

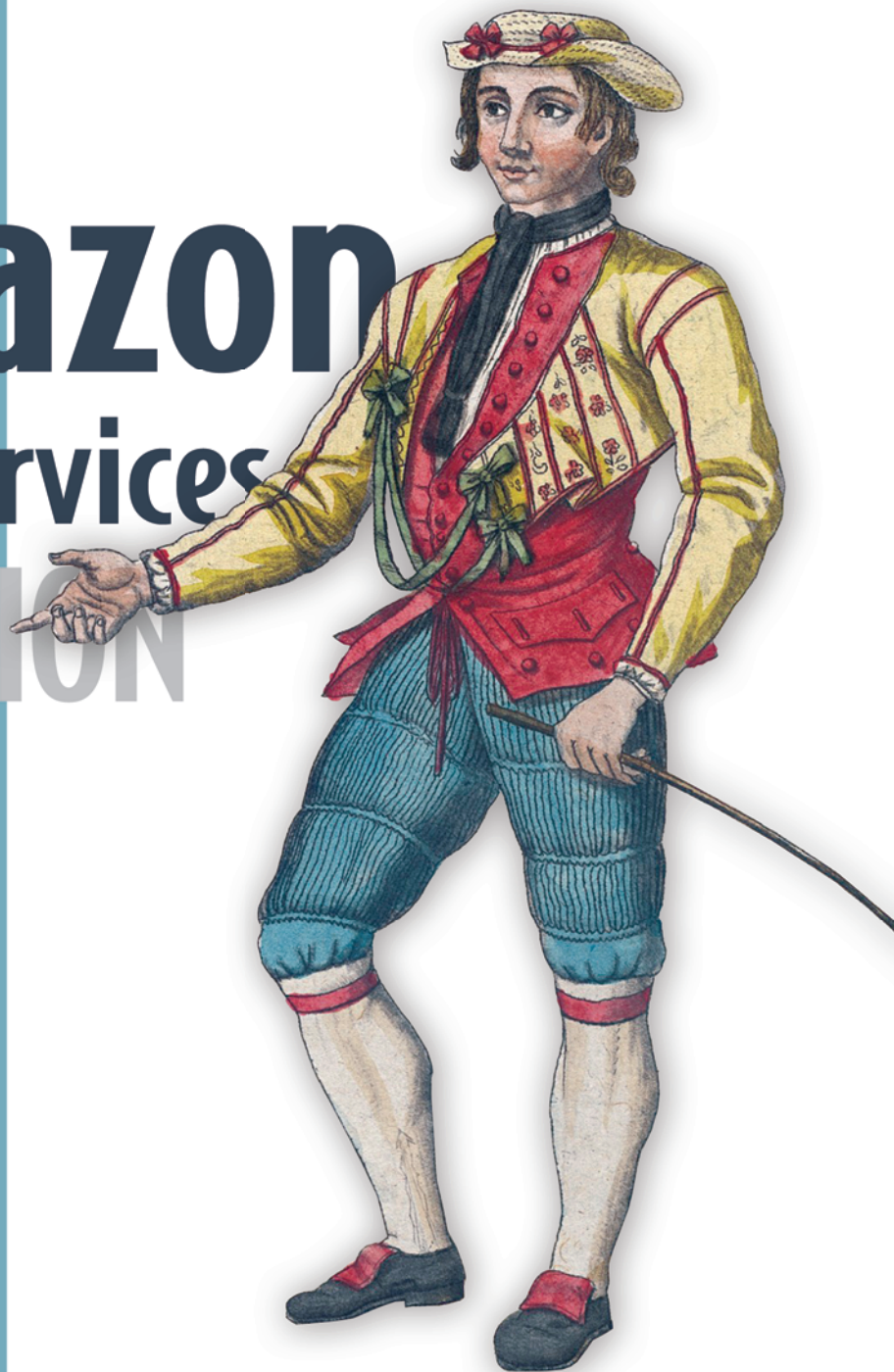
Amazon Web Services IN ACTION

SECOND EDITION

Michael Wittig
Andreas Wittig

Foreword by Ben Whaley

SAMPLE CHAPTER





*Amazon Web Services in Action,
Second Edition*

by Michael Wittig
and Andreas Wittig

Chapter 12

Copyright 2018 Manning Publications

brief contents

PART 1	GETTING STARTED	1
1	■ What is Amazon Web Services?	3
2	■ A simple example: WordPress in five minutes	36
PART 2	BUILDING VIRTUAL INFRASTRUCTURE CONSISTING OF COMPUTERS AND NETWORKING	57
3	■ Using virtual machines: EC2	59
4	■ Programming your infrastructure: The command-line, SDKs, and CloudFormation	102
5	■ Automating deployment: CloudFormation, Elastic Beanstalk, and OpsWorks	135
6	■ Securing your system: IAM, security groups, and VPC	165
7	■ Automating operational tasks with Lambda	199
PART 3	STORING DATA IN THE CLOUD	233
8	■ Storing your objects: S3 and Glacier	235
9	■ Storing data on hard drives: EBS and instance store	258

