Milpa

Milpa is a cloud-native management controller that runs your software and transparently manages the underlying infrastructure. Milpa's unique workflow allows developers to focus on delivering applications instead of managing servers. When launching an application, Milpa will provision a cloud instance for your application, setup cloud firewall rules, register the application in service discovery and coordinate the rollout of application updates. With milpa, there are no VMs to provision. Your cluster size automatically scales up when you need to run more applications and scales down when those applications are no longer running.

Overview

At the core of Milpa are basic types that can be used to describe an application and supporting resources needed by that application. Milpa has borrowed the same easy to understand types and abstractions used by Kubernetes making it an easy to use platform for running applications in the cloud. Milpa currently implements Pods, Nodes, Services, ReplicaSets, Deployments, Secrets, Events and LogFiles. More types will be added in future releases.

Pods are the most basic level of computing that users will interact with in Milpa. A pod consists of one or more applications that should be colocated on a compute instance and work together. Milpa pods can be as simple as a single process running a web server or a more complex set of applications that must be colocated to work together. In the microservices world, a pod is typically composed of a primary application and one or more supporting processes that help with logging (e.g. fluetd), networking (e.g. consul or synapse) or application monitoring (e.g. New Relic, AppDynamics).

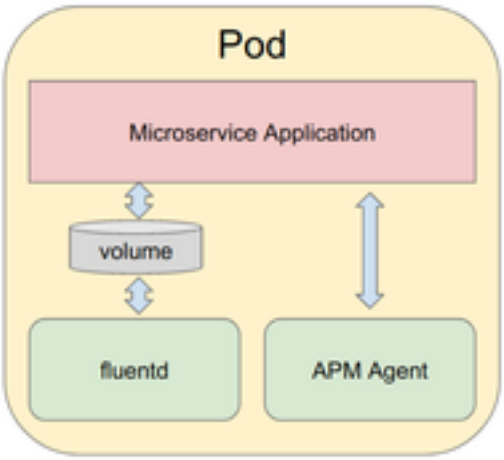

Figure 1: Milpa Pod

In Milpa, the individual applications that make up a pod are called Units. Units are analogous to containers in Kubernetes or Tasks in Nomad. Each Unit is identified by a unique name, a pointer to a repository in a remote Docker registry, and a command that will be executed to run the Unit.

Pods are run on a cloud instance which is called a Node in Milpa. Each Pod runs on its own Node and unlike other cluster managers, Pods are never colocated on a Node. Isolating Pods on their own Node results in a simple system for application deployment and also improves multi-tenant

application security, isolation and debuggability as all Units in the Pod share cpu, memory and network resources on the node. This is different from the Docker world which enforces a stronger but more complex layer of isolation for cpu, memory and networking in order to solve application multi-tenancy issues. While cpu, memory and networking are shared in Milpa, each Unit gets its own filesystem namespace on a Milpa Node. This makes packaging and deploying Units simple and conflict free.

A note about use of Docker in Milpa

Milpa uses Docker container images as the primary application packaging system but does not run Docker daemon on the Nodes. Docker makes a fantastic packaging system for applications but running Docker daemon on production instances is an unnecessary layer in the Milpa system. As Milpa matures, we plan to support more process isolation features but not at the expense of an easy to use, debuggable and secure system.

Installation

Milpa Requirements

- An AWS account with permissions to create EC2 instances and resources, write to DynamoDB and Route 53.
- Milpa Nodes must run in a VPC (Milpa does not support VPC classic).
- Milpa should be installed on an instance in the EC2 VPC where instances should be launched or have network access to the VPC.
- The account Milpa runs under (specified in server.yml) must have permission to access Elotl's AMIs.
- For high availability setups, access to an external etcd cluster is required.
- Milpa requires approximately 1 CPU core per 200 pods it will control (this will be improved in future releases).
- Please validate that your AWS Instance Limits are above your expected max Pod count.
- Milpa is benchmarked to scale up to 1400 Pods.
- Milpa is tested and certified to run on Ubuntu 14.04, Ubuntu 16.04, Debian 8. Other flavors of Linux are expected to work, but not recommended since they are untested.

Installation Steps

Run the installer

```
$ chmod +x milpa-installer-latest
$ sudo ./milpa-installer-latest
Verifying archive integrity...  100%   All good.
Uncompressing Milpa installer  100%

An example server configuration file has been created at
/opt/milpa/etc/server.yml. Please review it before enabling Milpa.

2018-04-09 17:33:31.687229 I | embed: listening for peers on http://localhost:2380
2018-04-09 17:33:31.687953 I | embed: listening for client requests on localhost:2379
2018-04-09 17:33:31.719386 I | etcdserver: name = default
2018-04-09 17:33:31.719675 I | etcdserver: data dir = /opt/milpa/data
2018-04-09 17:33:31.719962 I | etcdserver: member dir = /opt/milpa/data/member
2018-04-09 17:33:31.720268 I | etcdserver: heartbeat = 100ms
2018-04-09 17:33:31.720610 I | etcdserver: election = 1000ms
2018-04-09 17:33:31.721178 I | etcdserver: snapshot count = 100000
2018-04-09 17:33:31.722535 I | etcdserver: advertise client URLs = http://localhost:2379
2018-04-09 17:33:31.723093 I | etcdserver: initial advertise peer URLs = http://localhost:2380
2018-04-09 17:33:31.723579 I | etcdserver: initial cluster = default=http://localhost:2380
2018-04-09 17:33:31.729116 I | etcdserver: starting member 8e9e05c52164694d in cluster cdf818194e3a8
2018-04-09 17:33:31.730468 I | raft: 8e9e05c52164694d became follower at term 0
2018-04-09 17:33:31.730927 I | raft: newRaft 8e9e05c52164694d [peers: [], term: 0, commit: 0, applie
2018-04-09 17:33:31.731364 I | raft: 8e9e05c52164694d became follower at term 1
2018-04-09 17:33:31.738111 W | auth: simple token is not cryptographically signed
2018-04-09 17:33:31.740882 I | etcdserver: starting server... [version: 3.3.2, cluster version: to_b
2018-04-09 17:33:31.748424 I | etcdserver/membership: added member 8e9e05c52164694d [http://localhos
```

```
2018-04-09 17:33:32.432695 I | raft: 8e9e05c52164694d is starting a new election at term 1
2018-04-09 17:33:32.434749 I | raft: 8e9e05c52164694d became candidate at term 2
2018-04-09 17:33:32.435674 I | raft: 8e9e05c52164694d received MsgVoteResp from 8e9e05c52164694d at
2018-04-09 17:33:32.438561 I | raft: 8e9e05c52164694d became leader at term 2
2018-04-09 17:33:32.442861 I | raft: raft.node: 8e9e05c52164694d elected leader 8e9e05c52164694d at
2018-04-09 17:33:32.448597 I | etcdserver: setting up the initial cluster version to 3.3
2018-04-09 17:33:32.452949 N | etcdserver/membership: set the initial cluster version to 3.3
2018-04-09 17:33:32.459642 I | etcdserver/api: enabled capabilities for version 3.3
2018-04-09 17:33:32.465462 I | embed: ready to serve client requests
2018-04-09 17:33:32.469325 N | embed: serving insecure client requests on 127.0.0.1:2379, this isstr

After reviewing /opt/milpa/etc/server.yml, please start Milpa via:

initctl start milpa


Client tarball is available at /home/vagrant/milpactl.tar.gz. You can
copy this tarball to any machine you would like to install milpactl on.
```

Milpa and all the files and directories it needs will be placed under `/opt/milpa`. The installer will also create a symlink in `/usr/local/bin` for `milpactl`. You might want to add `/usr/local/bin` to your `PATH` to make it easier to use `milpactl`.

Edit `/opt/milpa/etc/server.yml`.

Mandatory: Make sure your AWS keys are set, and your license key is in `server.yml` configuration file.

```
$ sudo vi /opt/milpa/etc/server.yml
```

Optional: Please refer to the Server Configuration section for more information on rest of key values in `server.yml`.

Start Milpa

```
$ sudo initctl start milpa
```

on upstart-based systems (like Ubuntu 14.04) or

```
$ sudo systemctl start milpa
```

on systemd-based distributions (Ubuntu 16.04 or Debian Stretch).

Set Permissions on milpactl (Optional)

By default, any user is allowed to use `milpactl` to interact with Milpa on the system. If this is not desired, permissions on the `milpactl` client private key can be changed:

```
vagrant@vagrant-ubuntu-trusty-64:~$ ls -l /opt/milpa/certs/
total 24
-rw-r--r-- 1 milpa milpa 501 Apr 16 23:02 ca.crt
-rw------- 1 milpa milpa 227 Apr 16 23:02 ca.key
```

```
-rw-r--r-- 1 milpa milpa 639 Apr 16 23:02 client.crt
-rw-r--r-- 1 milpa milpa 227 Apr 16 23:02 client.key
-rw------- 1 milpa milpa 639 Apr 16 23:02 server.crt
-rw------- 1 milpa milpa 227 Apr 16 23:02 server.key
vagrant@vagrant-ubuntu-trusty-64:~$ milpactl version
Milpa server version: 1.0.0
vagrant@vagrant-ubuntu-trusty-64:~$ sudo chmod 0600 /opt/milpa/certs/client.key
vagrant@vagrant-ubuntu-trusty-64:~$ ls -l /opt/milpa/certs/
total 24
-rw-r--r-- 1 milpa milpa 501 Apr 16 23:02 ca.crt
-rw------- 1 milpa milpa 227 Apr 16 23:02 ca.key
-rw-r--r-- 1 milpa milpa 639 Apr 16 23:02 client.crt
-rw------- 1 milpa milpa 227 Apr 16 23:02 client.key
-rw------- 1 milpa milpa 639 Apr 16 23:02 server.crt
-rw------- 1 milpa milpa 227 Apr 16 23:02 server.key
vagrant@vagrant-ubuntu-trusty-64:~$ milpactl version
Failed to dial server: Error loading TLS configuration from /opt/milpa/certs:
error loading key pair /opt/milpa/certs/client.crt /opt/milpa/certs/client.key:
open /opt/milpa/certs/client.key: permission denied
Fell back to current directory /home/vagrant for certs and encountered the
following error: errorloading key pair /home/vagrant/client.crt
/home/vagrant/client.key: open /home/vagrant/client.crt: no such file or directory
```

Copy Client to Remote Machines (Optional)

