CHAPTER 7

Deployment and Monitoring

This chapter looks at the most common tools and techniques for automating MongoDB deployments in production, on-prem, and in cloud-based self-managed environments. We also review different monitoring solutions and tips to help DevOps teams avoid issues and react quickly to changes in the deployment.

The DevOps Toolkit

The DevOps toolkit is still evolving very quickly as cloud providers continue to add more features, and large enterprises are moving more of their workloads out into the “public cloud”. All details provided are accurate at the time of writing.

Goals

Ideally, our DevOps toolkit should closely mirror our database requirements, namely, scalability as the deployment grows, isolation of components, durability of data, and performance.

Community vs. Enterprise

MongoDB Inc. (the company behind MongoDB) chose early to make the database an open source product partly to drive adoption, but also in the belief that an open source database would become more secure, stable, and trusted.

In order to build a business, the company also built some additional features reserved for Enterprise customers. The DevOps tools mentioned in this chapter fall into this category. Any licensed customers are free to use these tools on-prem or on self-
managed cloud infrastructure and also receive unlimited technical support to ensure their deployments are online and optimal at all times.

MongoDB releases follow the common paradigm of even-numbered releases for production, and odd numbers for preproduction work. For example, version 4.3 is a preproduction release for testing, which is then generally released for production as version 4.4.

Minor releases such as 4.4.1 to 4.4.2 include security and performance improvements, but never break backward compatibility. You should normally upgrade to any minor releases within weeks to benefit from important stability improvements and bug fixes. By contrast, major version upgrades should be tested in a UAT environment with real data and workloads before upgrading in a production deployment.

**Ops/Cloud Manager**

*Ops Manager* is the Enterprise management platform for MongoDB which is designed to deploy, monitor, back up, and scale MongoDB deployments on your own infrastructure. *Cloud Manager* is almost identical except that the metrics and backups are stored and managed by MongoDB Inc., but actual user data never leaves your own infrastructure. Table 7-1 shows the differences between Ops and Cloud Manager.

**Table 7-1. A comparison of Ops and Cloud Manager**

|                                   | Ops Manager                                      | Cloud Manager                                      |
|-----------------------------------|--------------------------------------------------|-----------------------------------------------------|
| Automates deployments             | Yes, on-prem                                     | Yes, on-prem                                        |
| Monitors deployments              | Yes, all metadata and metrics on-prem            | Yes, but metadata and metrics in the cloud          |
| Backs up deployments              | Yes, backups can be all on-prem (using filesystem, blockstore, or local S3-compatible store), or to Amazon S3 cloud storage | Yes, backups are stored on an Amazon S3 bucket managed by MongoDB the company |
| Requirements                      | Automation agents installed on all hosts          | Automation agents installed on all hosts            |
|                                   | Replica set to store Ops Manager config (AppDB)   |                                                     |
|                                   | Replica set to store oplog and blockstore (for backups) |                                                     |
|                                   | Hosts for Ops Manager instances (plus optional load balancer) |                                                     |
Both tools are able to help manage, reconfigure, grow, and upgrade versions of your MongoDB deployments all in a rolling manner to avoid downtime. For anything more than a single replica set in production, the risk of manually applying updates is otherwise too high.

### Rolling changes

When it comes to automating production MongoDB deployments, making rolling changes is critical. Since a well-constructed, healthy cluster will have at least one node of redundancy, it should be possible to upgrade clusters one node at a time, ensuring that all nodes are back online before applying the next changes.

Making multiple changes at once (e.g., restarting two nodes in a three-node replica set) would make a shard become momentarily read-only, blocking an entire application. This sort of rolling logic is a foundational feature of Ops/Cloud Manager and hard to replicate with off-the-shelf DevOps frameworks and custom scripts.

### Challenges

As anyone in the DevOps world can attest, automation is hard. Automation of complex, distributed systems without downtime is even harder. We want to guarantee that we can perform changes in a repeatable way that is robust and makes it easy to keep components on the latest, most stable, and secure versions. The following are some particular challenges when managing MongoDB deployments.

### Changing states

MongoDB clusters are stateful. Primary and secondary nodes have particular roles which can change over time. They also have to persist and sync data. We can’t just launch a replacement image into a replica set – that would trigger an initial sync – impacting both the sync source and the network. We may need to respect data protection regulations, by tightly controlling the location and flow of data.
Security

We also need to manage security in a cluster. This may include creating appropriate x.509 certificates to authenticate cluster membership, retrieving master keys from a KMIP server, or connecting to Kerberos to authenticate new connections. We may need to set up TLS in the right order so that we don’t lock ourselves out of the cluster, preventing the completion of changes.

