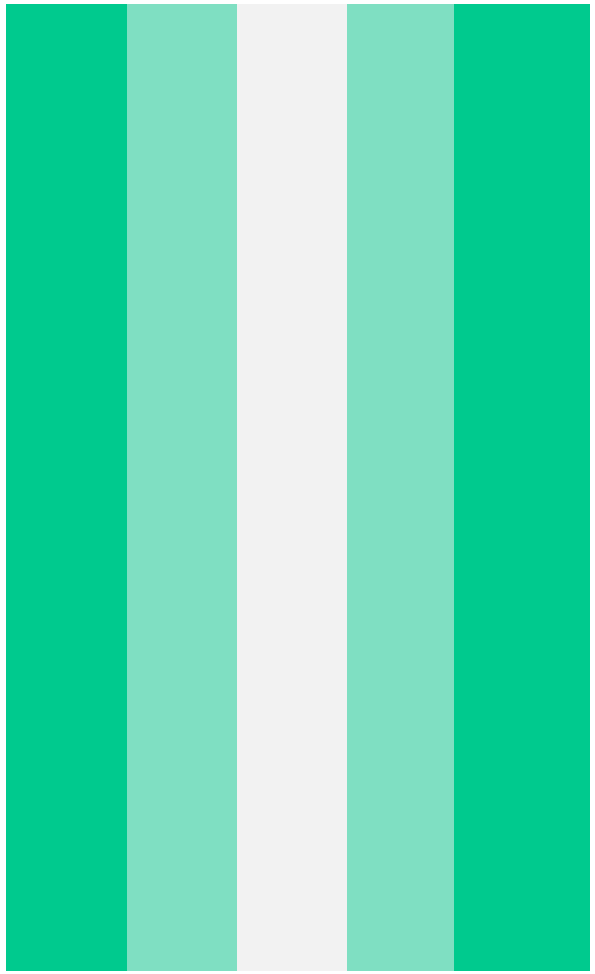
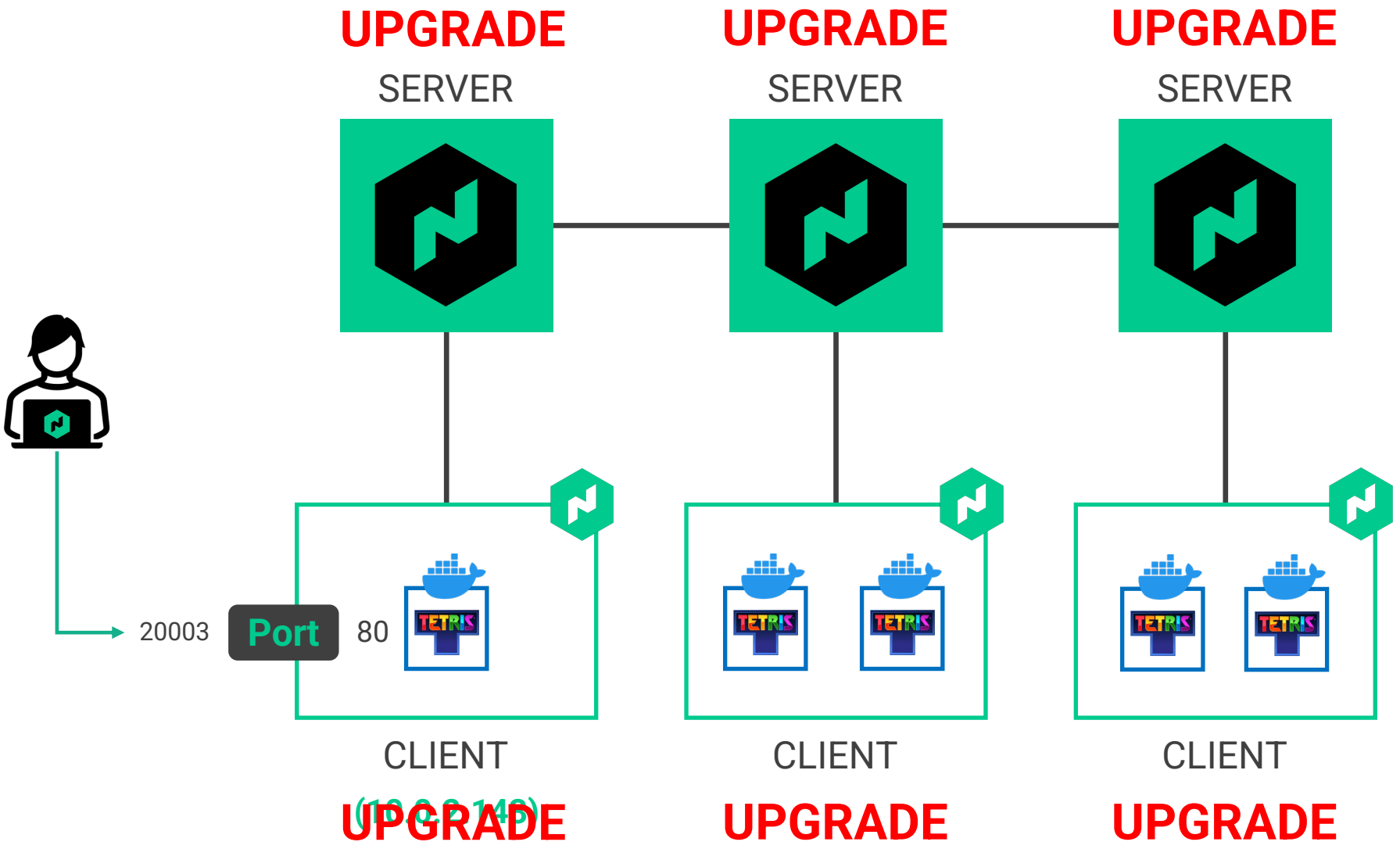