The installer creates a tarball from `milpactl`, the command line tool that is used to interact with Milpa, and the certificates `milpactl` and `milpa` uses for authentication, in the home directory of the user that runs the installer. This tarball can be copied to other boxes if needed:

```
ubuntu@milpa-server-box:~$ scp milpactl.tar.gz milpa-client-box:
milpactl.tar.gz                                           100% 6395KB   3.1MB/s   00:02
ubuntu@milpa-server-box:~$ ssh milpa-client-box
ubuntu@milpa-client-box:~$ tar xvzf milpactl.tar.gz
milpactl/
milpactl/client.crt
milpactl/milpactl
milpactl/ca.crt
milpactl/client.key
ubuntu@milpa-client-box:~$ cd milpactl/
ubuntu@milpa-client-box:~/milpactl$ ./milpactl --endpoints=milpa-server-box:54555 version
2018-04-09 23:06:22.463317 I | Milpa server version: 0.01dev
ubuntu@milpa-client-box:~/milpactl$
```

Server Configuration

Example server configuration file:

```
apiVersion: v1
clusterName: your-cluster-name
cloud:
  aws:
    region: us-east-1
    accessKeyID: FILL_IN
    secretAccessKey: FILL_IN
    vpcID: default
    imageOwnerID: 689494258501
serviceDiscovery:
  privateDNS:
    ttl: 30
etcd:
  client:
    endpoints: []
    certFile: ""
    keyFile: ""
    caFile: ""
  internal:
    dataDir: /opt/milpa/data
    configFile: ""
nodes:
  firewallMode: OpenToVPC
  standbyNodes:
    - instanceType: "t3.micro"
      count: 2
      spot: false
  defaultInstanceType: t3.nano
  bootImageTags:
    company: elotl
  defaultFirewallRules:
    - protocol: TCP
      port: 22
    - protocol: ICMP
      port: -1
  cloudInitFile: /opt/milpa/etc/cloudinit.yml
license:
  username: name@example.com
  password: password123
  id: provided-by-elotl
  key: provided-by-elotl
```

Cluster Name

The field `clusterName` will be used as the name of the cluster managed by this Milpa instance. A Milpa cluster will tag all cloud resources with the cluster name and attempt to manage any resources

that have tags matching its cluster name. If two Milpa installs have the same cluster name, confusion and inadvertent deletes of VMs and security groups are bound to happen. Milpa will attempt to prevent collisions between cluster names by registering the cluster name in a DynamoDB table.

Cloud Configuration

The `cloud` section in the server configuration configures the cloud provider for Milpa. Right now, only `aws` (Amazon Web Services) is supported.

Supported fields in `aws`:

- `region`: This is the AWS region where Milpa will create instances. Example: `us-east-1`. The environment variable `AWS_REGION` can also be used instead and will override the value in server.yml.
- `accessKeyID`: This is the AWS access key ID Milpa will use for interacting with the AWS API. The environment variable `AWS_ACCESS_KEY_ID` can also be used instead and will override the value in server.yml. If this value and the environment variable are empty, Milpa will use AWS shared configuration credentials from a user's ~/.aws directory. If there is not a shared credentials folder on the machine and Milpa is running on a machine inside AWS, Milpa will use credentials from an IAM role in the instance profile assigned to the machine. For more inforation on IAM roles for instances, please see the AWS EC2 Documentation.
- `secretAccessKey`: This is the AWS secret access key. The environment variable `AWS_SECRET_ACCESS_KEY` can also be used instead. Milpa uses the same precedence rules to determine the `secretAccessKey` value as it does for `accessKeyID`.
- `vpcID`: This is the VPC ID in which Pod instances will be created. If empty, Milpa will attempt to use the VPC the controller is located in. This can also be set to "default" to select the default VPC.
- `imageOwnerID`: This is the AWS owner ID for the AMIs Milpa is allowed to use. This should be the owner ID of Elotl, 689494258501.

Example:

```
cloud:
  aws:
    region: us-east-1
    accessKeyID: FILL_IN
    secretAccessKey: FILL_IN
    vpcID: "default"
    imageOwnerID: 689494258501
```

Service Discovery Configuration

The `serviceDiscovery` section in the server configuration configures service discovery. Right now, only `privateDNS` is supported. Milpa will register Services and Pods in private DNS using these settings.

Supported fields:

- `ttl`: the DNS TTL (time to live), in seconds, that Milpa will use for registering Services and Pods.

Example:

```
serviceDiscovery:
  privateDNS:
    ttl: 30
```

Etcd Storage Configuration

Milpa can be configured to use either an existing etcd cluster or run its own embedded etcd storage engine to persist all created objects and cluster configuration. The `etcd` section controls the behavior of etcd and allows the user to choose between these two configurations.

External Etcd Cluster

To use an external etcd cluster (a setup encouraged for fault tolerant setups), specify a list of client.endpoints of the etcd cluster. For a cluster protected by TLS certs, you can also specify client cert, key and CA files. See the etcd documentaiton for information on generating certificates for your etcd cluster.

Supported fields:

- `endpoints`: The list of servers participating in the etcd cluster. Should be specified in the form '127.0.0.1:2379' or http://127.0.0.1:2379 or https://127.0.0.1:2379 . If the endpoint list is empty, the internal etcd server will be used.
- `certFile`: Path to the client-server TLS certificate file.
- `keyFile`: Path to the client-server TLS key file.
- `caFile`: Path to the client-server TLS CA file.

Example:

```
etcd:
  client:
    endpoints: [https://localhost:2379, https://172.20.5.12:2379]
    certFile: ""
    keyFile: ""
    caFile: ""
```

Embedded Etcd

For development, testing and possibly CI setups, Milpa can use its own embedded etcd database. Using the internal etcd server simplifies number of pieces you'll need to provision to get Milpa running. The internal etcd database will be used if no endoints are specified in the `etcd.client.endpoints` setting. It's likely etcd's default settings will suffice for most users but it's possible to customize the embedded etcd storage engine similar to how one would customize a standalone etcd installation. To customize the internal etcd database, the etcd.internal.configFile parameter should point to an etcd configuration file. An example of such a file can be found at https://github.com/coreos/etcd/blob/master/etcd.conf.yml.sample

Supported fields:

- `dataDir`: The directory that will be used by etcd for storage. The Milpa user must have write access to the directory. Defaults to `/opt/milpa/data`. The data directory can also be specified in the `configFile`, but should not be specified in both locations.

- `configFile`: Path to an etcd configuration file that will be used to further customize the behavior of etcd. All etcd command-line flags are supported. See the etcd documentation for more information.

Example:

```
etcd:
  internal:
    configFile: /opt/milpa/etc/etcd.yml
```

Example Contents of Etcd Configuration File:

```
data-dir: /opt/milpa/data
snapshot-count: 10000
heartbeat-interval: 100
election-timeout: 1000
max-snapshots: 5
```

Node Configuration

In the server configuration, the `nodes` section configures default Node parameters.

Fields:

- `standbyNodes`: Used to specify pools of standby Nodes Milpa will keep so that newly created pods can be dispatched to nodes quickly. `standbyNodes` should be a list of `standbyNode` values that contain three parameters: `instanceType`: the name of the cloud instance type, `spot`: whether the instance should be a spot instance, and `count`: the number of standby instances of this type.
- `defaultInstanceType`: This is the default cloud instance type Milpa will use when creating Pods if the Pod does not set an instance type or specify the resources the Pod requires.
- `bootImageTags`: This is a dictionary of image tags. When creating nodes, Milpa will use the latest available image that is tagged with all these tags.
- `defaultVolumeSize`: A resource field specifying the default size for the root volume of nodes in bytes. To set the root volume to 8 GiB, specify the value as "8Gi". Optional field.
- `cloudInitFile`: Path to a cloudInitFile that will be used to provision all Nodes that Milpa boots. Milpa will detect modifications to this file and reload the file when it changes. Nodes started after a modification to the file will get the updated cloudInit file. For more information please see the Cloud-Init section in this document.
- `firewallMode:` If the firewall mode is set to `OpenToVPC`, Milpa will set the cloud firewall (Security Groups in AWS) to allow traffic from the local VPC by default. For all other traffic, Services must be used to open ports to ingress traffic from outside the VPC. If the firewall mode is set to `Closed` then one or more Services must be defined to allow connections to Milpa Nodes. Defaults to `OpenToVPC`.
- `defaultFirewallRules`: Use this to specify ports that should be opened for all Milpa Nodes. These rules will be added to a security group (AWS) or firewall rules (GCE) that are assigned to each Milpa Node. This setting can be used to open ports on all Milpa Nodes if `firewallMode` is set to `Closed`.

Example:

```
nodes:
  firewallMode: OpenToVPC
  standbyNodes:
```

```
    - instanceType: "t3.micro"
      count: 2
      spot: false
  defaultInstanceType: t3.nano
  bootImageTags:
    company: elotl
    version: 1.0.3
  defaultFirewallRules:
    - protocol: TCP
      port: 22
    - protocol: ICMP
      port: 0
  cloudInitFile: /opt/milpa/etc/cloudinit.yml
```

License Configuration

Licenses for customers are created and managed by Elotl. Your license might limit the number of machines that are allowed to run Milpa, or the time period when Milpa can be used with the same license id and key.

Fields:

- `username`: The username used for this license. One user might be associated with multiple licenses.
- `password`: The password for the username.
- `id`: The license ID.
- `key`: The license key.
- `file`: The file into which Milpa will save license-related information. Optional. By default, Milpa will use `$HOME/.milpa.license`.