Size and complexity

Managing large MongoDB deployments manually, and configuring each node manually, can introduce a number of risks. These include longer downtime (and possible initial sync due to insufficient oplog), or the introduction of configuration errors. A mature well-tested automated system should remove or at least greatly reduce these risks.

Fault tolerance

Since MongoDB is already self-healing, and once a cluster is correctly configured, it shouldn’t require much intervention (manual or automated) until such time an upgrade or configuration change is required or hardware needs to be replaced.

System architects can choose the level of fault tolerance suited to their business case. For example, a replica set with five data-bearing members (rather than the usual three) can function with up to two members down.

Framework choice

The technologies and stacks of DevOps move fast. Many of the frameworks and methodologies discussed in this chapter may have changed by the time you read this book. When choosing a framework, there are a few issues to consider.

Reuse

We want to loosely couple our automation components, reusing whatever is already available in the products we use (i.e., Enterprise-only tools like Ops Manager and the Kubernetes Operator), and avoid coding our own configuration/scripts from scratch.
Avoid lock-in

In general, we don’t want to overcommit to a particular framework. Even popular solutions may stop receiving support, and we could suddenly find that we can’t easily upgrade to the newest, most secure operating system because our tooling just doesn’t support it.

We don’t want to lock ourselves into our current infrastructure, be that raw metal, OpenStack, OpenShift, hyperconvergence, and so on.

We want to maintain the option to move painlessly to a cloud provider, or even better, to a multi-cloud configuration where we don’t depend on any one provider. This was particularly painful during the Covid-19 epidemic where those reliant on Azure found that there was simply no capacity available to spin up new compute instances.

Multi-cloud topologies

If you plan to deploy to the cloud and maximize high availability, using multiple cloud providers is one option. You could have a European shard with a node in GCP europe-west3, Azure Germany West Central, and AWS eu-central-1. This would place all nodes in or near Frankfurt, Germany, to reduce latency, but we need to be careful about the costs of bandwidth between providers.

The complexity of multi-cloud DevOps is also higher. Some tools like Terraform include platform-specific modules which can seamlessly deploy the low-level infrastructure. Other tools like Ansible may require custom playbooks for certain cloud providers who offer customized host images, networking stacks, or firewalls.

Base images

Each cloud provider offers a somewhat tailored (and optimized) base image for each Linux flavor. It’s tempting to use each provider’s image as a foundation, since they come with preconfigured optimizations. To avoid lock-in, you could instead build your own customized images with HashiCorp’s Packer (https://packer.io/) or a similar infrastructure-as-code (IaC) tool.

In many cases, it’s better to use Terraform to manage any cloud-specific aspects of the environment (e.g., security groups, VPNs, auto-scaling), although there are cloud-specific modules for configuration management tools like Ansible.
Virtual machines

Virtual machines (VM) now form the cornerstone of both on-prem and cloud computing stacks. By breaking up very large hosts into multiple virtual machines, MongoDB deployments can be deployed more efficiently but kept separate. This means that instability in one VM, or one node, should not adversely affect the stability of any others even on the same VM host.

Figure 7-1 shows a single shard cluster spread over eight virtual machines deployed on three physical VM hosts. Any one physical host can fail without affecting the cluster’s operation.

Figure 7-1. Virtual machines provide resource separation

Most VM solutions support the concepts of *affinity* and *anti-affinity*. Affinity rules require that a group of VMs should be deployed on the same VM host. By contrast, anti-affinity rules can be used to prevent VMs for a replica set (like 1b, 2b, and 3a) to be deployed on the same host.

Co-location

The issue of “noisy neighbors” stealing CPU and memory resources from other VMs on the same host can still be a problem in the cloud. On-prem, this can normally be configured away by preventing VMs from “borrowing” resources from other VMs.
This lets sysadmins provide consistent resources and guaranteed baseline performance to all processes in a deployment.

**Other benefits**

VMs also give a number of other benefits, including the ability to more easily resize VMs with more computing cores, additional memory resources, and taking snapshots of storage volumes.