Expanding on Nomad Jobs

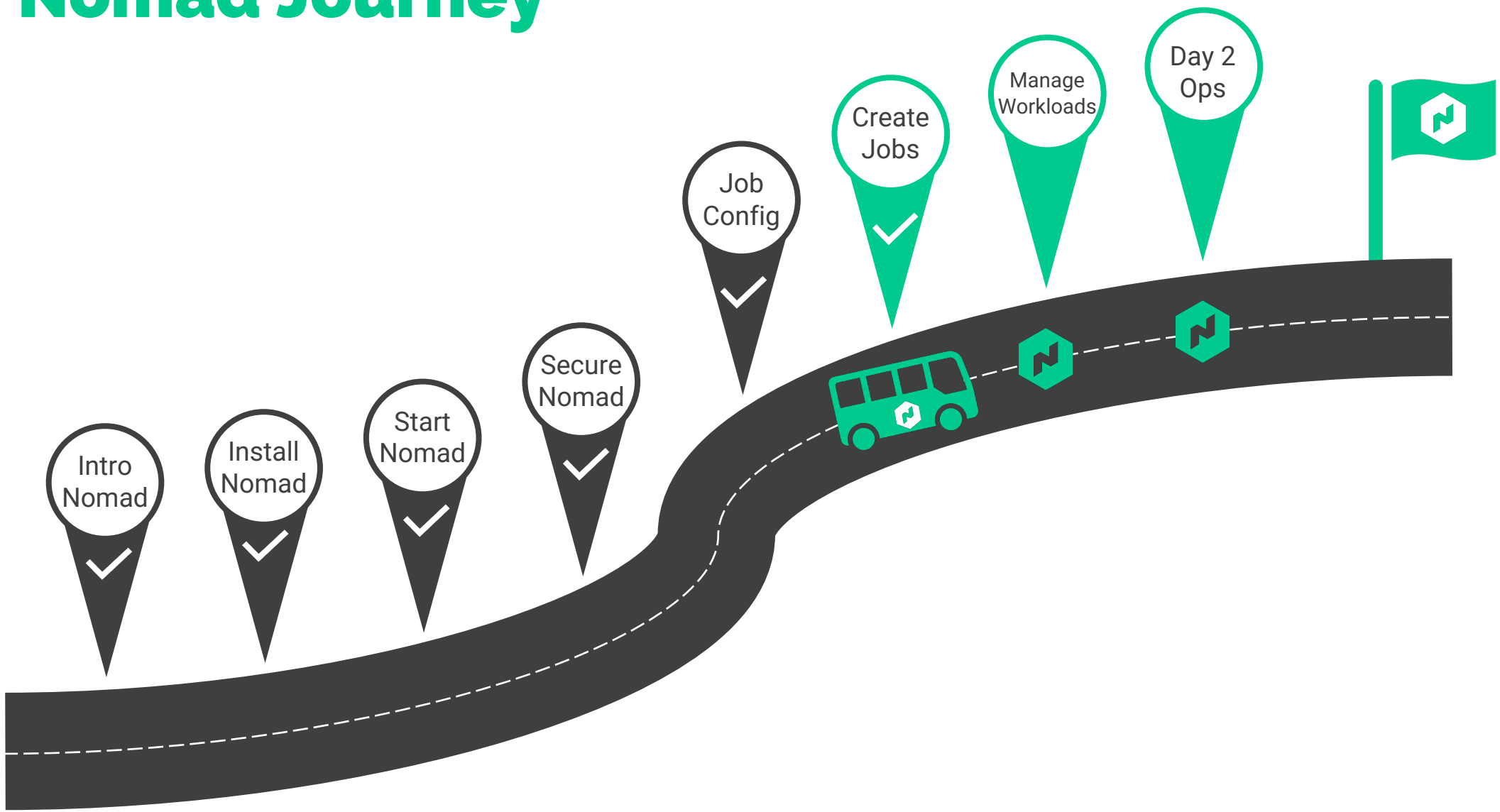
Expanding on Nomad Jobs



How Can We Improve Our Environment?

- ❑ Spread our application across nodes for high availability
- ❑ Monitor our application logs
- ❑ Upgrading the version of our application
- ❑ Improve the way we access our application
- ❑ Scale our application for dynamic workloads
- ❑ Use variables to limit hardcoding values
- ❑ Integrate Nomad with Vault and Consul

Nomad Journey



Job Placement

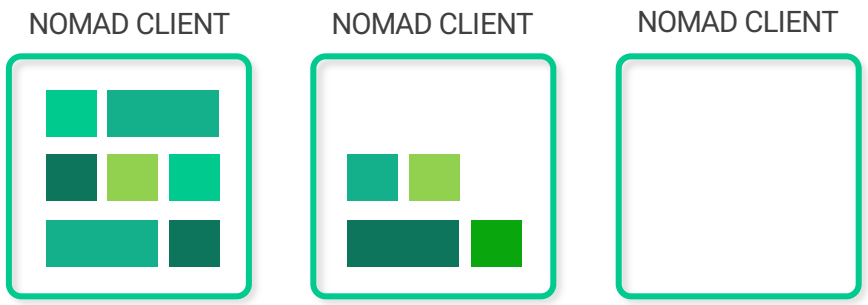
Job Placement

- When deploying applications, understanding where they will run is essential to ensure the maximum uptime and high availability for consumers
- By default, Nomad will use the "**binpack**" algorithm for allocations on available client nodes
- Bin packing can save costs by maximizing the resources of Nomad clients
- However, bin packing can introduce risk to your application because it may not be deployed across multiple client nodes
- You can configure Nomad scheduling to use a "**spread**" algorithm for the entire cluster, or you can customize it per job within the job specification

Placement

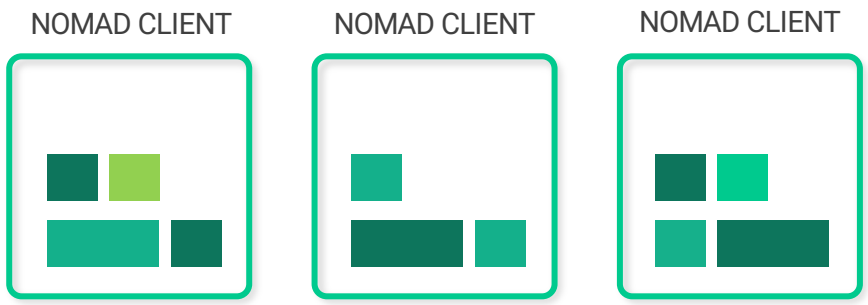
```
job → spread
job → group → spread
```

Bin Packing



Bin Packing (default)