Example:

```
license:
  username: name@example.com
  password: password123
  id: provided-by-elotl
  key: provided-by-elotl
  file: /var/lib/milpa/license.data
```

Disabling Usage Reporting

Milpa reports back cloud usage statistics to Elotl. This can be disabled via the `disableUsageReporter` flag. Example:

```
# Disable sending back usage statistics.
disableUsageReporter: true
```

Milpactl

Milpactl is the command line interface used to interact with Milpa. A user can create, read, update and delete Milpa resources using milpactl. Below are a couple of examples of these operations.

```
# Multiple resources can be specified in a single resource.yml file if
# they are separated by `---\n`
$ milpactl create -f <path/to/resource.yml>
$ milpactl update -f <path/to/resource.yml>
$ milpactl delete -f <path/to/resource.yml>
$ milpactl delete <resourceKind> <resourceName>
# Use --cascade=false to prevent deleting child resources
$ milpactl delete --cascade=false <resourceKind> <resourceName>
$ milpactl get <resourceKind> <resourceName>
# Use --output=(json|yaml) to dump the entire resource object
$ milpactl get --output=json <resourceKind> <resourceName>
$ milpactl get --output=yaml <resourceKind> <resourceName>
```

Supported Abreviations

- deploy: deployment
- ep: endpoints
- ev: event
- no: node
- po: pod
- rs: replicaset
- svc: service

Logs

Get logs from a running or finished pod. Also can be used to get Node logs.

```
# get logs from a Pod, if no unitName is specified, return the
# logs for the first Unit if there is only one Unit in the Pod.
$ milpactl logs <podName> -u <unitName>

# follow Milpa logs for a running Pod, similar to "tail -f"
$ milpactl logs -f <podName> -u <unitName>

# get Milpa agent logs from a Node
$ milpactl logs <nodeName>
```

Scale

Updates the number of replicas of a Deployment or ReplicaSet.

```
# Scale the deployment named mysql to 3.
$ milpactl scale --replicas=3 deployment mysql
```

```
# Scale the Deployment or ReplicaSet in resource.yml
$ milpactl scale --replicas=3 -f resource.yaml
```

Port Forwarding

Port forwarding allows you to forward local ports to a Milpa Node.

```
# Listen on ports 5000 and 6000 locally, forwarding data to/from
# ports 5000 and 6000 in the Pod
$ milpactl port-forward mypod 5000 6000

# Listen on port 8888 locally, forwarding to 5000 in the Pod
$ milpactl port-forward mypod 8888:5000

# Listen on a random port locally, forwarding to 5000 in the Pod
$ milpactl port-forward mypod :5000

# Listen on a random port locally, forwarding to 5000 in the Pod
$ milpactl port-forward  mypod 0:5000
```

Exec

Execute a command in the context of a Unit in a Pod.

```
# Get output from running 'date' from Pod "mypod", using the first Unit by default
milpactl exec mypod date

# Get output from running 'date' in python-unit from Pod "mypod"
milpactl exec mypod -u python-unit date

# Switch to raw terminal mode, send stdin to 'bash' in python_unit from Pod "mypod"
# and send stdout/stderr from 'bash' back to the client
milpactl exec mypod -u python-unit -it -- /bin/bash -il
```

Attach

Attach to a process that is already running inside an existing Unit. This can be used to send stdin commands to the process. You can see stdout from a process using the "milpactl logs" command.

```
# Get output from running Pod "mypod", using the first Unit by default
milpactl attach mypod

# Get output from python-unit from Pod "mypod"
milpactl attach mypod -u python-unit

# Attach to python-unit in Pod "mypod", send stdin to the Unit
# and send stdout/stderr from python-unit back to the client
milpactl attach mypod -u python-unit -i
```

Usage Reports

Milpa tracks the usage of cloud resources and allows you to query current usage as well as historical usage. Usage reports can be filtered using label selectors.

Usage is broken into 4 buckets

- Instance - tracks the usage of cloud instances.
- Peripheral - tracks usage of additional attached GPUs and TPUs but not GPUs that are part of an AWS machine type.
- Storage - tracks the usage of attached storage in GB hours.
- Network - tracks usage of Load Balancers and other network related cloud resources. Milpa does not track egress or ingress traffic.

```
# Query current hourly usage
$ milpactl usage

# Query usage from May 1st until the present.
# The date parser understands most date and time formats.
$ milpactl usage --start-date='20170501'

# Query usage from May 1st until June 1st
$ milpactl usage --start-date='20180501' --end-date='20170601'

# Query usage from May 1st until the present but filter using label selector
$ milpactl usage --start-date='20180501' -l 'env in (dev, qa), app=writer'
```

Currently, usage data is kept for 90 days.

We plan to make tracking of cloud costs available in a future release.

Other Operations

- `version`: get the version of the Milpa server
- `dump`: Milpactl can also dump the internal state of controllers. This is primarily useful for debugging Milpa and shouldn't be needed by a user.

Secure Communication

By default milpactl communicates with milpa over a TLS secured connection. Certificates for Milpactl are created during the initial install of the server (default location /opt/milpa/certs). A different location for certificates can be specified on the command line with the `--cert-dir` option.

To disable security communication entirely (for development and testing work, strongly discouraged for any production installs), start Milpa with the `--disable-tls` command line flag and use `--disable-tls` with all Milpactl commands.

Multiple Milpa Servers

Multiple servers in a fault tolerant setup can be specified with the `--endopints` argument. Milpa will attempt to contact the master for write operations but will choose a random server from the

endpoints list for read operations.

Cloud-init

Milpa supports provisioning new Nodes through a subset of functionality provided by the popular cloud-init system. Users can specify a cloud-init file in server.yml and the cloud-init file will be applied when a Node is booted by Milpa.

Milpa's cloud-init system provides the following initialization functions:

- Initialize users and set SSH authorized keys
- Set SSH authorized keys for the root user
- Set the hostname
- Write arbitrary files (allowed encodings: plain text, base64, gzip and gzip+base64)
- Execute scripts

Cloud-init Example

```
users:
  - name: "dbowie"
    passwd: "$6$qhNlkpFW$p.YhGhk1zFd0bQ1Quk/3O42qEtp7vjZ5DB8C/l/VUB..."
    groups:
      - "wheel"
    ssh_authorized_keys:
      - "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDPU7h8CaYA1VH/CwY3Ah..."

# without a user, ssh_authorized_keys will be added to the root user
ssh_authorized_keys:
  - "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC0g+ZTxC7weoIJLUafOgrm+h..."

write_files:
  - content: |
        [Unit]
        Description=Socket for the API

        [Socket]
        ListenStream=2375
        Service=docker.service

        [Install]
        WantedBy=sockets.target
    owner: root:root
    path: /root/config
    permissions: "0644"
  - encoding: gzip
    content: !!binary |
        H4sIAIDb/U8C/1NW1E/KzNMvzuBKTc7IV8hIzcnJVyjPL8pJ4QIA6N+MVxsAAAA=
    path: /usr/bin/hello
    permissions: "0755"

hostname: foo.bar.com
```

```
runscript: |
  #!/bin/ash
  echo "dbowie   ALL=(ALL)   NOPASSWD: ALL" >> /etc/sudoers.d/dbowie
  apk update
  apk add curl
```

Reloading and Limits

The cloud-init file can be updated while Milpa is running so it's not necessary to restart Milpa. An updated cloud config will only be applied to Nodes started after the file has been updated.

Limitations

The cloud-init file is served to Milpa Nodes through the EC2 User Data. User Data is limited to a size of 16KB. Internally, Milpa uses approximately 4KB of the User Data leaving 12K of data for a user's cloud-init file. If the cloud-init file is too large, Milpa will not be able to start Nodes. Please contact Elotl Support if 12KB is not sufficient.

Milpa Types

The Milpa API types are functionally equivalent to the corresponding Kubernetes API types. When a Milpa type doesn't support a Kubernetes feature, the corresponding type attributes are omitted from the Milpa type. Milpa also has additional types and type attributes to support functionality that is unique to Milpa. All attributes are described in the API section of this document.

The supported API types for Milpa are

- Node
- Pod
- Service
- ReplicaSet
- Deployment
- Secret
- Event
- LogFile
- Endpoints

In the future we plan to add support for additional API types.

Nodes

A Node represents a cloud instance. Nodes are created by Milpa to service new Pod requests and can also be buffered as standby Nodes to support faster application dispatch.

Node lifecycle phases

Nodes follow a fairly linear progression through phases from creation to termination.

- **Creating**: A Node has been created in Milpa and a request for a new instance has been sent to the cloud provider. The cloud provider has not replied to the request for a new instance.
- **Created**: The cloud provider has replied to the request for a new instance and the new instance is booting. The Node has an `instanceID`.
- **Available**: The Node has finished booting and has been assigned an IP address. Waiting Pods can be dispatched to the Node.
- **Claimed**: A Pod has claimed the Node and is either dispatching to the Node or the Pod is running on the Node.
- **Cleaning**: The Node is no longer claimed and is cleaning up resources inside Milpa before being terminated.
- **Terminating**: The Node is in the process of shutting down.
- **Terminated**: The Node has been shut down.

Default Nodes

A default instance type and boot image tags must be specified in server.yml. Pods with an empty image tag spec or instance type spec will use the defaults in server.yml.

```
apiVersion: v1
clusterName: your-cluster-name
nodes:
  defaultInstanceType: t3.small
```

A default volume size can be specified in server.yml. All booted Nodes will have the specified volume size in GB. The root volume size can be increased in the pod spec but cannot be decreased.

Standby Nodes

When one or more `standbyNodes` pools is specified in the nodes section server.yml, Milpa will keep a standby pool of running Nodes to ensure applications are dispatched to running Nodes quickly. The following server.yml will ensure that an additional 5 t3.small instances and 2 c5.large instances are kept in a standby pool for use by future Pods. It's possible to exhaust the standby pool when launching many Pods at once. Milpa continually checks the number of Nodes in the standby pools, and boots new Nodes if necessary.

```
apiVersion: v1
clusterName: your-cluster-name
nodes:
  standbyNodes:
    - instanceType: "t3.small"
      count: 5
```

```
      spot: false
    - instanceType: "c5.large"
      count: 2
      spot: false
```

Pods

As stated in the Overview, a Milpa Pod is a group of one or more applications that will run together on a Node.

Since Milpa does not colocate multiple Pods on a single Node, Milpa Pods have some unique attributes not found in other management controllers:

- Users must specify the type of cloud instance on which a Pod should run or what type of resources the Pod requires.
- Users can specify that a Milpa Pod should run on a cloud spot instance.

Pod lifecycle

Typically Pods have a linear lifecycle progressing from created to a terminal state.

- **Waiting**: The Pod has been created in Milpa and is waiting to begin running. It's likely the Pod is waiting for a Node to boot and become available.
- **Dispatching**: Milpa has matched an available Node to the creating Pod. Milpa is working to lauch the Pod on the Node.
- **Running**: The Pod has been successfully dispatched and is now starting or running the Pod Units.
- **Succeeded**: The Pod has finished running and all Units have returned successfully. This is a terminal state.
- **Failed**: The Pod has failed to run, either there was a problem dispatching the Pod or one or more Units failed. Depending on the pod's `restartPolicy`, this can be a terminal state.
- **Terminated**: The Pod has been stopped by request. This is a terminal state.

Pod Labels

Labels are used throughout Milpa to organize, query and select objects. When a Pod is dispatched to a cloud instance, the Pod labels are copied to the cloud instance as instance tags allowing the Node to be queried using the cloud console. Tags can be used to filter by application and environment in cloud billing reports.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: queuecheck
  labels:
    app: writer
    env: qa
```

Instance Type

Since each Milpa Pod runs on its own cloud instance, Pods can select the exact machine type they need by naming the cloud instance type in the `instanceType` field of the Pod spec. For AWS installations, example instance types would be `t3.nano`, `m4.16xlarge`, `p2.xlarge`, etc.