**Containers**

Computing containers are another clever way of separating, delivering, and running processes without negatively interacting with others. For MongoDB deployments, a container would include the data files and the config files necessary for a node to start up and reach out to its other cluster members. MongoDB binaries and other required packages can be shared among all the containers on the same host, so they don’t necessarily require additional storage space.

Like VMs, containers keep memory, CPU, and I/O resources separate and limited, making it easier to run multiple MongoDB nodes on the same physical host without side effects.

Using cgroups in Linux, containers can be configured with resource ceilings, meaning that a config server node can use (for example) only 2GB of RAM, but a shard member on the same host might be allowed up to 200GB.

**Compared to VMs**

VMs require an entire operating system to achieve the same level of resource allocation, but of course the overhead of running an entire OS inside each VM client wastes a lot of valuable system host RAM.

In Figure 7-2 we see an identical cluster topology to Figure 7-1, except that we have only three VMs. By using containers instead, we can use the host’s memory and storage resources more efficiently.
Containers are more lightweight, but often more complex to deploy. They also have tooling better suited to developers, good for organizations using a continuous delivery methodology. Using containers also makes testing more accurate since exact containers can be tested and deployed together.

Virtual machines are likely to remain the base level of system administration for at least the near future, as they are the foundation of cloud computing.

**Docker**

While alternatives do exist (VirtualBox by Oracle, Vagrant, and Wox, among others), Docker is currently the most common and easy-to-use container.

Docker provides strong isolation in terms of memory and networking. Ports from these containers need to be explicitly exposed to allow communication to the outside world.

Docker images are often deployed across multiple hosts and data centers using a container orchestration system such as Kubernetes.

---

**Figure 7-2. Containers provide separation without VM overhead**

Containers are more lightweight, but often more complex to deploy. They also have tooling better suited to developers, good for organizations using a *continuous delivery* methodology. Using containers also makes testing more accurate since exact containers can be tested and deployed together.

Virtual machines are likely to remain the base level of system administration for at least the near future, as they are the foundation of cloud computing.

**Docker**

While alternatives do exist (VirtualBox by Oracle, Vagrant, and Wox, among others), Docker is currently the most common and easy-to-use container.

Docker provides strong isolation in terms of memory and networking. Ports from these containers need to be explicitly exposed to allow communication to the outside world.

Docker images are often deployed across multiple hosts and data centers using a container orchestration system such as Kubernetes.
Cloud-based container services

Amazon Elastic Container Service (ECS) is a highly scalable container management service for managing Docker containers on a cluster. Amazon Elastic Container Service for Kubernetes (EKS) extends this by adding a standards-based Kubernetes management system for an additional cost.

Other cloud providers offer similar solutions with Azure Container Service (AKS) and Google Kubernetes Engine (GKE).

Hyperconverged infrastructure

Another relatively new technology (as of this book’s writing) is hyperconverged infrastructure (HCI). One of the goals of this approach is to allow organizations to build up two or more data centers and seamlessly maintain redundant copies of all computing resources. Should a failure occur suddenly in one data center, the system will automatically launch replacement instances of all services in the remaining data centers. This requires that data from one data center is instantly and transparently replicated to the other data centers through a storage area network at all times.

Organizations may be tempted to rely on the hyperconvergence framework for maintaining database uptime service-level objectives (SLOs), but mixing MongoDB’s highly available replication system and hyperconvergence is both problematic and overly redundant.

MongoDB already includes all the data center and network state awareness but is designed specifically for the control of structured documents. Applications can control their level of redundancy through write and read concerns, giving much more fine-grained control than HCI.

In production, MongoDB can be installed inside hyperconverged infrastructures, but the built-in storage replication should be disabled. MongoDB will handle replication, and the additional load and latency from HCI provides no additional benefit.

Also, if the MongoDB cluster is correctly configured for fault tolerance (Chapter 2) and global topologies (Chapter 6), then losing a single data center should not impact an application. Therefore, booting up a new replacement node immediately after a failure is just not necessary and could add extra load if that node had to initial sync.
Production requirements

The Production Notes in your version of the MongoDB documentation is the authoritative reference for how best to configure your server to optimize MongoDB stability and performance.

These notes cover things like what permissions are required on the filesystem, how to set up the journal and logs paths to avoid drive I/O conflicts, network security, WiredTiger cache settings and compression, memory configuration such including NUMA and no transparent HugePages, using XFS and noatime for the filesystem, correct network settings (compression, TCP keepalives, etc.), optimizing VM (memory swapping), and how to make SELinux work.