- Utilize all of a node's resources before deploying applications on a different node
- Maximizes resource utilization and density
- Minimize infrastructure cost



Spread-Based Scheduling

- Evenly deploy applications by prioritizing based on nodes that are least utilized
- Distribute risk and ensures faster response and recovery in case of a node outage
- Optimized for performance

Customizable Scheduling

- Choose “binpack” or “spread”
- Customizable as one simple string in Nomad’s configuration
- Scheduling algorithm applies to all applications deployed on the cluster

```
# Server & Raft configuration
server {
  enabled          = true
  bootstrap_expect = 3
  encrypt          = "Do7GerAsNtzK527dxRZJwpJANdS2NTFbKJIxIod84u0="
  license_path     = "/etc/nomad.d/nomad.hclic"
  server_join {
    retry_join = ["10.4.23.44", "10.4.54.112", "10.4.56.33"]
  }
  default_scheduler_config {
    scheduler_algorithm = "spread"
  }
}
```

Job Placement

- When using spread, the scheduler will attempt to place allocations equally among the available values of the given target
- Spread can be used at the job level and/or the group level
- Spread can distribute tasks across datacenters if you have federated datacenters

```
job "tetris" {
  datacenters = ["dc1"]

  group "games" {
    count = 5

    spread {
      attribute = "${node.datacenter}"
      target "dc1" {
        percent = 100
      }
    }
```

Job Scheduling

```
job "tetris" {
  datacenters = ["dc1", "dc2"]

  group "games" {
    count = 5

    spread {
      attribute = "${node.datacenter}"
      target "dc1" {
        percent = 70
      }
      target "dc2" {
        percent = 30
      }
    }
```

Schedule 70% of allocations to DC1 and 30% to DC2

```
job "tetris" {
  datacenters = ["nyc", "sfo"]

  group "games" {
    count = 5

    spread {
      attribute = "${meta.dc}"
      target "nyc-prod" {
        percent = 50
      }
      target "sfo-dr" {
        percent = 50
      }
    }
```

Schedule 50% of allocations to the primary datacenter (NYC) and 50% to the DR datacenter (SFO) using user-defined metadata

Job Scheduling

```
job "tetris" {
  datacenters = ["dc1", "dc2"]

  spread {
    attribute = "${node.datacenter}"
    target "dc1" {
      percent = 70
    }
    target "dc2" {
      percent = 30
    }
  }

  group "frontend" {
    # ...
    task "webapp" {
      # ...

   group "backend" {
    # ...
    task "data" {
      # ...
```

Spread is now under the job stanza

Job → Spread

This applies to all groups and tasks in the job specification

Multiple groups in the job specification

Job Scheduling

Multiple Spread Configurations

Spread all groups included in the job specification across both the **nyc** and **sfo** datacenters

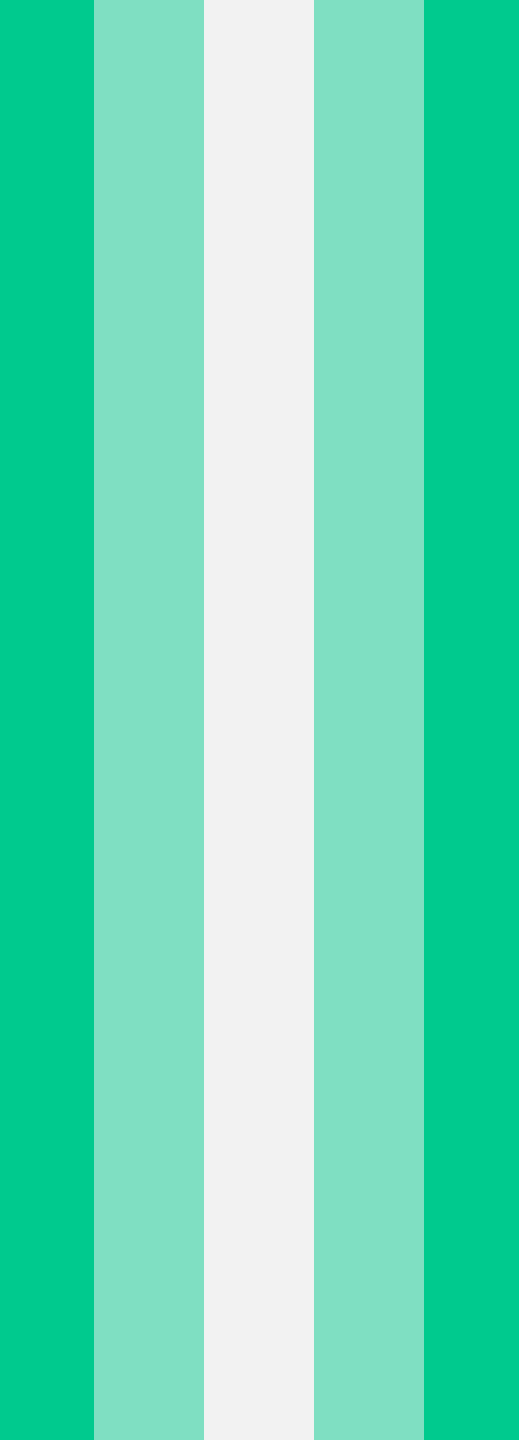
Within each datacenter, spread allocations for the **frontend** group across Nomad clients that have user-defined metadata for **prod1** and **prod2**

Note: If there is a conflict, the group level spread will take priority

```
job "tetris" {
  datacenters = ["nyc", "sfo"]

  spread {
    attribute = "${node.datacenter}"
    target "nyc" {
      percent = 70
    }
    target "sfo" {
      percent = 30
    }
  }

  group "frontend" {
    spread {
        attribute = "${meta.env}"
        target "prod1" {
            percent = 50
        }
        target "prod2" {
            percent = 50
        }
    }

  group "webapp" {
      # ...