```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  instanceType: m4.16xlarge
  units:
    - name: nginx
      image: nginx
      command: ["/usr/sbin/nginx", "-g", 'daemon off;']
```

Resources

Instead of specifying an `instanceType` in the Pod spec, the user can specify a combination of CPU, Memory and GPU requirements and Milpa will choose the cheapest instance that satisfies those requirements. `cpu`, `memory`, `gpu` and `volumeSize` are string type resource quantities.

`cpu` is specified in units of cores, `gpu` is specified as the number of gpus. For CPUs, fractional requests are allowed. For example, setting cpu to `0.1` would select a T3.nano instance on AWS since T3.nano instances provide a baseline performance equivalent to 10% of a full CPU.

Memory and volume size are specified in units of bytes. As in kubernetes resource specifications, values can be integer numbers or can be specified with one of the following suffixes E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki.

Some cloud instance families have cheaper shared CPUs (e.g. T2/T3 instances) while most instance types feature dedicated CPUs. To prevent Milpa from selecting a cheaper T2/T3 instance with a shared CPU, set `dedicatedCPU` to true.

For AWS's T family of instances you can choose to set the instance's CPU into "Unlimimted" mode and use more than the baseline CPU. This is done by setting `resources.sustainedCPU` to `true`. In this mode, the CPU will not be limited to the baseline performance of the instance type, instead the user pays a flat fee of $0.05/hour per vCPU of usage. If you use less than 100% of a vCPU/hour you'll be charged for only the portion you use above the baseline performance. For more information see AWS documentation for T2/T3 Unlimimted.

In attempting to choose the cheapest instance for your specified workload, Milpa will consider using T instances with sustainedCPU enabled. To prevent Milpa from considering T instances in unlimited mode, set `resources.sustainedCPU: false`.

To start a node without a public IP, set `privateIPOnly` to true. To use a private Node, your cloud network must be configured to allow access to your Docker registry and access to S3, and Milpa must be able to reach the Node on its private IP address.

The following manifest will provision a c5.large instance with 60GiB disk volume for the Pod since that is the lowest cost instance type that satisfies the resource requirements.

```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
```

```
  resources:
    cpu: "2"
    memory: "2Gi"
    volumeSize: "60Gi"
    dedicatedCPU: true
    privateIPOnly: false
  units:
    - name: nginx
      image: nginx
      command: ["/usr/sbin/nginx", "-g", "daemon off;"]
```

Root Volume Size

The `volumeSize` parameter of the ResourceSpec is used to specify the size (in bytes) of the root volume for the Pod. Milpa Nodes can be run with a root volume as small as 1GiB. To support larger applications out of the box, the `defaulVolumeSize` been set to 5GiB in the copy of server.yml distributed with Milpa. To change that value, update server.yml and restart milpa. If your Pod requires more disk space than the default, you will need to specify a `volumeSize` in the resources section of the Pod spec. The root volume size will be grown when a Pod is dispatched to the Node. While the root volume size can be increased from the default size, it cannot be decreased.

Spot Instance Support

Milpa allows users to run their Pods on spot instances in order to save money on their cloud computing bills. The `spec.spot.policy` attribute of the Pod determines if the Pod will be run on a spot instance. There are 3 possible values for a Pod's spot policy: `Always`, `Preferred` or `Never`. The default value is `Never` since spot instances can be terminated at any time (with a 2 minute warning on AWS) and can lead to data loss or unavailable systems. The spot policy `Always` will always to try and run the pod on a spot instance, even if no spot instances are available. When a spot instance is not available, the Pod will remain in the Waiting phase waiting for a spot instance. Finally, `Preferred` will attempt to run the pod on a spot instance but if no spot instances are available the Pod will be run on an OnDemand instance and be charged at the OnDemand price.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: nginxpod
spec:
  instanceType: m3.medium
  spot:
    policy: always
  units:
    - name: nginx
      image: nginx
      command: ["/usr/sbin/nginx"]
```

Note: Spot instance termination notification is on the roadmap for a future release of Milpa. In the meantime, applications can query the EC2 metadata service to be notified of pending spot instance termination.

Placement

A user can select the availability zone for a pod by specifying the string value of the availability zone in `spec.placement.AvailabilityZone` field.

```
spec:
  placement:
    availabilityZone: "us-east-1a"
```

If an Availability Zone is not specified, Milpa will attempt to deduce public and private subnets in your infrastructure. If there are subnets in the VPC without an Internet Gateway attached, those subnets will be used to launch Pods that only have a private IP address (`spec.resources.privateIPOnly: true`) while Pods with a public IP address (`spec.resources.privateIPOnly: false`) will be launched in subnets that have an Internet Gateway attached. If all subnets in the VPC have an Internet Gateway attached, a pod will be deployed to any suitable subnet irrespective of whether the Pod requires a public IP address.

Units

Since Milpa doesn't run a container runtime on the Nodes, individual applications are specified with a `Unit` declaration. Each Unit must include a `name` that is unique for the Pod, an `image` specification and a `command` to run the application in the Unit.

Unit Images

Milpa can pull images from DockerHub, AWS ECR and private registries.

Pulling Images from Docker Hub

The usual Docker conventions are used to specify an image hosted on Docker Hub, see the Docker documentation for a detailed explanation on how to specify an image.

Example image value for an image hosted on Docker Hub: `library/python:3.6-alpine`.

Pulling Images from AWS ECR

If using AWS ECR, Milpa must have AMI permissions to pull images and must be able to generate an ECR authorization token for the repository. To pull an image from ECR, simply use the full ECR image name (e.g. `ACCOUNT.dkr.ecr.REGION.amazonaws.com/imagename:tag`) in the Unit's image field.

Pulling Images from a Private Repo

To specify a private registry, simply prepend the namespace with the url and port of the private registry. For example, `myregistry.local:5000/testing/test-image` will pull `testing/test-image` with the `latest` tag from the registry at `myregistry.local:5000`.

Milpa supports `imagePullSecrets` for handling passwords for private repositories. See the Image Pull Secrets section for more details.

Unit Environment variables

Environment variables can be specified for each Unit by including a list of names and values in the Unit spec. See the Secrets as Environment Variables section for more information on how to use Secrets in environment variables.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: helloserver
spec:
  units:
    - name: helloserver
      image: examplecorp/helloserver:latest
      command: ["/helloserver"]
      env:
        - name: MY_MILPA_VAR
          value: my_env_var_value
```

Unit Ports

When `firewallMode` is set to `Closed` in server.yml, specifying service ports in the Unit specification will attach cloud firewall rules to the Pod and open the specified ports to the VPC. For more flexibility including opening ports to the public internet, specify ports using a `Service` resource.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: helloserver
spec:
  units:
    - name: helloserver
      image: examplecorp/helloserver:latest
      command: ["/helloserver"]
      ports:
        - name: hello
          protocol: TCP
          port: 8002
```

Init Units

Similarly to Kubernetes init containers, Milpa also supports init Units.

Init Units are exactly like regular Units, except:

- They always run to completion.
- Each one must complete successfully before the next one is started.

Other than that, you can specify an init Unit the same way as a regular Unit (using the same fields such as `name:`, `image:`, `command:`, etc).

If an init Unit fails, Milpa will restart the Pod according to its `restartPolicy`:

- With a `restartPolicy` of `always` or `onFailure`, the Pod will be restarted until the init Unit succeeds.
- If the Pod has a `restartPolicy` of `never`, it is not restarted, and the Pod will transition to a `Failed` state.

To specify a Unit as an init Unit, add the `initUnits` field to the Pod `.spec` as an array of `Unit` objects. The status of the init Units is returned in `.status.initUnitStatuses` field as an array of the Unit statuses (similar to the `.status.unitStatuses` field).

Order matters for init Units. They will be started in the order specified, one at a time.

Even when `restartPolicy` for the Pod is `always`, init Units will only be restarted when they fail, i.e. exit with a non-zero exit code (otherwise your Pod would be stuck with the first init Unit getting restarted).

Example Pod and Service manifest with init Units:

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
---
apiVersion: v1
kind: Service
metadata:
  name: mydb
spec:
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
---
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  units:
  - name: myapp-unit
    image: alpine
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

```
  initUnits:
  - name: init-myservice
    image: alpine
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: alpine
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

To run the example, please make sure that DNS service discovery is enabled in your server configuration:

```
serviceDiscovery:
  privateDNS:
    ttl: 30
```

The Services from the manifest will be then registered in DNS. The two init Units will succeed once DNS can resolve the registered service names in DNS, and only then `myapp-unit` will start running.

EmptyDir

An EmptyDir is a volume that can be shared between separate Units in the same Pod. All Units that mount the volume can read and write to the emptyDir directory.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: alpine
  labels:
    name: alpine
    app: alpine
spec:
  units:
  - name: "sleep"
    image: library/alpine:3.6
    command: ["sleep 300"]
    volumeMounts:
    - mountPath: /cache
      name: shared-volume
  volumes:
  - name: shared-volume
    emptyDir: {}
```

EmptyDir can be backed by either available disk space on the Node, or by tmpfs. The default is to use disk space.

To create a tmpfs-backed emptyDir volume:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: alpine
```

```
  labels:
    name: alpine
    app: alpine
spec:
  units:
  - name: "sleep"
    image: library/alpine:3.6
    command: ["sleep 300"]
    volumeMounts:
    - mountPath: /cache
      name: shared-volume
  volumes:
  - name: shared-volume
    emptyDir:
      medium: Memory
      sizeLimit: 128
```

The parameter `sizeLimit` specifies in MB the size of the tmpfs volume Milpa will create for the Pod.

Milpactl Volume

A powerful feature of Milpa is the ability to run pods on Milpa that can query and control Milpa. To make accessing the api easy from pods, there's a quick shortcut to put `milpactl` and API client certificates on a pod. To add `milpactl` to a pod, you need to provide an annotation that specifies the name of the volume that will be populated with `milpactl` and the certificates. The pod's spec must also specify the volume and volume mounts. The following example shows an annotation specifying that 'my-milpactl-vol' volume will be populated with `milpactl` and client certificates will be placed into a directory at `/milpactldir` in the helloserver unit.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: helloserver
  annotations:
    pod.elotl.co/milpactl-volume-name: "my-milpactl-vol"
spec:
  units:
    - name: helloserver
      image: elotl/helloserver
      command: ["/helloserver"]
      volumeMounts:
        - name: milpactl-vol
          mountPath: /milpactldir
  volumes:
    - name: my-milpactl-vol
      packagePath:
        path: milpactl
```

As an example, with the above configuration, the `helloserver` unit could list all pods in the cluster by executing the command `/milpactldir/milpactl get pods`

Updating Pods

Pods can be updated by Milpactl. Units can be removed and added and Unit images can be changed. The underlying cloud resources (instanceType and volume size) cannot be changed.

```
$ milpactl update -f ./path/to/pod_manifest.yml
```

Deleting Pods

Deleting a Pod will delete the Pod and terminate the Node the Pod was running on.

```
$ milpactl delete pod <podname>
```

Services

Milpa Services allow users to select Pods and apply networking policies to those Pods. In Milpa, there are three network functions that can be applied via Services:

1. Services can be used to apply cloud firewall rules to Pods matching the service selector.
2. If a service discovery system is configured in server.yml (e.g. DNS Service Discovery), a Service will register matching Pods in the service discovery backend.
3. A cloud load balancer can be created to route traffic to Pods matching the service selector.

