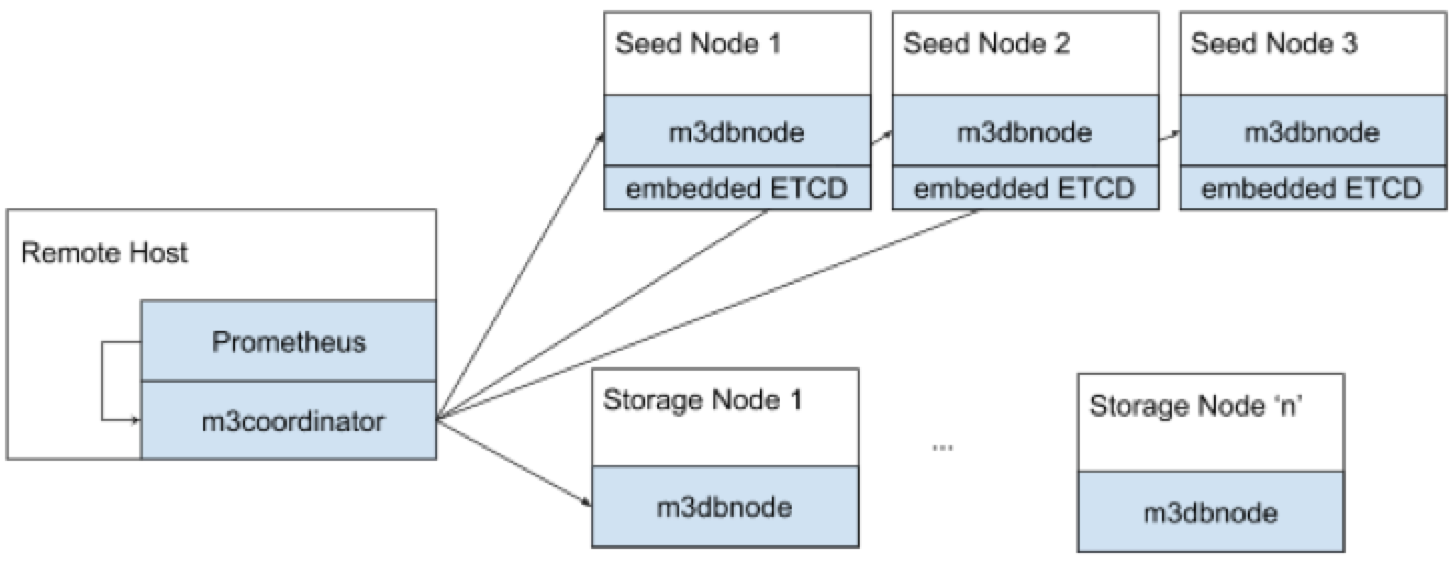
Creating an M3 Cluster with Binaries

This guide shows you the steps involved in creating an M3 cluster using M3 binaries, typically you would automate this with infrastructure as code tools such as Terraform or Kubernetes.

Note

This guide assumes you have read the quickstart, and builds upon the concepts in that guide.

M3 Architecture



Node types

An M3 deployment typically has two main node types:

- Storage nodes (`m3dbnode`) are the workhorses of M3, they store data and serve reads and writes.
- Coordinator nodes (`m3coordinator`) coordinate reads and writes across all nodes in the cluster. It's a lightweight process, and does not store any data. This role typically runs alongside a Prometheus instance, or is part of a collector agent such as statsD.

A `m3coordinator` node exposes two external ports:

- `7201` to manage the cluster topology, you make most API calls to this endpoint
- `7203` for Prometheus to scrape the metrics produced by M3DB and M3Coordinator

There are two other less-commonly used node types:

- Query nodes (`m3query`) are an alternative query option to using M3's built-in PromQL support.
- Aggregator nodes cluster and aggregate metrics before storing them in storage nodes. Coordinator nodes can also perform this role but are not cluster-aware.

Prerequisites

M3 uses etcd as a distributed key-value storage for the following functions:

- Update cluster configuration in realtime
- Manage placements for distributed and sharded clusters

Download and Install a Binary

You can download the latest release as pre-compiled binaries from the M3 GitHub page. Inside the expanded archive are binaries for `m3dbnode`, which combines a coordinator and storage node, and a binary for `m3coordinator`, which is a standalone coordinator node.