```

DEMO

Job Placement



Job Constraints

Job Constraints

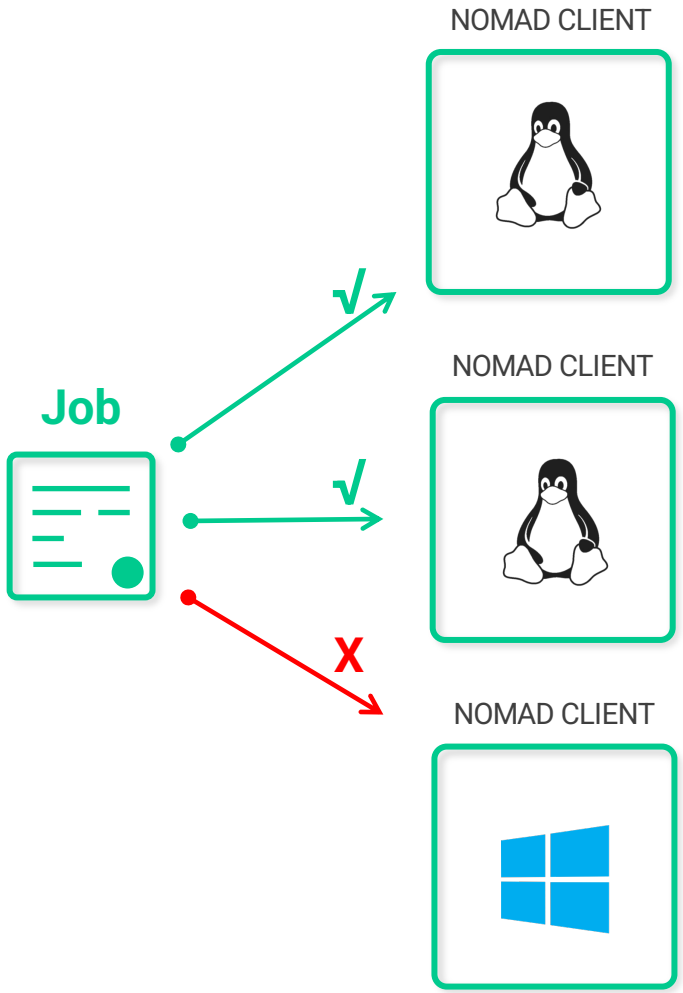
- Constraints are requirements Nomad must evaluate about the client, such as the operating system, architecture, kernel version, and more before allocations are made….
- Constraint requirements are specified at the job, group, or task level
- Examples:
  - Client must be Linux
  - Client must be running x64 architecture
  - Client must be running on an underlying AWS instance that is m5.8xlarge
  - Client must have specific metadata

Placement

```
job → constraint
job → group → constraint
job → group → task → constraint
```

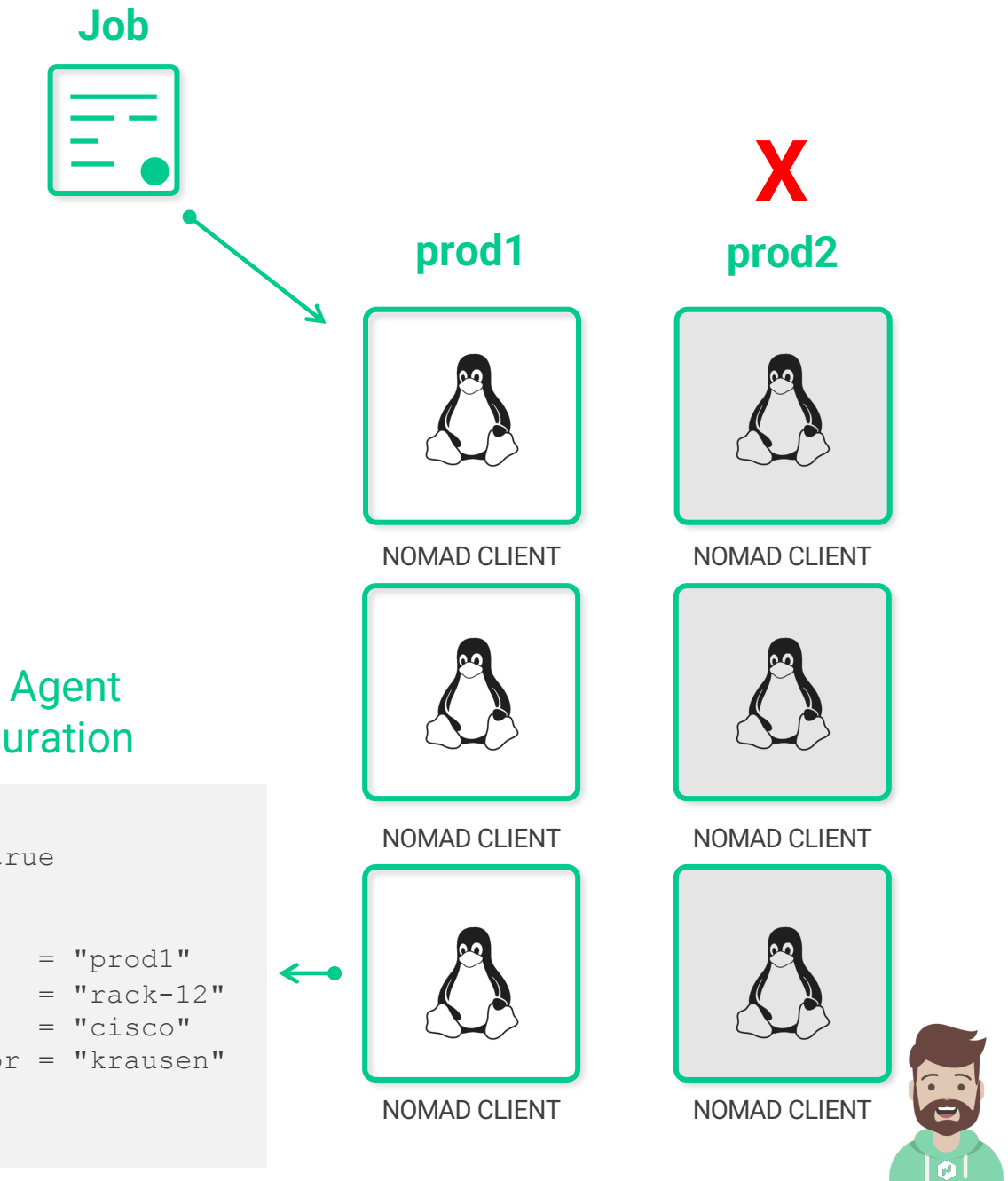
Job Constraints

```
job "tetris" {
  datacenters = ["nyc", "sfo"]

  constraint {
    attribute = "${attr.kernel.name}"
    value     = "linux"
  }
```