Most of these settings are dependent on the exact operating system and version, the hardware available, and any virtualized environments, so there are no official ready-made virtual images or scripts to prepare the environment. It’s up to the system administrator to make sure the servers are set up correctly for the MongoDB nodes that will run there.

Standardized servers

Most organizations choose to run standardized servers for their entire cluster (config servers, mongos instances, and shard nodes). If not exactly same specs in terms of memory and storage devices, hosts should have at least the exact same operating system version and security settings.

Any number of automation and orchestration tools can build, prepare, and upgrade servers in a standard, repeatable way to easily replace or add new servers as the database needs to grow over time, or hardware fails and needs to be replaced.

Operating system upgrades should also be made in a prepared, controlled manner and ideally require little or no manual steps to complete.

This way, configuration is at least consistent across the entire cluster, and we can rule out server-level misconfiguration when troubleshooting any changes in behavior or performance of the MongoDB components running on top.
Operation system choices

While MongoDB has been developed to run on a number of popular architectures and OSes in high-performance production-ready configurations, there are just a few which are widely used in production. These include enterprise Linux (Red Hat, CentOS, Oracle, SUSE, Ubuntu Server LTS) and Windows.

Cloud variants

For those running on self-managed cloud hosts, systems like Amazon Linux are also popular since these images already have optimized configurations for the idiosyncratic virtual storage and network implementations. Unfortunately, they are often forks of older versions of Linux, and the location of configuration files and packages included can diverge significantly from standards. This difference makes it even harder to manage a mixed on-prem and cloud environment.

Since you can find official images of most enterprise Linux flavors for cloud providers, it might be best to keep the same OS, but add cloud-specific configuration tasks which configure cloud storage devices different to local hardware RAIDs, for example.

Older RHEL versions

Some older versions of Enterprise Linux can include known security or stability issues. In general, you should plan to upgrade to the latest downstream releases from Red Hat within a year of release.

For example, if you are using native LDAP via the operation system's OpenLDAP library, it's strongly recommended to use at least RHEL 7.5. At this release, OpenLDAP switched to an OpenSSL implementation which brings important thread safety.

MongoDB only provides official packages for Enterprise or LTS (long-term support) releases. Other OS releases do not have sufficiently long support lifecycles.

Virtual machines

If you need to run multiple smaller MongoDB nodes (such as for mongos or config server nodes), it can be tempting to co-locate these on a single server, or VM running on different ports. However, the complexity when troubleshooting such systems and the difficulty ensuring that both processes share system resources in an optimal way is much harder.
It’s far better to first set up a VM host and deploy multiple VMs with fixed resources (i.e., with memory ballooning disabled).

**Kubernetes**

However, for some organizations, VMs are already too cumbersome to manage. Many enterprises have moved their production applications to Kubernetes-based infrastructures like Amazon EKS and OpenShift.

Historically, with the Kubernetes ecosystem it wasn’t possible to deploy pods with persistent storage which could survive certain pod moves or hardware failures, making it unsuitable for database deployments. With the addition of PersistentVolumes and StatefulSets, pods can now be moved without losing data and needing to perform a complete initial sync.

**Operators**

*Kubernetes Operators* can be used to package, deploy, and manage MongoDB as a Kubernetes-native application, essentially functioning as a custom controller.

MongoDB Enterprise licenses include access to a Kubernetes operator which together with Ops/Cloud Manager can deploy appropriate *Kubernetes Resource Containers* and then *MongoDB Database Resources* on top. In essence, you can properly automate MongoDB replica sets and sharded clusters inside Kubernetes environments.

---

**Tip** When running applications or microservices inside pods, always set the appName in the connection string. This application name will appear in the log files, and since the IP addresses can change as pods are redistributed, this unique name can help tracking logged issues back to the generating application.

---

**Tools**

*Infrastructure as code (IaC)* is the process of managing and provisioning hosts through machine-readable definition files or scripts that can be easily version controlled and tested. It is a way of encapsulating error-prone manual configuration procedures like run books into software code that makes actual deployment streamlined, reliable, and infinitely scalable. The most common IaC tool today is Terraform.
Continuous configuration automation (CCA) tools can be thought of as an extension of IaC frameworks, but also the provisioning and configuration of applications and services on top of the managed hosts.