The following Service will apply to any Pods with the a label of `app: echoer`. It will create a cloud firewall policy with one rule that opens port 8002 to a network on the public internet. The firewall policy will be attached to any Pods matching the Service label selector. The Service will also register matching Pods in an AWS Private DNS zone.

```
---
kind: service
apiVersion: v1
metadata:
  name: echo-svc
  labels:
    app: echoserver
spec:
  selector:
    matchLabels:
      app: echoer
  ports:
    - name: echo
      protocol: tcp
      port: 8002
  sourceRanges:
    - "128.32.0.0/16"
```

Cloud Firewall Rules

As described in Server Configuration there are 2 firewall modes for a Milpa cluster: `OpenToVPC` and `Closed`.

Firewall Mode - OpenToVPC

When firewallMode is set to `OpenToVPC`, all Milpa Nodes receive a security group that opens all ports on the Node to the VPC. The controller will not create a security group for any Service with all source ranges inside the VPC since those ports are already open. Any connection to the Nodes from outside the VPC will still be dropped. This mode sacrifices a layer of security for ease of ues and supplies a way to get around limits of firewall implementations in some cloud networks.

Firewall Mode - Closed

When firewallMode is set to `Closed`, it's necessary to create a Service to open ports for a Pod. Every Service with one or more CIDR blocks in the sourceRange list of the service spec will create a

corresponding cloud firewall rule (in EC2 language, this is a Security Group) with inbound rules specified in the Service manifest. Inbound traffic that does not match `ports` and `sourceRange` specified in the Service will be dropped. Thus, one or more Services must be applied to a Pod in order to allow ingress traffic to that Pod. For ease of use, you can specify `VPC` in the sourceRange list instead of the VPC's CIDR to allow connections from the local VPC.

In a future version of Milpa, we hope to extend `sourceRanges` to allow the user to specify other Services as the origin of allowed traffic.

Pod Ports

As discussed in Pod Ports, cloud firewall rules can also be applied in the Pod spec. Ports specified in the Pod spec are only open to the VPC's subnet.

```
apiVersion: v1
kind: Pod
metadata:
  name: helloserver
spec:
  units:
    - name: helloserver
      image: elotl/helloserver
      command: ["/helloserver"]
      ports:
        - name: hello
          protocol: TCP
          port: 8002
```

Service Discovery

Service discovery is a key component of a dynamic infrastructure. Currently Milpa supports service discovery in AWS via private DNS zones provided by Route53 but other service discovery systems could be added in future releases. Also, if you don't want to use Milpa's service discovery, it's easy to use your own service discovery system with Milpa. Simply include the service discovery container (e.g. `image: library/Consul`) as a Unit in your Pod spec.

Enabling Service Discovery

To enable DNS service discovery ensure the following section is present and uncommented in `/etc/milpa/server.yml`:

```
serviceDiscovery:
  privateDNS:
    ttl: 30
```

If the record TTL is disabled it defaults to 30 seconds.

Private DNS Service Discovery

When Private DNS service discovery is enabled, Milpa will create a private zone in route53 named `clusterName.local` (where `clusterName` is the name of the cluster set in server.yml). Each Service will create an A record containing the IP address of every Pod that the service's selector matches. The A record is named:

`<servicename>.default.<clusterName>.local`

If the Service doesn't match any Pods, the A record will not be created. Milpa will also create an SRV record for each port listed in the Service manifest spec. Each SRV record has the form:

`_<portname>._<protocol>.<servicename>.default.<clusterName>.local`

For the example service manifest at the start of this section in the `example` cluster, the A record would be named `helloworld-svc.default.example.local` and the SRV record would be named: `_hello._tcp.helloworld-svc.default.example.local`. The contents of the SRV record are a space separated list of priority (always 1), weight (always 1), the DNS name of the instance and the port the Service is available at. If the Service matches multiple Pods, multiple records will be created in the SRV record. For the example Service, the contents of the SRV record would be: `1 1 <ec2 instance DNS name> 8002`.

DNS Resolution on Pods

Milpa configures all Pods to use the local VPC resolver and sets the default resolver search path to be `default.<clusterName>.local`. This allows the user to specify the name of the Service instead of the entire domain name. For example, the following command would successfully curl port 8080 on the "web" service from within a Pod in the same cluster:

```
curl http://web:8080
```

Service Selectors

Milpa uses Kubernetes labels and selectors to select Pods controlled by a Service. Unlike Kubernetes Services, Milpa Services support set-based match requirements in addition to label selectors. Check the Kubernetes documentation for more information about Kubernetes selectors.

```
spec:
  selector:
    matchLabels:
      app: myapplabel
      env: development
```

Load Balancer Services

If a Service is marked as `spec.type: LoadBalancer` a cloud load balancer will be provisioned for the Service. For AWS, only Classic Elastic Load Balancers are supported at this time. A load balancer is created when the load balancer Service is created and any Pods that match the service's selector will be added and removed from the load balancer as they are created and destroyed.

Load Balancer Configuration

To create a load balancer, the Service's type must be set to `LoadBalancer` and a set of front-end and Node ports must be specified in the ports section of the Service spec. Each port in the specification must supply both a port number in `port` and in `nodePort`. The `nodePort` is the target port on the Pod/Node. The `sourceRange` specifies the networks the load balancer will accept connections from. As a simple example, the following is a load balancer that listens on port 80 and forwards connections to port 8080 on matching Pods.

```
---
kind: Service
apiVersion: v1
metadata:
  name: nginx-svc
spec:
  type: LoadBalancer
  selector:
    matchLabels:
      app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 80
      nodePort: 8080
  sourceRanges:
    - 0.0.0.0/0
```

Amazon differentiates between internet-facing load balancers and internal load balancers. If sourceRanges is unspecified or only lists networks within the VPC (or lists "VPC" as the sourceRange) an internal load balancer will be created. Otherwise an internet-facing load balancer will be created. Changing the source ranges from only internal networks to containing external networks (and vice versa) will cause the load balancer to be removed and re-created.

An Amazon ELB needs to be connected to backing subnets. Milpa will connect load balancers to subnets which match the mode of the load balancer. Internet facing load balancers will be attached to public subnets (subnets that have a route to an internet gateway) while internal subnets will prefer subnets that lack an internet gateway.

Load Balancer Status

When a load balancer is created, the load balancer address will be stored in the service's `status.loadBalancer` field. AWS load balancers do not have an IP address but instead do have a DNS name. The load balancer is also registered in service discovery like other Services.

Load Balancer Annotations

Cloud load balancers offer a variety of configuration options depending on the cloud provider. Annotations are used to configure these configuration options. Below are annotations for most common usecases.

AWS Access Logs Annotations

```
metadata:
  annotations:
    # enable or disable log delivery
    service.elotl.co/aws-load-balancer-access-log-enabled: "true"
    # Minutes between delivery of access logs. Must be 5 or 60 minutes.
    service.elotl.co/aws-load-balancer-access-log-emit-interval: "5"
    # S3 bucket that the logs will be delivered to. The bucket must be
    # located in the same region as the load balancer.
    service.elotl.co/aws-load-balancer-access-log-s3-bucket-name: "cluster-load-balancer-logs"
    # The prefix that will be appended to the standard log path inside the bucket
    service.elotl.co/aws-load-balancer-access-log-s3-bucket-prefix: "lblogs/web"
```

AWS Connection Draining Annotations

```
metadata:
  annotations:
    # Enable connection draining
    service.elotl.co/aws-load-balancer-connection-draining-enabled: "true"
    # The amount of time to keep existing connections open before
    # removing them from an unhealthy or removed backend instance.
    service.elotl.co/aws-load-balancer-connection-draining-timeout: "30"
```

AWS Idle Timeout Annotations

```
metadata:
  annotations:
    # the number of seconds a connection can be idle before the load balancer
    # closes the connection.
    service.elotl.co/aws-load-balancer-connection-idle-timeout: "45"
```

AWS Cross Zone Load Balancing Annotation

```
metadata:
  annotations:
    # turn on cross-zone load balancing
    service.elotl.co/aws-load-balancer-cross-zone-load-balancing-enabled: "true"
```

AWS Health Check Annotations

```
metadata:
  annotations:
    # Time between health checks. Must be between 5 and 300
    service.elotl.co/aws-load-balancer-healthcheck-interval: "5"
```

```
    # Time to wait for a healthcheck to complete. Must be between 2 and 60.
    service.elotl.co/aws-load-balancer-healthcheck-timeout: "2"
    # The number of successive health check failures required for a backend to
    # be considered unhealthy. Must be between 2 and 10
    service.elotl.co/aws-load-balancer-healthcheck-unhealthy-threshold: "2"
    # The number of successive successful health checks required for a backend to
    # be considered healthy. Must be between 2 and 10
    service.elotl.co/aws-load-balancer-healthcheck-healthy-threshold: "2"
```

AWS SSL Certificate Annotations

```
metadata:
  annotations:
    # which ports will serve https traffic (defaults to * if unspecified
    # and an aws-load-balancer-ssl-cert is specified)
    service.elotl.co/aws-load-balancer-ssl-ports: "443"
    # specifies the protocol of the backend service/pod behind the load balancer
    # If "http" (default) or "https", an HTTPS listener that terminates the
    # connection and parses headers is created.
    # If set to `ssl` or `tcp`, a "raw" SSL listener is used.
    service.elotl.co/aws-load-balancer-backend-protocol: "http"
    # The annotation used on the service to request a secure listener.
    # This value must be a valid certificate ARN.
    # For more information, see http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuid
    service.elotl.co/aws-load-balancer-ssl-cert: "arn:aws:acm:us-east-1:123456789012:certificate/123
```

Updating Services

It's possible to update a Service spec. Milpa will make the necessary changes to the Service and matched Pods.

```
$ milpactl update -f /path/to/service-manifest.yml
```

Deleting Services

When a Service is deleted, all associated cloud resources will be updated to reflect the deletion.

```
$ milpactl delete service <servicename>
```

ReplicaSets

ReplicaSets are used to run multiple replicas of a Pod. Typically a user would not want to run a ReplicaSet on their own, instead they should use a Deployment resource. Deployments use ReplicaSets under the hood to control and manage Pods. When managing ReplicaSets, Milpa will attempt to balance Pods across availability zones. If an availability zone goes offline or no longer has capacity or availability, Milpa will attempt to balance created Pods across the remaining healthy availability zones. When an availability zone comes back online Milpa will not automatically re-balance Pods back into the recovered availability zone. That said, any new Pods spawned by the ReplicaSet will attempt to fill the recovered availability zone.