Job Constraints

```
job "tetris" {
  datacenters = ["nyc", "sfo"]

  constraint {
    attribute = "${meta.env}"
    value     = "prod1"
  }
```



Job Constraints

Additional Examples

Distinct Hosts

Don't co-locate any tasks on the same client

```
group "frontend" {
  constraint {
    operator  = "distinct_hosts"
    value     = "true"
  }

  task {
      driver = "docker"
      #...
  }
```

Cloud Metadata

Prevent the task to only run on nodes of size m4.xlarge

```
group "frontend" {
  constraint {
    attribute = "${attr.platform.aws.instance-type}"
    value     = "m4.xlarge"
  }

  task {
    driver = "docker"
    #...
  }
```

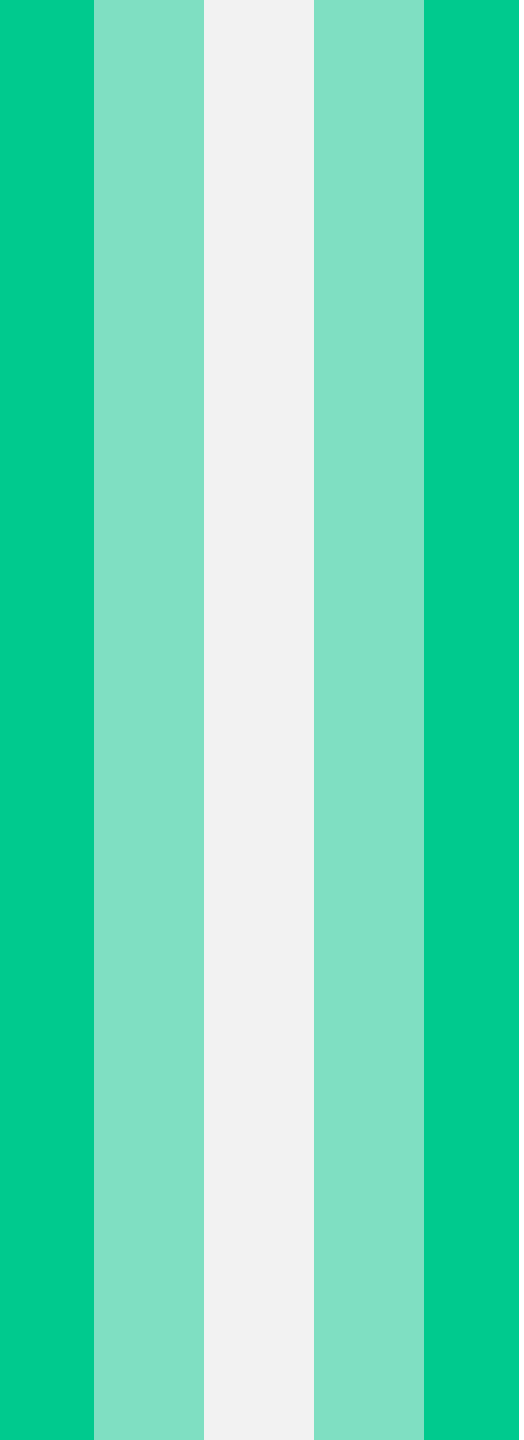
Client OS

Restrict the task to running on clients running Ubuntu

```
group "frontend" {
  constraint {
    attribute = "${attr.os.name}"
    value     = "ubuntu"
  }

  task {
    driver = "docker"
    #...
  }
```

DEMO

Job Constraints



Networking

Networking

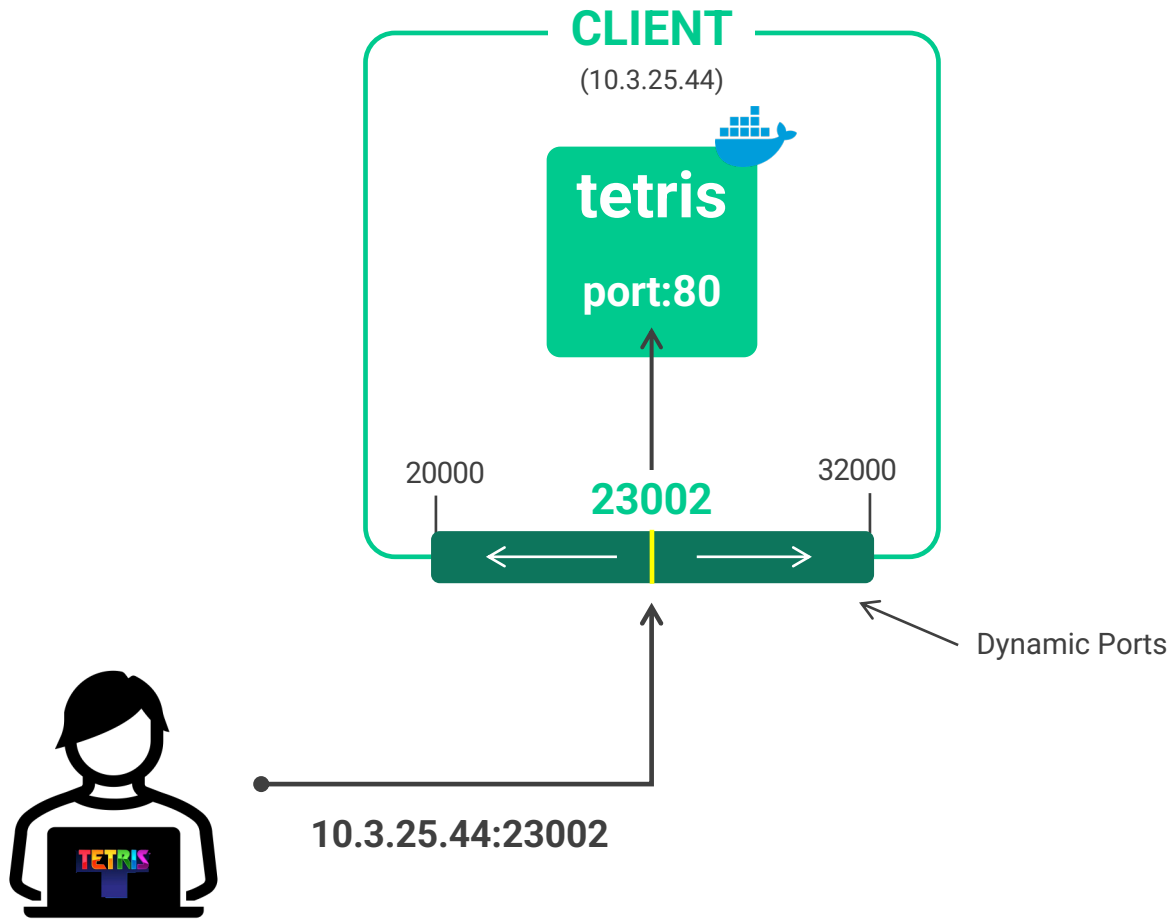
- When deploying applications, networking is a critical component to ensure users can access the running applications
- Networking is defined at the group level using the `network` stanza
- Options for networking include:
  - `bridge`: group will have an isolated network namespace with an interface bridged with the host
  - `host`: (default) - each task will join the host network namespace – a shared network namespace is not created
  - `cni/<network>`: task group will have an isolated network namespace with the CNI network
  - `none`: each task will have an isolated network without any network interfaces