Configuration management

We’ll quickly review some of the most popular configuration management and automation tools: Ansible, Puppet, and Chef. All three tools are open source, very widely used, well documented with great community contributions, but also offer optional paid Enterprise support. They can also work with tools like Terraform to provision the host infrastructure prior to configuration.

Other notable alternatives are SaltStack and Pulumi.

Ansible

Ansible (first released in 2012, and acquired by Red Hat, Inc. in 2015) is an automation platform that allows you to deploy your MongoDB installation and configuration in a simple, repeatable manner. Its paradigm is about configuring simple rules, grouped in fixed-order steps and templates, rather than coding scripts in a procedural language.

It uses SSH to connect from a workstation to multiple remote machines without requiring any special agents on the remote hosts, just its own SSH keys.

Ansible is simple to get started, with a wide user base and lots of examples online. It is also easy to keep changes in source control. However, it’s hard to test changes and may require many runs to debug complex tasks grouped into playbooks.

Puppet

Puppet (first released in 2005) is model-driven, statically checked, and was built with systems administrators in mind. It follows a client-server (or agent-master) architecture. A commercial version, Puppet Enterprise, is available through Puppet Labs.

The Puppet master compares the current state of the target machine against the machine-level configuration details and then sends instructions to the conversion layer for action. An agent checks regularly for changes and triggers updates.
A module called puppet/mongodb (maintained by Puppet Labs and the open source community) manages server installation and configuration of the MongoDB processes, as well as Ops Manager setup. This module has an advanced and complete feature set, including a mongodb_shard provider which allows definition of shards and their members. At the time of writing, it supports RHEL/CentOS 5/6/7 and Ubuntu 10 and 12, as well as Debian 6 and 7.

Puppet is a bit more complex than some other management tools (such as Chef and Ansible), but it has a lot of modules that can be used to resolve issues around database management.

**Chef**

Like Puppet, Chef (first released in 2009) compares host resources against a desired state before making changes on a Chef node. Unlike Ansible, it requires a Chef client to be installed on each remote host in order for that host to become a Chef node.

Configuration is written into Chef recipes which are grouped together as cookbooks for easier management. An open source cookbook, SC-MongoDB Cookbook, is the most advanced preexisting cookbook for Chef and is currently under active development.

**On-prem comparison**

Any of the three tools presented can be used to prepare dependencies and deploy complex MongoDB deployments. Table 7-2 compares some of the key differences which may help choosing between them based on your platform and infrastructure preferences.

---

1. [https://forge.puppet.com/puppet/mongodb](https://forge.puppet.com/puppet/mongodb)
2. [https://supermarket.chef.io/cookbooks/sc-mongodb](https://supermarket.chef.io/cookbooks/sc-mongodb)
**Table 7-2. A comparison of configuration management tools**

|                | Ansible                        | Puppet                        | Chef                          |
|----------------|--------------------------------|-------------------------------|-------------------------------|
| Architecture   | Pushes config (from single node) | Pull (Master-slave)           | Pull (Master-slave)           |
| Configuration paradigm | Declarative (Plays and Playbooks) | Procedural (Tasks and Plans)  | Procedural (Recipes and Cookbooks) |
| Update paradigm | Step-based                     | Idempotent and state-based    | Idempotent and state-based    |
| Remote requirements | SSH daemon only, no agent     | Master server + Puppet agent | Master server + Chef agent required |
| Security       | SSH key + user privileges     | Certificate                   | SSL and shared secrets        |
| Getting started | Easy                           | Harder                        | Harder                        |
| Complex tasks  | Hard to debug                  | Powerful                      | Possible                      |
| Platform       | Python/pip                     | Ruby/gem                     | Ruby/gem                     |
| Supported OSes (remote) | Master: Linux/BSD with Python Remote: any POSIX with Python, and Windows | Master: Enterprise Linux, Ubuntu, SUSE Remote: Linux, macOS, Windows, Solaris, and many others | Workstation: RHEL, Ubuntu, Windows, macOS Server: Enterprise Linux, Ubuntu Node/remote: AIX, Linux, BSD, Windows, macOS |
| Best suited    | Initial setup on top of prebuilt VM images or Docker | Regularly changing environments. Where compliance is important. | Regularly changing environments |

**Automation**

Often the goal of CM tools is to standardize and avoid human errors when deploying new hosts and software stacks. With some additional effort, this configuration code can be extended to tear down, move, and scale systems over thousands of nodes, running on different underlying hardware/cloud and on different operating systems.
With awareness of entire systems and dependencies between components, automation can be made to orchestrate changes in a consistent way, for example, to ensure that multiple nodes from the same replica set aren’t taken down at the same time.