Build from Source

Prerequisites

- Go
- Make

Build

Clone the codebase and run `make m3dbnode` to generate a binary for a combination coordinator and storage node, or `make m3coordinator` to generate a binary for a standalone coordinator node.

Provision a Host

M3 in production can run on local or cloud-based VMs, or bare-metal servers. M3 supports all popular Linux distributions (Ubuntu, RHEL, CentOS), and let us know if you have any issues with your preferred distribution.

Network

Tip

If you use AWS or GCP, we recommend you use static IPs so that if you need to replace a host, you don’t have to update configuration files on all the hosts, but decommission the old seed node and provision a new seed node with the same host ID and static IP that the old seed node had. If you’re using AWS you can use an Elastic Network Interface on a Virtual Private Cloud (VPC) and for GCP you can use an internal static IP address.

This example creates three static IP addresses for three storage nodes, using the embedded coordinator.

This guide assumes you have host names configured, i.e., running `hostname` on a host in the cluster returns the host ID you use when creating the cluster placement.

Tip

When using GCP the name of your instance is the host name. When you create an instance, click *Management, disks, networking, SSH keys*, under *Networking*, click the default interface, click the *Primary internal IP* drop down, select *Reserve a static internal IP address*, give it an appropriate name and description, and use *Assign automatically*.

Tip

When using AWS, you can use the host name supplied for the provisioned VM as your host ID, or use the `environment` host ID resolver and pass the host ID when launching the database process with an environment variable.

For example, if you used `M3DB_HOST_ID` for the environment variable name, use the following in your configuration:

```
hostID:
  resolver: config
  value: ${M3DB_HOST_ID:""}
```

Then start the `m3dbnode` process with:

Pre-built binary Output

```
M3DB_HOST_ID=m3db001 m3dbnode -f <config-file.yml>
```

Info

Depending on your operating system setup, you might need to prefix the command with `sudo`.

Kernel Configuration

Depending on the default limits of your bare-metal machine or VM, M3 may need some Kernel tweaks to run as efficiently as possible, and we recommend you review those before running M3 in production.

Configuration files

You configure each M3 component by passing the location of a YAML file with the `-f` argument.

The file groups configuration into `coordinator` or `db` sections that represent the `M3Coordinator` and `M3DB` instances of single-node cluster.

Tip

You can find more information on configuring M3DB in the operational guides section.

Note

The steps in this guide have the following 3 seed nodes, you need to change your configuration to suit the details of yours, including the details of an etcd cluster in the `m3dbCluster` > `endpoints` section of the M3 configuration file.

- m3db001 (Region=us-east1, Zone=us-east1-a, Static IP=10.142.0.1)
- m3db002 (Region=us-east1, Zone=us-east1-b, Static IP=10.142.0.2)
- m3db003 (Region=us-east1, Zone=us-east1-c, Static IP=10.142.0.3)

M3DB node

Start with the M3DB configuration template and change it to suit your cluster.

The example below connects to an etcd instance in a zone called `eu-1` This example updates the `service` and `seedNodes` sections to match the node details above:

```
config:
  discovery:
    type: m3db_cluster
    m3dbCluster:
      env: default_env
      endpoints:
        - 10.142.0.1:2379
        - 10.142.0.2:2379
        - 10.142.0.3:2379
```

Start the storage nodes

Start each seed node in the cluster using the same configuration file, and adjusting the `M3DB_HOST_ID` value to match the host name.

```
M3DB_HOST_ID=m3db001 m3dbnode -f <config-file.yml>
```

Tip

You can daemon-ize the node startup process using your favorite utility such as systemd, init.d, or supervisor.

Create Namespace and Initialize Placement

This guide uses the *http://localhost:7201/api/v1/database/create* endpoint that creates a namespace, and the placement if it doesn't already exist based on the `type` argument.

You can create placements and namespaces separately if you need more control over their settings.

In the example below, the configuration for each host matches the details outlined above for the three nodes used. `isolationGroup` specifies how the cluster places shards to avoid more than one replica of a shard appearing in the same replica group. You should use at least as many isolation groups as your replication factor. This example uses the availability zones `us-east1-a`, `us-east1-b`, `us-east1-c` as the isolation groups which matches our replication factor of 3. Read more details in this guide.