ReplicaSet Example

```
apiVersion: v1
kind: ReplicaSet
metadata:
  name: helloworldreplica
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      instanceType: t3.nano
      units:
        - name: helloserver
          image: elotl/helloserver
          command: ["/helloserver"]
```

Milpa ReplicaSets are different from Kubernetes replica sets in a few ways:

1. Milpa ReplicaSets don't support selectors. Instead they create internal labels to manage the Pods they create.
2. At this time, Milpa ReplicaSets don't support the ability to set `minReadySeconds`. When a Pod is in the `Running` phase and is available, it is marked as an available replica and included in the count of `availableReplicas`.

Pod availability in a ReplicaSet is defined differently depending on the pod's `restartPolicy`:

- For pods marked as `restartPolicy` of Always, an available pod is defined as: all `initUnits` have successfully run to completion and all `pod.Status.unitStatuses` are in the running state.
- An available pod with `restartPolicy` of OnFailure is defined as all `initUnits` have successfully run to completion and all `pod.Status.unitStatuses` are in the running state or have terminated successfully.
- An available pod with `restartPolicy` of Never is defined as all `initUnits` have successfully run to completion and all `pod.Status.unitStatuses` are in the running or terminated state.

Updating a ReplicaSet

The ReplicaSet manifest can be updated to scale the number of replicas up and down. The template can be updated but existing replicas will not be modified, only new replicas will reflect the updated Pod template spec.

```
$ milpactl update -f ~/myreplica.yml
```

Deleting a ReplicaSet

By default, deleting a ReplicaSet will delete the ReplicaSet and all Pods the ReplicaSet has created. To delete the ReplicaSet and keep the Pods it manages, set `--cascade=false` in the delete command

```
$ milpactl delete --cascade=false replicaset <replicasetname>
```

Deployments

A Milpa Deployment is used to manage ReplicaSets and Pods. Deployments create ReplicaSets which in turn create Pods. Whenever a Deployment's template changes, a new set of replicas are rolled out. Pods from the old ReplicaSet are terminated while the new ReplicaSet is incrementally scaled up to match the Deployment spec.

It's possible to pause deployments by setting `spec.paused` equal to true and updating the manifest with `milpactl`. When paused, the Deployment will not progress further. ReplicaSets will continue to attempt to run the number of replicas specified in their spec but the Deployment will not continue to roll out new replicas or scale down old replicas.

Deployment Example

```
apiVersion: v1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 2
  maxSurge: 1
  maxUnavailable: 0
  template:
    metadata:
      labels:
        app: helloserver
    spec:
      instanceType: t3.nano
      units:
        - name: helloserver
          image: examplecorp/helloserver:latest
          command: ["/helloserver"]
          env:
            - name: MY_MILPA_VAR
              value: variable_value_1
```

When created, the Deployment will create a ReplicaSet with labels that it will use to track the ReplicaSet. If the Deployment is updated, it will create a new ReplicacSet that will run an increasing number of replicas while the old ReplicaSet is scaled down to zero replicas and eventually deleted. As mentioned in the ReplicaSets section, the ReplicaSets will attempt to balance Pods across availability zones in the VPC when they are scaled up and down.

The `maxSurge` and `maxUnavailable` parameters are used to control the number of extra Pods that are created as well as the maximum number of unavailable Pods. These are integer numbers representing the number of Pods that can be scheduled above and below the speced number of replicas in the Deployment. Specifying these values as a percentage of replicas is currently not supported in Milpa.

If a Deployment is updated frequently, it's possible to have multiple old ReplicaSets in flight at the same time. Milpa will work to shutdown the old replicas while increasing the number of replicas from the latest ReplicaSet.

Updating Deployments

To roll out a new Deployment, simply update the deployment spec (e.g. by changing the version of the image you'd like to deploy) and use Milpactl to update the Deployment manifest:

```
$ milpactl update -f ./hellodeployment.yml
```

Deleting a Deployment

By default, deleting a Deployment will delete the Deployment and all ReplicaSets and Pods the Deployment has created. To delete the Deployment and keep other objects, set `--cascade=false` in the delete command.

```
$ milpactl delete --cascade=false deployment <deployment name>
```

Deployment Caveats

- Milpa Deployments don't support user-created selectors. Deployments only manage ReplicaSets they have created.
- Currently there is no command line option to pause or roll back Deployments. To roll back a Deployment, you must update the deployment spec in the manifest to match the previous version of the Deployment.

Secrets

Secrets are used to store sensitive information that shouldn't be stored in a Pod manifest or container image.

Creating secrets

Secrets are created via a secrets manifest. At this time, they cannot be created via the command line. When creating a Secret, the secret must be base64 encoded and then added to the manifest. First encode the secret:

```
$ echo -n some-username | base64
c29tZS11c2VybmFtZQ==
$ echo -n super-secret-password | base64
c3VwZXItc2VjcmV0LXBhc3N3b3Jk
```

Add the Secret to a Secret manifest:

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  milpa-username: c29tZS11c2VybmFtZQ==          # some-username
  milpa-password: c3VwZXItc2VjcmV0LXBhc3N3b3Jk  # super-secret-password
```

Use Milpactl to create the Secret in Milpa.

```
$ milpactl create -f /path/to/secret.yml
```

Secrets as Environment Variables

Secrets can be referenced in a Pod spec and used as environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  units:
    - name: testunit
      image: elotl/example
      command: ["/run-example"]
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: milpa-username
        - name: SECRET_PASSWORD
          valueFrom:
```

```
            secretKeyRef:
              name: test-secret
              key: milpa-password
```

Milpa does not currently support mounting secrets as a volume.

Image Pull Secrets

To create a Secret that will be used as an `imagePullSecret` for accessing private Docker registries, first create a Secret with values for `username`, `password` and `server` (the url of the Docker Registry). In the Pod spec, reference the Secret name in `imagePullSecrets`. Since multiple secrets can be mentioned in the Pod spec (for pulling Units from multiple private registries), the host portion of the image name in the Unit spec must match the `server` value in the Secret.

```
---
apiVersion: v1
kind: Secret
metadata:
  name: private-creds
data:
  server: cHJpdmF0ZXJlZy5leGFtcGxlLmNvbQ==  # privatereg.example.com
  username: ZWxvdGw=                        # elotl
  password: cGFzc3dvcmQ=                    # password
---
apiVersion: v1
kind: Pod
metadata:
  name: helloserver
  labels:
    app: helloserver
spec:
  instanceType: t3.nano
  units:
    - name: mycontainer
      image: privatereg.example.com/user/mycontainer
      command: "[/mycontainer"]
  imagePullSecrets:
    - private-creds
```

Caveats

- Secrets must be created before any Pod that depends on the Secret runs. If Secret values are not available to a Pod, the Pod will be marked as failed.
- Values of Secrets will not be printed by Milpactl. Instead, Milpactl will print the length of the Secret in bytes. That length will be base64 encoded.
- Like in Kubernetes, the API server stores secret data as plaintext in the backing storage DB. This means the same precautions and risks for Kubernetes secrets apply to Milpa secrets: https://kubernetes.io/docs/concepts/configuration/secret/#risks

Logs

It's possible to see the stdout and stderr output of a Unit via Milpactl's `logs` command. To aid in troubleshooting, Milpactl can also access the Node agent's logs that detail operations taken to initialize and start Units on the Node. By default, Milpa will display only the last 10 log lines. To see more logs, specify the number of lines to view with the `--lines` argument.

View Pod Logs

```
# View the last 45 lines from the http Unit in the nginx Pod
$ milpactl logs --lines=45 nginx -u http

# access the last 500 byes of the agent log on the Node <UUID>
$ milpactl logs --limit-bytes=500 f8656b8d-c695-40a0-9803-d99978e58eed

# follow the logs of the http Unit in the nginx Pod (similar to tail -f)
$ milpactl logs -f nginx -u http

# if a Pod only has one Unit, the Unit name can be omitted
$ milpactl logs -f nginx
```

Old Logs

To help debug failures, Milpa saves the last 4K of logs to the internal key value store and keeps that information available for 1 hour. Logs are accessed in the same manner as logs for running Pods. If a Pod or Node with the same name exists and is currently running, the old logs will not be available for query.

```
# access old Pod logs for a Unit
$ milpactl logs <podname> -u <unitname>

# access old Node agent logs
$ milpactl logs f8656b8d-c695-40a0-9803-d99978e58eed
```

Events

Events are generated by Milpa to report various events that happened during operation. They are useful to see what has been happening in a cluster and the state transitions of objects in Milpa.

Checking events

Use Milpactl to pull the list of Events:

```
$ milpactl get events
alpine                                  Pod       alpine                                  pod-created
alpine                                  Pod       alpine                                  pod-updated
alpine                                  Pod       alpine                                  pod-updated
alpine                                  Pod       alpine                                  pod-updated
cc39f298-0cd2-4c4e-a55b-3f2a9d82de65    Node      cc39f298-0cd2-4c4e-a55b-3f2a9d82de65    node-cleanin
alpine                                  Pod       alpine                                  pod-deleted
$
```

Here, a Pod named `alpine` is created, it is scheduled to run on a Node, successfully finishes and then Milpa removes it and cleans up its Node.

Fault Tolerance

Milpa Leader-Follower Setup

Multiple Milpa servers can be run in a cluster for a fault tolerant setup. Typical fault tolerant setups consist of two Milpa servers, one leader and one follower but more than one follower can exist in the cluster.

```
clusterName: your-cluster-name
etcd:
  client:
    endpoints:
      - https://localhost:2379
      - https://172.20.5.12:2379
```

When running in a multi-server cluster, all Milpa servers must share the same `clusterName` and be configured to use an external etcd cluster for storage. The external etcd cluster should be built to tolerate faults (3 or 5 nodes). When starting up, one Milpa server will be elected the leader of the cluster and the other servers in the cluster will act as read-only followers. If the leader fails and is unable to talk to etcd (or is shut down), one of the follower servers will be elected the master of the cluster.

In a leader follower setup, only the leader can accept write operations for the cluster. However, all servers can serve read operations. The Milpa client can be configured to talk to all servers in a cluster using the `--endpoints` flag. Writes will be redirected to the master while reads will be load balanced across nodes.