Placement { `job` → `group` → `network`

Networking

Host Mode

- Access via host IP address and dynamically allocated port
- Default port range is 20000 – 32000
- Relies on the task drivers to implement port mapping



Networking Mode - Host

```
1  group "games" {
2    count = 5
3
4    network {
5      port "web" {
6        to = 80
7      }
8    }
9
10   task "tetris" {
11     driver = "docker"
12
13     config {
14       image          = "bsord/tetris"
15       ports          = ["web"]
16       auth_soft_fail = true
17     }
```

This configuration is used when your application listens on a fixed port and you want to map to a dynamic port on the host

- `port` – used to specify both dynamic and reserved ports. This example uses a reserved port using the `to` parameter
- uses a label called "`web`"
- Note that we do not need a `mode = host` here because that is the default configuration

Reference the port label inside the task

Networking Mode - Host

This configuration is used when your application listens on a fixed port, and you need to map it to a static port on the host

- `port` – used to specify both dynamic and reserved ports. This example specifies the `static` port to allocate on the host (not recommended)

Reference the port label inside the task

```
1  group "games" {
2    count = 5
3
4    network {
5      port "web" {
6        static = 80
7        to = 80
8      }
9    }
10
11   task "tetris" {
12     driver = "docker"
13
14     config {
15       image          = "bsord/tetris"
16       ports          = ["web"]
17       auth_soft_fail = true
18     }
```

Networking Mode - Host

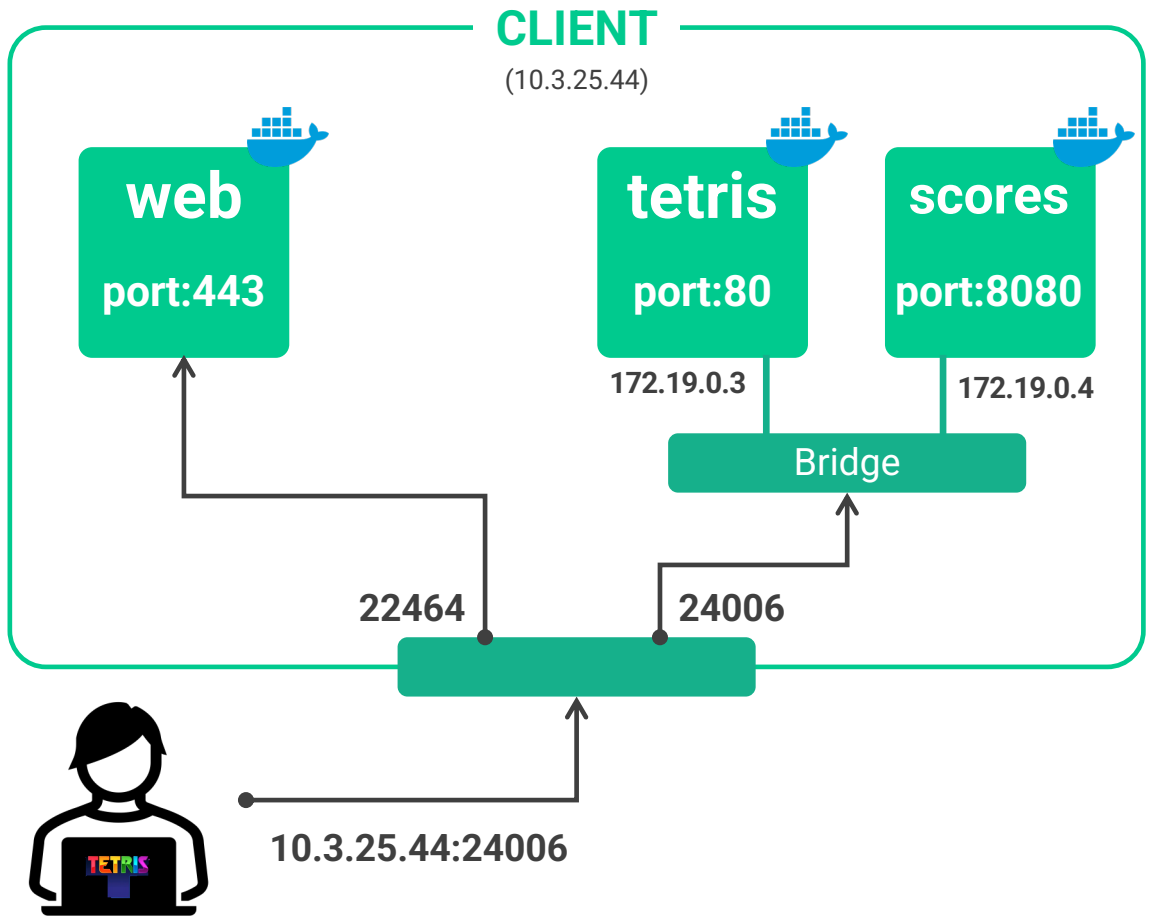
```
1  group "games" {
2    count = 5
3
4    network {
5      port "web" {}
6    }
7
8    task "tetris" {
9      driver = "docker"
10
11     config {
12       image          = "bsord/tetris"
13       ports          = ["web"]
14       auth_soft_fail = true
15     }
```

