting the Yellowfin deploying, a Yellowfin Repository database will need to be created and synced with same version of Yellowfin that will be used in the Yellowfin cluster.

As with the "Yellowfin cluster on Docker Swarm" example (section 4.2.4), Yellowfin recommends a container-aware load balancer or reverse proxy that has sticky session support. This example will be using the Traefik reverse proxy to implement that recommendation via the use of sticky sessions.

For the Traefik on Kubernetes setup process, please refer to section 5.1.2.a.

Once Traefik is setup, please save the following to deployment file to "*yellowfin-cluster.yml*"

```
---
### Yellowfin Cluster Service ###
apiVersion: v1
kind: Service
metadata:
  name: yellowfin-cluster

spec:
  ports:
    - protocol: TCP
      name: web
      port: 8080
  selector:
    app: yellowfin-cluster
---
### Yellowfin Cluster Deployment ###
kind: Deployment
apiVersion: apps/v1
metadata:
  namespace: default
  name: yellowfin-cluster
  labels:
    app: yellowfin-cluster

spec:
  replicas: 2
```

```
  selector:
    matchLabels:
      app: yellowfin-cluster
  template:
    metadata:
      labels:
        app: yellowfin-cluster
    spec:
      containers:
        - env:
          - name: APP_MEMORY
            value: "6144"
          - name: CLUSTER_PORT
            value: "7800"
          - name: JDBC_CLASS_NAME
            value: INSERT_DATABASE_TYPE_HERE
          - name: JDBC_CONN_ENCRYPTED
            value: "true"
          - name: JDBC_CONN_PASS
            value: INSERT_JDBC_PASSWORD_HERE
          - name: JDBC_CONN_URL
            value: jdbc:INSERT_JDBC_CONNECTION_STRING_HERE
          - name: JDBC_CONN_USER
            value:INSERT_DATABASE_USER_HERE
          - name: NODE_BACKGROUND_TASKS
            value:
REPORT_BROADCAST_BROADCASTTASK,REPORT_BROADCAST_MIREPORTTASK,FILTER_CACHE,SOURCE_FILTER_REF
RESH,SOURCE_FILTER_UPDATE_REMINDER,THIRD_PARTY_AUTORUN,ORGREF_CODE_REFRESH,ETL_PROCESS_TASK
,SIGNALS_DCR_TASK,SIGNALS_ANALYSIS_TASK,SIGNALS_CLEANUP_TASK,COMPOSITE_VIEW_REFRESH,SIGNALS
_CORRELATION_TASK
          - name: NODE_PARALLEL_TASKS
            value: 4,4,4,4,4,4,4,4,4,4,4,4,4
          name: yellowfin-cluster
          image: yellowfinbi/yellowfin-app-only:<RELEASE_VERSION_GOES_HERE>
          ports:
            - name: web
              containerPort: 8080

---
### Yellowfin Cluster Ingress ###
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: yellowfiningressroute
  namespace: default
spec:
  entryPoints:
    - webs
  routes:
  - match: Host(`INSERT_DNS_HOSTNAME`)
    kind: Rule
    services:
    - name: yellowfin-cluster
      port: 8080
      sticky:
        cookie:
          httpOnly: true
          name: stickyCookie
```

With the above example, substitute in the database connection settings, which can be found in the existing Yellowfin installation, inside the Yellowfin web.xml file and a DNS name for Traefik to listen to, for routing requests to the Yellowfin instance.

To deploy the example, run the following command in a terminal.

```
kubectl apply -f yellowfin-cluster.yml
```

This will deploy a 2 node Yellowfin cluster into the Kubernetes environment, with each node allocated 6GB of RAM, with Traefik forwarding requests to the Yellowfin instances when requests go to port 80 (standard HTTP port) of the DNS Hostname/Address that was substituted into the file. Traefik will ensure user requests are directed to the same Yellowfin instance for the duration of their session via the use of a cookie that will be added to the user's browser session.

When deploying Yellowfin with Traefik fronting it, the Yellowfin Kubernetes service will default to the "ClusterIP" service type in Kubernetes, so it will not expose any ports to the external interface of the Kubernetes cluster.

6. Other considerations

6.1 Yellowfin licensing and Docker containers

Depending on the Yellowfin deployment chosen, this will influence the type of Yellowfin license that will be required.

For a non-clustered instance of Yellowfin, where only 1 container is running per Yellowfin deployment, a standard Yellowfin license can be used. The container's hostname can be changed to match the Yellowfin license.

For a clustered deployment of Yellowfin, it is recommended to use a Yellowfin Wildcard license alongside a hostname that has a static prefix present, to cover multiple containers.