Command Output

```
curl -X POST http://localhost:7201/api/v1/database/create -d '{
  "type": "cluster",
  "namespaceName": "default",
  "retentionTime": "48h",
  "numShards": "1024",
  "replicationFactor": "3",
  "hosts": [
        {
            "id": "m3db001",
            "isolationGroup": "us-east1-a",
            "zone": "embedded",
            "weight": 100,
            "address": "10.142.0.1",
            "port": 9000
        },
        {
            "id": "m3db002",
            "isolationGroup": "us-east1-b",
            "zone": "embedded",
            "weight": 100,
            "address": "10.142.0.2",
            "port": 9000
        },
        {
            "id": "m3db003",
            "isolationGroup": "us-east1-c",
            "zone": "embedded",
            "weight": 100,
            "address": "10.142.0.3",
                "port": 9000
            }
        ]
    }'
```

If you need to setup multiple namespaces, you can run the command above multiple times with different namespace configurations.

Ready a Namespace

Once a namespace has finished bootstrapping, you must mark it as ready before receiving traffic by using the *http://localhost:7201/api/v1/namespace/ready*.

Command Output

```
#!/bin/bash
curl -X POST http://localhost:7201/api/v1/services/m3db/namespace/ready -
d '{
  "name": "default"
}' | jq .
```

Replication factor

We recommend a replication factor of **3**, with each replica spread across failure domains such as a physical server rack, data center or availability zone. Read our replication factor recommendations for more details.

Shards

Read the placement configuration guide to determine the appropriate number of shards to specify.

Writing and Querying Metrics

Writing Metrics

M3 supports ingesting statsd and Prometheus formatted metrics.

This quickstart focuses on Prometheus metrics which consist of a value, a timestamp, and tags to bring context and meaning to the metric.

You can write metrics using one of two endpoints:

- *http://localhost:7201/api/v1/prom/remote/write* - Write a Prometheus remote write query to M3DB with a binary snappy compressed Prometheus WriteRequest protobuf message.
- *http://localhost:7201/api/v1/json/write* - Write a JSON payload of metrics data. This endpoint is quick for testing purposes but is not as performant for production usage.

For this quickstart, use the *http://localhost:7201/api/v1/json/write* endpoint to write a tagged metric to M3 with the following data in the request body, all fields are required:

- `tags`: An object of at least one `name`/`value` pairs
- `timestamp`: The UNIX timestamp for the data
- `value`: The value for the data, can be of any type

Tip

The examples below use `__name__` as the name for one of the tags, which is a Prometheus reserved tag that allows you to query metrics using the value of the tag to filter results.

Tip

Label names may contain ASCII letters, numbers, underscores, and Unicode characters. They must match the regex `[a-zA-Z_][a-zA-Z0-9_]*`. Label names beginning with `__` are reserved for internal use. Read more in the Prometheus documentation.

Command 1 Command 2 Command 3

```
#!/bin/bash
curl -X POST http://localhost:7201/api/v1/json/write -d '{
  "tags":
    {
      "__name__": "third_avenue",
      "city": "new_york",
      "checkout": "1"
    },
    "timestamp": '\"$(date "+%s")\"',
    "value": 3347.26
}'
```

Querying metrics

M3 supports three query engines: Prometheus (default), Graphite, and the M3 Query Engine.

This quickstart uses Prometheus as the query engine, and you have access to all the features of PromQL queries.

To query metrics, use the *http://localhost:7201/api/v1/query_range* endpoint with the following data in the request body, all fields are required:

- `query`: A PromQL query
- `start`: Timestamp in `RFC3339Nano` of start range for results
- `end`: Timestamp in `RFC3339Nano` of end range for results
- `step`: A duration or float of the query resolution, the interval between results in the timespan between `start` and `end`.

Below are some examples using the metrics written above.

Return results in past 45 seconds

Linux macOS/BSD Output

```
curl -X "POST" -G "http://localhost:7201/api/v1/query_range" \
  -d "query=third_avenue" \
  -d "start=$(date "+%s" -d "45 seconds ago")" \
  -d "end=$( date +%s )" \
  -d "step=5s" | jq .
```

Values above a certain number

Linux macOS/BSD Output

```
curl -X "POST" -G "http://localhost:7201/api/v1/query_range" \
  -d "query=third_avenue > 6000" \
  -d "start=$(date "+%s" -d "45 seconds ago")" \
  -d "end=$( date +%s )" \
  -d "step=5s" | jq .
```

Was this page helpful?

- Suggest edits to this page