**Provisioning and orchestration**

Before we can configure our MongoDB environments and deployments, we may first want to automate provisioning of the hosts. In some cases, we may also need to provision networking infrastructure, storage, and even load balancers.

For most deployments, Terraform is currently the winner in the *infrastructure-as-code* space both for provisioning reproducible on-prem as well as cloud-agnostic infrastructure.

**Terraform**

While Terraform can also perform configuration, it is currently better to use targeted configuration management tools like Ansible, Puppet, or Chef. Like most CM tools, all Terraform configuration files can be easily version controlled. Custom *Provider* plug-ins can be created in the *Go* programming language.

While possible to use in a free Community version, the maintainer HashiCorp sells an Enterprise version with support.

**Purpose**

Terraform monitors the state of the environment, and if anything is out of order or missing, it can automatically provide a replacement resource. This is fantastic for environments that require a very steady state.

**Cloud infrastructure**

Terraform can communicate via cloud provider APIs using standard API keys, in much the same way as on-prem master servers.
Official provider for Atlas

MongoDB Inc. in conjunction with HashiCorp has created an official provider that is designed for MongoDB’s Atlas fully hosted solution. Currently, there is no official plug-in for using Terraform with non-Atlas on-prem MongoDB deployments.

While it’s possible to create a custom or community provider yourself, this can be a very complex undertaking, to build and test a provider. As such, using Terraform alone for on-prem MongoDB deployments is not an ideal approach at this time.

Integration with Ansible

Terraform can call Ansible configuration (via Packer) to automate the building up of machine images with MongoDB and dependency packages, and configuration files at the OS level. Terraform can add special tags to your servers for Ansible to find and configure each one accordingly.

In the other direction, Ansible can also be hooked in to the workflow, calling Terraform to launch multiple computing instances based on images already built by Ansible. Often some final steps will be required post-provisioning to actually deploy cluster-specific MongoDB configuration and launch the instances.

CloudFormation

CloudFormation is one of the most popular ways to script the deployment of complex services on Amazon’s AWS cloud environment. It allows for both a declarative or programmatic approach to model and provision application environments on the cloud in a repeatable manner.

For example, you could define a CloudFormation script to deploy four EC2 computing hosts, one for the application and three for a MongoDB replica set. This script could ensure that all dependencies and MongoDB libraries are installed via yum, create a config and keyfile for the MongoDB nodes, and ensure that a replica set is initialized once all three nodes are online based on the hostnames allocated.

Unlike the other tools listed here, it is completely closed sourced and has no option for on-prem management.
Kubernetes operator

For Enterprise customers on any sort of self-managed infrastructure, MongoDB offers a custom-built operator using Kubernetes APIs to automate and manage MongoDB clusters. *MongoDB Enterprise Operator for Kubernetes* gives you full control over your MongoDB deployment from a single Kubernetes control plane. You can use the operator with any Kubernetes-compatible service such as OpenShift and Pivotal Container Service (PKS).

Kubernetes for MongoDB Atlas

For situations where you are using Kubernetes to deploy applications in fully managed platform-as-a-service (PaaS) contexts, the MongoDB Atlas Open Service Broker can automate your databases on MongoDB’s Atlas service on whichever cloud provider is best for your application.

Monitoring

Continuous and automatic monitoring of the health of the deployment, including the detection of anomalies and alerting, is critical to avoid multiple failures and application impact.

Evaluating failure

Depending on the topology and design choices in a MongoDB cluster, there will be different levels of failure which require different urgency to repair. If your cluster has been well designed, any single component failing should not cause application downtime, but you should always react as soon as possible to restore failed components.

Performance

Monitoring the performance and latency of a cluster is one aspect that tends to get less attention until something goes wrong. By understanding the *baseline performance* of your cluster, and its weekly and hourly trends, you should be able to detect abnormal behaviors before they manifest as instabilities or performance drops in your application.
Naturally over time, the size of the database will grow as new users and services are added. Monitoring baselines metrics over time allows capacity planning. If you can predict when the current computing resources will be insufficient, you can scale them vertically or horizontally before performance is impacted.