- No configuration specifies a dynamic port allocation for the task
- The application/service would need to read an environment variable to know the port to bind to at startup
- `NOMAD_PORT_web` would be passed to the task so it can know the dynamic port
- Again, no need to specify `mode = host` because it's the default configuration

Networking

Bridge Mode

- Access via host IP address and dynamically allocated port
- Default port range is 20000 - 32000



Networking Mode - Bridge

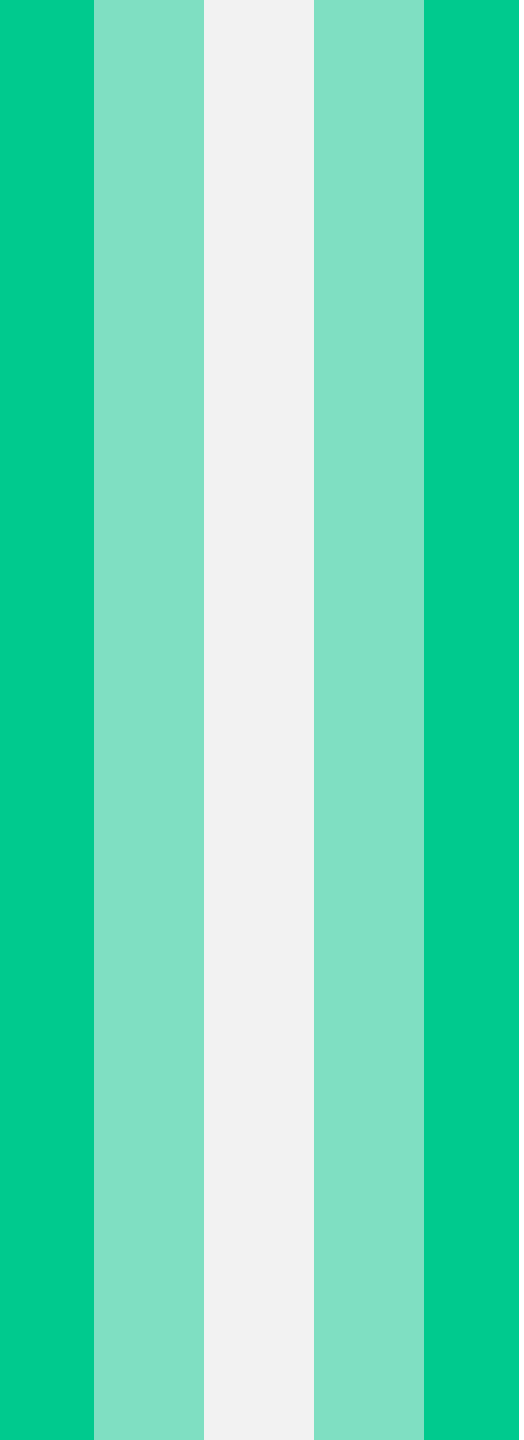
This configuration is used when your application listens on a fixed port, and you want to connect to a bridged network on the host

- `mode = bridge` required since it's not the default configuration

Reference the port label inside the task

```
1  group "games" {
2    count = 5
3
4    network {
5      mode = "bridge"
6      port "http" {
7        to = 8080
8      }
9    }
10
11   task "tetris" {
12     driver = "docker"
13
14     config {
15       image          = "bsord/tetris"
16       ports          = ["http"]
17       auth_soft_fail = true
18     }
```

DEMO | Networking



Working with Volumes

Working with Volumes

- For workloads that require storage for stateful workloads, Nomad can provide access to host volumes or CSI volumes
- Nomad is aware of host volumes during the scheduling process, allowing it to make scheduling decisions based on the availability of host volumes on a specific client
- However, Nomad is **NOT** aware of Docker volumes because they are managed outside of Nomad. Therefore, the scheduler cannot make decisions based on availability
- Of course, volumes are available to other resources beyond containers, such as `exec` and `java` apps

Placement { `job` → `group` → `volume`

Working with Volumes

- A `host` volume is essentially a path on the client that is made available to jobs. The data is stored locally on the Nomad client

- A `csi-volume` is exposed to jobs using a CSI plugin to consume externally created storage volumes. Examples of these include AWS EBS volumes, GCP persistent disks, Ceph, vSphere, and more

Placement { `job` → `group` → `volume`

Host Volumes

Host volumes must be first configured using the `client` block in the Nomad agent configuration file:

```
# Client Configuration - Node can be Server & Client
client {
  enabled = true
  server_join {
    retry_join = ["provider=aws tag_key=nomad_cluster_id tag_value=us-east-1"]
  }
  host_volume "database-primary" {
    path = "/opt/nomad/volumes/db-pri"
    read_only = false
  }
  host_volume "web-temp-data-01" {
    path = "/opt/nomad/volumes/web-temp"
    read_only = false
  }
}
```

Name of Volume

Path on Nomad client

Host Volumes

```
1  group "games" {
2    count = 5
3
4    # ...
5    volume "data-01" {
6      type      = "host"
7      source    = "web-temp-data-01"
8      read_only = false
9    }
10
11   task "tetris" {
12     driver = "docker"
13
14     volume_mount {
15       volume      = "data-01"
16       destination = "/var/lib/http"
17       read_only   = false
18     }
19
20     config {
21       image           = "bsord/tetris"
22       ports           = ["http"]
23       auth_soft_fail = true
24     }
25
26     resources {
27       cpu    = 50
28       memory = 256
29     }
30   }
31 }
```