- For example, in a Docker Swarm Stack file, under the Yellowfin service specification, the “hostname” attribute can be set to “yellowfin-{{.Task.Slot}}", which will provide a unique hostname for each container in that Yellowfin deployment, and can be covered by a Yellowfin Wildcard License that has “yellowfin-*” as the licensed hostname.
- To obtain a wildcard license, please contact a Yellowfin Account Manager or reach out to the Yellowfin sales team.

6.2 Yellowfin log files

The logging method used in the examples in this document all leverage the default logging driver for the environment they are in, which by default for Docker and Kubernetes is the “json-file” logging driver. Whilst a Yellowfin container is running, its logs are written to a JSON file on the host that it is running on by the logging driver, which manage the logs for the lifetime of the container. Once the container is removed, the logs will be removed with it.

For deployments where you need to be able to view and retain the logs of a Yellowfin container, irrespective of it still running or it being replaced by another container – like a production deployment - Yellowfin recommend shipping the logs to a centralized logging platform using shipping agents that integrate with Docker/Kubernetes.

For more information about logging in Docker and Kubernetes, please refer to Docker Logging Best Practices and Kubernetes Logging Architecture.

6.3 Upgrading Yellowfin on Docker and Kubernetes

With the 2 Yellowfin Docker Images that are being released alongside this white paper, only the App-only image allows the upgrading of Yellowfin whilst retaining content that has been created.

The All-In-One Yellowfin Image is a sandboxed deployment, and due to the way that the Yellowfin and PostgreSQL components are unpacked into the Docker Image during the Image build process, Yellowfin can't be upgraded without building a new Docker Image and replacing the existing container with a new one – which will trigger the removal of all the content that exists on the original sandbox container.

For the App-Only image, the upgrade process is similar to a standard Yellowfin Cluster upgrade, with a few extra steps to account for a Docker/Kubernetes deployment:

1) First, use the same process that was used to build the App-only Docker image, but this time substituting in the new Yellowfin Installer Jar that matches the version that you are upgrading to.
    a. For deploying across multiple Docker/Kubernetes servers, once the Docker Image is built, push the updated image to a Docker Registry.
2) Next, all the running Yellowfin containers for the deployment being upgraded need to be stopped.
    a. This can be done via scaling the number of Yellowfin instances deployed down to 0, using the deployment commands specific to each platform.
    b. By doing this, it leaves the configuration for the Yellowfin deployment intact, such as connection settings and environment variables that were passed in at deployment time.
3) Now, it is recommended by Yellowfin to take a backup of the Yellowfin Repository Database prior to upgrading, as this provides a rollback point in the unlikely situation that the upgrade fails, and something goes wrong in the database.
    a. For this, follow the standard database backup process recommended by your database vendor.
4) Using the Yellowfin upgrade Jar that matches the Yellowfin version that is being upgraded to, execute the following command to upgrade the Yellowfin Repository Database, substituting in the values that match the target Yellowfin environment:
    a. `java -jar yellowfin-20200701-update.jar -silent option.upgrade=database jdbcUser=Yellowfin_DB_Admin jdbcPassword=Yellowfin_DB_PASSWORD jdbcURL=jdbc:mysql://TARGET_DATABASE:3306/yellowfin`
    b. For a full breakdown of the parameters in the above command, please see the Yellowfin Wiki.

5) Finally, bring the Yellowfin instances back online and use the new Docker image.
    a. This can be done using the platform's native command to update the Yellowfin deployment Docker Image, and once that is completed, calling another platform native command to set the number of instances back to the pre-upgrade value using the same scaling command as earlier.
    b. As an example, in a Docker Swarm environment, the above 2 commands for a Yellowfin deployment would be executed in the following order
        i. `docker service update --image My_Docker_Registry:yellowfin-9.5.1 yellowfin` which instructs the Docker engine to update the image associated with the "yellowfin" deployment to the "yellowfin-9.5.1" Docker image found in the Docker Registry "My_Docker_Registry".
        ii. `docker service scale yellowfin=1` sets the amount of Yellowfin instances that we want running in the deployment to 1, which will bring the Yellowfin deployment back online, and using the new image.

7. Closing notes

Whilst Yellowfin aims to make the included examples in this white paper work across all the major cloud providers, as well as on-premises environments, most of the testing was done in an AWS environment using the following services:

- EC2 instances for the Docker and Docker Swarm environments

- AWS EKS for the Kubernetes environment

- AWS ECR providing all the environments a Docker registry to pull the Yellowfin Docker images from.

Modifications to the deployment files may be required on other cloud environments

Any feedback or questions relating to this document or how to deploy Yellowfin in a Docker or Kubernetes environment, please reach out support@yellowfin.bi.