**FTDC**

One way to understand performance is by analyzing the full-time diagnostic data capture (FTDC) metrics generated by each mongod and mongos process and stored in a directory called diagnostic.data within MongoDB’s storage.dbPath.

The FTDC data includes metrics about server, replication, and collection status, as well as connection metrics. It also captures valuable host metrics like CPU and memory usage, I/O, and network usage. These metrics are designed to provide a sort of “black box flight recorder” for telemetry on a deployment to investigate when an issue has been observed.

There will be hundreds of metrics recorded every second, but the format is so compact that a week of metrics require only a few hundred megabytes of storage. There are numerous tools to extract and read this data, including an official open source parsing library written in Go ([https://github.com/mongodb/ftdc](https://github.com/mongodb/ftdc)) and an analysis tool called Keyhole ([https://github.com/simagix/keyhole](https://github.com/simagix/keyhole)).

**Connection capacity**

One example of important metrics captured in FTDC (via the serverStatus command) concerns the number of active vs. currently open (but not necessarily active) connections. As each connection takes memory, it’s important to set limits to avoid incoming connection storms particularly from microservices or IoT devices (see Chapter 8).

**SNMP**

Enterprise MongoDB also supports collecting database metrics directly from each node via SNMP on both Linux and Windows into other monitoring solutions. Ops Manager also supports SNMP traps to deliver alerts configured via its Alerts interface.
Alerting

Ideally, someone should be alerted whenever a component fails. To avoid false positives, we may want to wait a few minutes before triggering these alerts to skip short-lived network failures.

It also recommended to temporarily disable alerts during planned maintenance windows when we expect to be shutting down and restarting nodes such as for server maintenance or MongoDB version upgrades.

There are a number of popular generic alerting tools; some which can be configured to check the “liveness” of a MongoDB node by connecting to it on its port, but are unlikely to be able to detect the internal health of the node.

Some monitoring solutions can watch the CPU and memory usage of a process and trigger alerts when these exceed certain thresholds. However, these external views of a process may simply be observing spikes in workload due to legitimately high application traffic.

Prometheus

Prometheus is an open source, community-driven project, which collects, alerts, and stores metrics in a system-agnostic way, and uses its own internal database for storage. It utilizes Grafana (an open source dashboard) to display metrics in various time series views.

Since Prometheus is designed as a generic monitoring solution, there is no built-in mechanism for monitoring MongoDB, but such functionality is supported via plug-ins called exporters.

One such exporter is the open source mongodb_exporter, currently maintained at github.com/percona/mongodb_exporter which responds with metrics to Prometheus server calls.

Enterprise tools

In contrast, the official Enterprise monitoring solutions Ops Manager and Cloud Manager have been built by MongoDB Inc. itself to monitor large clusters, record metrics, and trigger alerts. They are not only able to connect to each node but also evaluate internal health to confirm that the node is still able to respond to requests.
Ops/Cloud Manager also records metrics about the load on the node for later analysis and visualization. These metrics are available via a comprehensive API for integration with other monitoring services like New Relic.

Alerts configured in Ops/Cloud Manager can be sent through your existing Alerting tools, via integrations with PagerDuty, Flowdock, HipChat, Opsgenie, Datadog, Slack, and others.

**Key takeaways**

From this chapter, the key concepts to remember are as follows:

- Provisioning, orchestration, and configuration tools are complex and evolving quickly, with no single standard for deploying MongoDB cluster infrastructure.

- Since MongoDB is stateful and self-healing, the usual tools for auto-scaling application servers can’t be easily repurposed.

- Terraform is a popular tool for on-prem and multi-cloud provisioning, with Ansible a good choice for configuration management.

- For Kubernetes infrastructures like OpenShift, an official Enterprise-only operator plug-in is available to provision and configure MongoDB clusters.

- It’s critical to properly monitor a cluster to understand baseline performance, both for setting alerts at the correct threshold and also to know when it’s time to scale the cluster.

- Enterprise tools like Ops/Cloud Manager are the best choice to automate and monitor large, complex clusters as well as to integrate with existing monitoring/alerting systems like Prometheus or New Relic or via SNMP.

- For best security and stability, you should not wait to upgrade to the latest minor release of MongoDB and OS minor releases.