Specify the Required Volume for the `games` Group

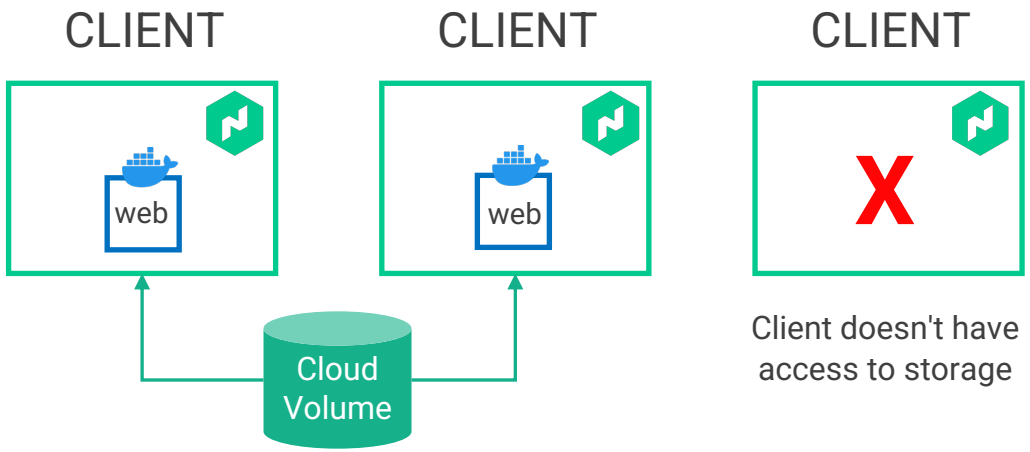
Specifies where a group volume should be mounted

- Once the volume is ready to use, it is referenced *within the group* using the `volume` parameter

- The task will reference the volume *required for the task to run* using the `volume_mount` parameter

CSI Volumes

- CSI volumes are dynamically mounted by CSI plugins.
- These plugins actually run as a system job and can mount volumes created by cloud providers or storage platforms. No updates to the client agent configuration file needed*!
- Again, Nomad is aware of CSI volumes, which allows Nomad to schedule your workloads based on the availability of volumes on a client



* ok, maybe one small one but not per volume

CSI Volumes

- Since the CSI plugins are written by the storage vendors, any CSI plugin that supports Kubernetes should work with Nomad

- CSI volumes are supported by a large number of storage platforms and cloud providers, including:

  - Alicloud Disk/NAS/OSS
  - AWS EBS
  - AWS EFS
  - AWS FSx
  - Azure Blob/Disk/File
  - CephFS
  - Cisco Hyperflex
  - Dell EMC PowerMax/PowerScale/Unity/etc
  - Google Cloud Filestore/Storage
  - IBM
  - NFS
  - Portworx (Pure)
  - SMB
  - Synology
  - vSphere

https://kubernetes-csi.github.io/docs/drivers.html

Required Steps for CSI Volumes

1. Create the volume on your platform of choice
2. Enable Privileged Docker jobs
3. Run the plugin job for the specific driver needed
4. Register the volume with Nomad
5. Deploy the application that will use the volume

Enable Priviledged Docker Jobs

- CSI plugins run as a privileged Docker job since they use bidirectional mount propagation to mount disks to the host
- The default configuration does not allow privileged Docker jobs, so you must *update your client configuration* to permit them

```
# Client Configuration - Node can be Server & Client
client {
  enabled = true
  server_join {
    retry_join = ["10.3.2.16","10.4.33.66", "10.3.2.18"]
  }
}

plugin "docker" {
  config {
    allow_privileged = true
  }
}
```

Create the Plugin Job

- Each CSI plugin supports one or more types of jobs – Controller, Nodes, or Monolith
- Most CSI plugins require that you run both a controller and a nodes job
- Node plugins are usually run as `system` jobs
  - These jobs use the `csi_plugin` stanza in the job spec file

Create the Plugin Job

this is a **`system`** job

location of the driver image

Run as a privileged job

Specifies this task provides a CSI plugin to the cluster

```
job "plugin-aws-ebs-nodes" {
  datacenters = ["dc1"]

  type = "system"

  group "nodes" {
    task "plugin" {
      driver = "docker"

      config {
        image = "amazon/aws-ebs-csi-driver:v0.10.1"

        args = [
          "node",
          "--endpoint=unix://csi/csi.sock",
          "--logtostderr",
          "--v=5",
        ]

        privileged = true
      }

      csi_plugin {
        id        = "aws-ebs0"
        type      = "node"
        mount_dir = "/csi"
      }

      resources {
        cpu    = 500
        memory = 256
      }
    }
  }
}
```

Deploy the Plugin Jobs to Nomad

Once you have the plugin job specifications ready, you can create the jobs in Nomad:

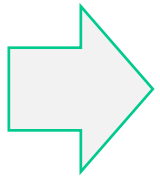
TERMINAL

```
$ nomad job run csi_plugin.hcl

==> 2023-01-26T17:20:29-05:00: Monitoring evaluation "b66aaf83"
    2023-01-26T17:20:29-05:00: Evaluation triggered by job "plugin-aws-ebs-nodes"
    2023-01-26T17:20:30-05:00: Allocation "f329afdd" created: node "7ff357a0", group "nodes"
    2023-01-26T17:20:30-05:00: Evaluation status changed: "pending" -> "complete"
==> 2023-01-26T17:20:30-05:00: Evaluation "b66aaf83" finished with status "complete"
```

Register the Volume

Each volume needs to be registered with Nomad to ensure the CSI plugins know about each volume

```
# volume registration
type        = "csi"
id          = "web-data"
name        = "mysql"
external_id = "vol-5c699e29779e11fa7"
plugin_id   = "aws-ebs"

capability {
  access_mode     = "single-node-writer"
  attachment_mode = "file-system"
}
```



TERMINAL

```
$ nomad volume register ebs_volume.hcl

ID = web-data
Name = web-data
External ID = vol-5c699e29779e11fa7
Plugin ID = aws-ebs
Provider = ebs.csi.aws.com
Version = v0.10.1
…
```

Create the Job to Use the Volume

And….finally. Create the job for the task(s) that will consume the new CSI volume

```
job "tetris" {
  datacenters = ["dc1"]
  type        = "service"

  group "tetris" {
    count = 1

    volume "web-data-temp" {
      type            = "csi"
      read_only       = false
      source          = "web-data"
      access_mode     = "single-node-writer"
      attachment_mode = "file-system"
    }

    task "mysql-server" {
      driver = "docker"

      volume_mount {
        volume      = "web-data-temp"
        destination = "/pdata"
        read_only   = false
      }

```

DEMO

Nomad Volumes