Supporting Multiple Availability Zones

Milpa keeps track of subnets, availability zones and subnet availability and uses that information to spread Deployments across availability zones. If Milpa detects a zone is unavailable due to capacity or other errors, attempts are made to distribute Pods evenly across the remaining availability zones.

IAM Policy

The following policy covers all permissions Milpa requires in order to run in AWS.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ec2",
            "Effect": "Allow",
            "Action": [
                "ec2:AuthorizeSecurityGroupIngress",
                "ec2:CreateRoute",
                "ec2:CreateSecurityGroup",
                "ec2:CreateTags",
                "ec2:CreateVolume",
                "ec2:DeleteRoute",
                "ec2:DeleteSecurityGroup",
                "ec2:DescribeAddresses",
                "ec2:DescribeElasticGpus",
                "ec2:DescribeImages",
                "ec2:DescribeInstances",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSpotPriceHistory",
                "ec2:DescribeSubnets",
                "ec2:DescribeTags",
                "ec2:DescribeVolumes",
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeVpcs",
                "ec2:ModifyInstanceAttribute",
                "ec2:ModifyInstanceCreditSpecification",
                "ec2:ModifyVolume",
                "ec2:ModifyVpcAttribute",
                "ec2:RequestSpotInstances",
                "ec2:RevokeSecurityGroupIngress",
                "ec2:RunInstances",
                "ec2:TerminateInstances",
                "ecr:BatchGetImage",
                "ecr:GetAuthorizationToken",
                "ecr:GetDownloadUrlForLayer",
                "elasticloadbalancing:DescribeLoadBalancerAttributes",
                "elasticloadbalancing:DescribeLoadBalancers",
                "elasticloadbalancing:DescribeTags",
                "route53:ChangeResourceRecordSets",
                "route53:CreateHostedZone",
                "route53:GetChange",
                "route53:ListHostedZonesByName",
                "route53:ListResourceRecordSets"
            ],
            "Resource": "*"
```

```
        },
        {
            "Sid": "dynamo",
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:PutItem",
                "dynamodb:DescribeTable",
                "dynamodb:GetItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/MilpaClusters"
        },
        {
            "Sid": "elb",
            "Effect": "Allow",
            "Action": [
                "elasticloadbalancing:DeleteLoadBalancer",
                "elasticloadbalancing:RemoveTags",
                "elasticloadbalancing:CreateLoadBalancer",
                "elasticloadbalancing:ConfigureHealthCheck",
                "elasticloadbalancing:AddTags",
                "elasticloadbalancing:ApplySecurityGroupsToLoadBalancer",
                "elasticloadbalancing:DeleteLoadBalancerListeners",
                "elasticloadbalancing:DeregisterInstancesFromLoadBalancer",
                "elasticloadbalancing:RegisterInstancesWithLoadBalancer",
                "elasticloadbalancing:ModifyLoadBalancerAttributes",
                "elasticloadbalancing:CreateLoadBalancerListeners"
            ],
            "Resource": "arn:aws:elasticloadbalancing:*:*:loadbalancer/milpa-*"
        }
    ]
}
```

Troubleshooting

Below is a brief list of steps commonly used to debug Pods that aren't running or are running incorrectly.

Pod Status

The first step in debugging a Pod is to look at the full representation of the Pod resource. To see the full representation of a Pod as yaml (json output is also available):

```
milpactl get pod <podname> -oyaml
```

In the output look at the state of the Pod and the Unit statuses (`at pod.status.unitStatus`). Also make sure the Units look correct and expected values are present. Fields set in the Pod manifest should be present in the Pod spec.

Milpa Logs

After looking at the Pod, the logs generated by the Milpa process can be used to find problems in the cluster. If Milpa is running via systemd, the logs can be followed with journalctl.

```
$ journalctl -fu milpa
```

In the logs, look for errors and other problems. Some warnings are expected (e.g. a Node refusing healthchecks when it's still booting) but others can point to a problem in the system. Milpa can recover from common problems like a Node going down or a subnet becoming unavailable (if there are other subnets available in your VPC) but other unforeseen circumstances (e.g. cloud errors and communications issues) will likely be shown in the logs.

Node and Pod Logs

If there's a problem starting a Unit or the Unit isn't behaving correctly, it can be useful to look at the Pod and Node logs. To do this, use the `milpactl logs` command. As explained in the Logs section, Milpa allows you to see current output from a Unit and saves the tail of a Unit's log files after the Unit has stopped. To access the logs of a Pod's unit:

```
$ milpactl logs <podname> -u <unitname>
```

Milpa also keeps Node logs that report information about the state of the Node and its attempts to run the Units. To see the Node's logs, find the name of the Node that the Unit is running on (Nodes names are UUIDs) using `milpactl get pods`. If the Pod has already stopped, the Event stream will show the name of Node the Pod was run on. That information is stored in pod-running events:

```
milpactl get events
mypod         Pod               mypod         pod-running             pod-controller\
mypod running on node b3a10289-e895-4535-8366-18e3bf3131b8
```

Using that information, the logs of the Node can be queried using Milpactl:

```
$ milpactl logs b3a10289-e895-4535-8366-18e3bf3131b8
```

Node logs can be useful to investigate communications problems with Docker registries and out of memory issues.

Exec

It can be useful to execute commands in the context of a running Unit and poke around within the Unit's namespace. To do this, you'll likely want to have a container image with a shell built into it. Alpine images typically use `ash` while other images often have `bash` built into them. To launch a shell in the context of a Unit:

```
# To run this command, make sure your image has /bin/bash built into it
$ milpactl exec -ti <podname> -u <unitname> -- /bin/bash
```

After running the above command, you can look at and modify a running Unit and work to figure out what's going on. It's possible that a configuration file isn't where it should be, ports are closed or the Unit's `command` arguments are incorrect.

Networking Troubles

If the Node `firewallMode` is set to `Closed` in server.yml, make sure that you've declared a Service for each port you want to open to your VPC. Also your Service must declare a `sourceRange` in the spec. Without a `sourceRange`, when the firewallMode is closed, the cloud firewall will not be opened for the specified Service port.

Setting Machines up for SSH Access

As a last resort, it might be necessary to enable ssh access to a Node in order to debug the Pod. The easiest way to do this is to configure a cloud-init file that authorizes a user to log into a Node. See the cloud-init section for the details on using cloud-init. Briefly, to enable ssh access to a node, `server.yml` will need to contain a pointer to a cloud-init file and that file should contain commands to either add a user and their ssh key or add an ssh key for the root user.

```
# snippet of /opt/milpa/etc/server.yml
nodes:
  cloudInitFile: /opt/milpa/etc/cloudinit.yml
```

```
# /opt/milpa/etc/cloudinit.yml contents
# ssh_authorized_keys will be added to the root user
ssh_authorized_keys:
  - "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC0g+ZTxC7weoIJLUafOgrm+h..."
```

Installing additional software on the Node

Sometimes it is necessary to dig deep and start using more advanced tools to poke around your infrastructure and Milpa Nodes. Milpa Nodes run a customized version of Alpine. Once you're setup with ssh access, it's possible to add additional software to the Node:

```
# steps to install and use curl on the Node
$ apk update
```

```
$ apk add curl bind-tools
$ curl mysvc:80
```

Support bundle

If rootcausing your failure leads to dead end and you need to contact Elotl Support, please include following payload while reaching out to Elotl.

- Sanitized `/opt/milpa/etc/server.yml` after removing your AWS keys.
- Milpa logs gathered via `journalctl -u milpa -l --no-pager`.
- Milpa dump gathered via `milpactl dump all > milpaDump.txt`.

Milpa API types

This section specifies the types used in the Milpa server API.

ObjectMeta

ObjectMeta is metadata that is maintained for all persisted resources, which includes all objects users create. This is added and kept up to date by Milpa.

name *string* Name of the resource.

labels *map[string]string* A dictionary of labels applied to this resource..

creationTimestamp *Time* Time of creation. Optional.

deletionTimestamp *Time* Time when the resource got deleted. Optional.

annotations *map[string]string* Unused. Optional.

uid *string* Universal identifier in order to distinguish between different objects that are named the same in differing timespans. E.g. if a user creates a Pod named foo, then deletes and recreates the Pod, we need a way to tell those two Pods apart. Optional.

namespace *string* Namespace placeholder. Currently Milpa does not support multiple namespaces so this will always be set to "default". Optional.

Pod

Pod is a collection of Units that run on the same Node.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Object metadata.

spec *PodSpec* Spec is the desired behavior of the pod. Optional.

status *PodStatus* Status is the observed status of the Pod. It is kept up to date by Milpa. Optional.

PodSpec

phase *PodPhase* Desired condition of the Pod.

restartPolicy *RestartPolicy* Restart policy for all Units in this Pod. It can be "always", "onFailure" or "never". Default is "always". The restartPolicy applies to all Units in the Pod. Exited Units are restarted with an exponential back-off delay (10s, 20s, 40s ...) capped at five minutes, the delay is reset after 10 minutes.

units *[]Unit* List of Units that together compose this Pod.

initUnits *[]Unit* Init Units. They are run in order, one at a time before regular Units are started.

imagePullSecrets *[]string* List of Secrets that will be used for authenticating when pulling images.

instanceType *string* Type of cloud instance type that will be used to run this Pod. Optional.

spot *PodSpot* PodSpot is the policy that determines if a spot instance may be used for a Pod. Optional.

resources *ResourceSpec* Resource requirements for the Node that will run this Pod. If both instanceType and resources are specified, instanceType will take precedence. Optional.

placement *PlacementSpec* Placement is used to specify where a Pod will be place in the infrastructure. Optional.

volumes *[]Volume* List of volumes that will be made available to the Pod. Units can then attach any of these mounts. Optional.

securityContext *PodSecurityContext* Pod security context. Optional.

Volume

Definition for Volumes.

name *string* Name of the Volume. This is used when referencing a Volume from a Unit definition.

emptyDir *EmptyDir* If specified, an emptyDir will be created to back this Volume. Optional.

packagePath *PackagePath* This is a file or directory inside a package that will be mapped into the rootfs of a Unit. Optional.

EmptyDir

EmptyDir is is disk or memory-backed Volume. Units can use it as scratch space, or for inter-unit communication (e.g. one Unit fetching files into an emptyDir, another running a webserver, serving these static files from the emptyDir).

medium *StorageMedium* Backing medium for the emptyDir. The default is "" (to use disk space). The other option is "Memory", for creating a tmpfs volume. Optional.

sizeLimit *int64* SizeLimit is only meaningful for tmpfs. It is the size of the tmpfs volume. Optional.

ResourceSpec

ResourceSpec is used to specify resource requirements for the Node that will run a Pod.

cpu *string* The number of cpus on the instance. Must be a string but can be a fractional amount to accomodate shared cpu instance types (e.g. 0.5) Optional.

memory *string* The quantity of memory on the instance. Since this is a quantity gigabytes should be expressed as "Gi". E.G. memory: "3Gi" Optional.

gpu *string* Number of GPUs present on the instance. Optional.

volumeSize *string* Root volume size. Both AWS and GCE specify volumes in GiB. However according to their docs, AWS will bill you in GB. Optional.

dedicatedCPU *bool* Request an instance with dedicated or non-shared CPU. For AWS T2 instances have a shared CPU, all other instance families have a dedicated CPU. Set dedicatedCPU to true if you do not want Milpa to consider using a T2 instance for your Pod. Optional.

sustainedCPU *bool* Request unlimited CPU for T2 shared instance in AWS Only. https://docs.aws.amazon.com/AWSEC2 unlimited.html Optional.

privateIPOnly *bool* If PrivateIPOnly is true, the Pod will be launched on a Node without a public IP address. By default the Pod will run on a Node with a public IP address. Optional.

Unit

Units run applications. A Pod consists of one or more Units.

name *string* Name of the Unit.

image *string* The Docker image that will be pulled for this Unit. Usual Docker conventions are used to specify an image, see **https://docs.docker.com/engine/reference/commandline/tag/ #extended-description** for a detailed explanation on specifying an image.

Examples:

- `library/python:3.6-alpine`
- `myregistry.local:5000/testing/test-image` Optional.

command *[]string* The command that will be run to start the Unit. If empty, the entrypoint of the image will be used. See https://kubernetes.io/docs/tasks/inject-data-application/define-command-argument-container/#running-a-command-in-a-shell Optional.

args *[]string* Arguments to the command. If empty, the cmd from the image will be used. Optional.

env *[]EnvVar* List of environment variables that will be exported inside the Unit before start the application. Optional.

volumeMounts *[]VolumeMount* A list of Volumes that will be attached to the Unit. Optional.

ports *[]ServicePort* A list of ports that will be opened up for this Unit. Optional.

workingDir *string* Working directory to change to before running the command for the Unit. Optional.

securityContext *SecurityContext* Unit security context. Optional.

VolumeMount

VolumeMount specifies what Volumes to attach to the Unit and the path where they will be located inside the Unit.

name *string* Name of the Volume to attach.

mountPath *string* Path where this Volume will be attached inside the Unit.

EnvVar

Environment variables.

name *string* Name of the environment variable.

value *string* Value of the environment variable. Optional.

valueFrom *EnvVarSource* An environment variable may also come from a Secret. Optional.

PodSpot

PodSpot is the policy that determines if a spot instance may be used for a Pod.

policy *SpotPolicy* Spot policy. Can be “always”, “preferred” or “never”, meaning to always use a spot instance, use one when available, or never use a spot instance for running a Pod.

PodStatus

Last observed status of the Pod. This is maintained by the system.

phase *PodPhase* Phase is the last observed phase of the Pod. Can be “creating”, “dispatching”, “running”, “succeeded”, “failed” or “terminated”.

lastPhaseChange *Time* Time of the last phase change

readyTime *Time* Time after the pod was dispatched when the all units in the pod were running. Optional.

boundNodeName *string* Name of the node running this Pod.

boundInstanceID *string* ID of the node running this Pod.

addresses *[]NetworkAddress* IP addresses and DNS names of the Node running this Pod.

startFailures *int* Number of failures encountered while Milpa tried to start a Pod.

spotFailure *bool* Indicates if there was a failure finding a spot instance for this Pod. Only meaningful until the Pod gets to a “running” state. Optional.

unitStatuses *[]UnitStatus* Shows the status of the Units on the Pod with one entry for each Unit in the Pod’s Spec.

initUnitStatuses *[]UnitStatus* Shows the status of the init Units on the Pod with one entry for each init Unit in the Pod’s Spec.

Node

Node is a cloud instance that can run a Pod.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Object metadata.

spec *NodeSpec* Spec is the desired behavior of the Node.

status *NodeStatus* Status is the observed status of the Node. It is kept up to date by Milpa.

NodeSpec

NodeSpec defines the desired behavior of the Node.

instanceType *string* Cloud instance type of this Node.

bootImage *string* Cloud image that is used for this instance.

terminate *bool* Indicates that this Node has been requested to be terminated. Optional.

spot *bool* This is a spot cloud instance.

resources *ResourceSpec* Resource requirements necessary for booting this Node. If both instanceType and memory and cpu resources are specified, instanceType will take precedence. If the cloud provider allows a variable number of CPUs/memory for an instance type, the combination of resources and instance type will be used. Optional.

placement *PlacementSpec* Placement of the Node in the infrastructure. Optional.

NodeStatus

NodeStatus is the last observed status of a Node.

phase *NodePhase* Phase is the last observed phase of the Node.

instanceID *string* Cloud instance ID of this Node.

addresses *[]NetworkAddress* IP addresses and DNS names of this Node.

boundPodName *string* If a Pod is bound to this Node, this is the name of that Pod.

ReplicaSet

A ReplicaSet is a set of replicas of a certain Pod.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Object metadata.

spec *ReplicaSetSpec* Replica set specification. Optional.

status *ReplicaSetStatus* Last observed status of the replica. Optional.

ReplicaSetSpec

Replica set specification.

replicas *int32* Replicas is the number of desired replicas.

template *PodTemplateSpec* Template is the object that describes the pod that will be created if insufficient replicas are detected.

ReplicaSetStatus

Last observed status of the replica.

replicas *int32* Number of replicas.

availableReplicas *int32* The number of ready replicas for this replica set. Optional.

PodTemplateSpec

PodTemplateSpec is the object that describes the Pod that will be created if insufficient replicas are detected.

metadata *ObjectMeta* Object metadata.

spec *PodSpec* Spec defines the behavior of a Pod. Optional.

Deployment

Deployment enables declarative updates for Pods and ReplicaSets.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Object metadata.

spec *DeploymentSpec* Spec defines the behavior of a Deployment. Optional.

status *DeploymentStatus* Status is the last observed status of a Deployment. Optional.

DeploymentSpec

DeploymentSpec defines the behavior of a Deployment.

replicas *int32* Number of desired Pods. This is a pointer to distinguish between explicit zero and not specified. Defaults to 1.

template *PodTemplateSpec* Template describes the Pods that will be created.

paused *bool* Indicates that the Deployment is paused and will not be processed by the deployment controller. Optional.

maxUnavailable *int32* The maximum number of Pods that can be unavailable during the update. Value is an absolute number (ex: 5). This can not be 0 if MaxSurge is 0. By default, a fixed value of 1 is used.

maxSurge *int32* The maximum number of Pods that can be scheduled above the original number of Pods. Value must be an absolute number. By default, a fixed value of 1 is used.

DeploymentStatus

DeploymentStatus is the last observed status of a Deployment.

replicas *int32* Total number of non-terminated Pods targeted by this Deployment (their labels match the selector). This is the sum of Replicas from all replica sets.

phase *DeploymentPhase* Last observed phase of this Deployment. Can be “progressing” or “available”. Optional.

updatedReplicas *int32* Total number of non-terminated Pods targeted by this Deployment that have the desired template spec. Optional.

availableReplicas *int32* Total number of available Pods targeted by this Deployment. Optional.

unavailableReplicas *int32* Total number of unavailable Pods targeted by this Deployment. This is the total number of Pods that are still required for the Deployment to have 100% available capacity. They may either be Pods that are running but not yet available or Pods that still have not been created. Optional.

Secret

Secret holds secret data.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Object metadata.

data *map[string][]byte* A dictionary of secret data. The binary data itself should be base64 encoded.

Example:

```
data:
  password: cGFzc3dvcmQ=
```

Event

Event is a report of an event that happened in Milpa. They are stored separately from the objects they apply to.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Undocumented.

involvedObject *ObjectReference* The object that this event is about.

status *string* Should be a short, machine understandable string that describes the current status of the referred object. This should not give the reason for being in this state. Examples: “running”, “cantStart”, “cantSchedule”, “deleted”. It’s OK for components to make up statuses to report here, but the same string should always be used for the same status. Optional.

source *string* The component reporting this Event. Should be a short machine understandable string. Optional.

message *string* Human readable message about what happened. Optional.

LogFile

LogFile holds the log data created by a Pod Unit or a Node.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Undocumented.

parentObject *ObjectReference* The object that created this log. Optional.

Content *string* The content of the logfile. If the logfile is long, this will likely be the tail of the file. Optional.

Endpoints

Endpoints is a collection of endpoints that implement the actual Service. Example:

Name: “mysvc”, Subsets: [{ Addresses: [{“ip”: “10.10.1.1”}, {“ip”: “10.10.2.2”}], Ports: [{“name”: “a”, “port”: 8675}, {“name”: “b”, “port”: 309}] },] In Milpa, since Pods share an IP address and DNS name with Nodes, there will typically only be one subset per Endpoints structure.

kind *string* Kind is a string value for the resource this object represents. Optional.

apiVersion *string* APIVersion defines the versioned schema of this representation of an object. Optional.

metadata *ObjectMeta* Object metadata.

subsets *[]EndpointSubset* The set of all endpoints is the union of all subsets. Optional.

EndpointSubset

EndpointSubset is a group of addresses with a common set of ports. The expanded set of endpoints is the Cartesian product of Addresses x Ports. For example, given: { Addresses: [{“ip”: “10.10.1.1”}, {“ip”: “10.10.2.2”}], Ports: [{“name”: “a”, “port”: 8675}, {“name”: “b”, “port”: 309}] } The resulting set of endpoints can be viewed as: a: [10.10.1.1:8675, 10.10.2.2:8675], b: [10.10.1.1:309, 10.10.2.2:309]

addresses *[]EndpointAddress* Undocumented. Optional.

ports *[]EndpointPort* Undocumented. Optional.

EndpointAddress

EndpointAddress is a tuple that describes single IP address.

ip *string* The IP of this endpoint.

hostname *string* Optional: Hostname (FQDN) of this endpoint Meant to be used by DNS servers etc. Optional.

instanceID *string* Optional: InstanceID of the Node hosting this endpoint. Optional.

nodeName *string* Optional: Node hosting this endpoint. This can be used to determine endpoints local to a Node. Optional.

targetRef *ObjectReference* The object related to the endpoint. Optional.

EndpointPort

EndpointPort is a tuple that describes a single port.

name *string* The name of this port (corresponds to ServicePort.Name). Optional if only one port is defined. Must be a DNS_LABEL. Optional.

port *int32* The port number.

protocol *Protocol* The IP protocol for this port. Optional.

portRangeSize *int* The size of the port range for this port. Optional.