- 10 ■ Sharing data volumes between machines: EFS 274
- 11 ■ Using a relational database service: RDS 294
- 12 ■ Caching data in memory: Amazon ElastiCache 321
- 13 ■ Programming for the NoSQL database service: DynamoDB 349

PART 4 ARCHITECTING ON AWS.....381

- 14 ■ Achieving high availability: availability zones, auto-scaling, and CloudWatch 383
- 15 ■ Decoupling your infrastructure: Elastic Load Balancing and Simple Queue Service 413
- 16 ■ Designing for fault tolerance 431
- 17 ■ Scaling up and down: auto-scaling and CloudWatch 463

12

Caching data in memory: Amazon ElastiCache

This chapter covers

- Benefits of a caching layer between your application and data store
- Terminology like cache cluster, node, shard, replication group, and node group
- Using/Operating an in-memory key-value store
- Performance tweaking and monitoring ElastiCache clusters

Imagine a relational database being used for a popular mobile game where players' scores and ranks are updated and read frequently. The read and write pressure to the database will be extremely high, especially when ranking scores across millions of players. Mitigating that pressure by scaling the database may help with load, but not necessarily the latency or cost. Also, relational databases tend to be more expensive than caching data stores.

A proven solution used by many gaming companies is leveraging an in-memory data store such as Redis for both caching and ranking player and game metadata.

Instead of reading and sorting the leaderboard directly from the relational database, they store an in-memory game leaderboard in Redis, commonly using a Redis Sorted Set, which will sort the data automatically when it's inserted based on the score parameter. The score value may consist of the actual player ranking or player score in the game.

Because the data resides in memory and does not require heavy computation to sort, retrieving the information is incredibly fast, leaving little reason to query from a relational database. In addition, any other game and player metadata such as player profile, game level information, and so on that requires heavy reads can also be cached within this in-memory layer, freeing the database from heavy read traffic.

In this solution, both the relational database and in-memory layer will store updates to the leaderboard: one will serve as the primary database and the other as the working and fast processing layer. For caching data, they may employ a variety of caching techniques to keep the data that's cached fresh, which we'll review later. Figure 12.1 shows where the cache sits between your application and the database.

A cache comes with multiple benefits:

- The read traffic can be served from the caching layer, which frees resources on your data store, for example for write requests.
- It speeds up your application because the caching layer responds more quickly than your data store.
- You can downsize your data store, which can be more expensive than the caching layer.

Most caching layers reside in-memory and that's why they are so fast. The downside is that you can lose the cached data at any time because of a hardware defect or a restart. Always keep a copy of your data in a primary data store with disk durability, like the relational database in the mobile game example. Alternatively, Redis has optional failover support. In the event of a node failure, a replica node will be elected to be the new primary and will already have a copy of the data.

Depending on your caching strategy, you can either populate the cache in real-time or on-demand. In the mobile game example, on-demand means that if the leaderboard is not in the cache, the application asks the relational database and puts the result into the cache. Any subsequent request to the cache will result in a cache hit, meaning the data is found. This will be true until the duration of the TTL (time to live) value on the cached value expires. This strategy is called *lazy-loading* the data from the primary data store. Additionally, we could have a cron job running in the background that queries the leaderboard from the relational database every minute and puts the result in the cache to populate the cache in advance.

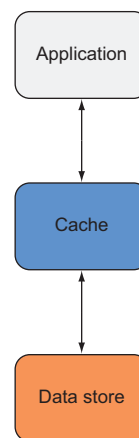


Figure 12.1 Cache sits between the application and the database

The lazy-loading strategy (getting data on demand) is implemented like this:

- 1 The application writes data to the data store.
- 2 If the application wants to read the data, at a later time it makes a request to the caching layer.
- 3 The caching layer does not contain the data. The application reads from the data store directly and puts the read value into the cache, and also returns the value to the client.
- 4 Later, if the application wants to read the data again, it makes a request to the caching layer and finds the value.

This strategy comes with a problem. What if the data is changed while it is in the cache? The cache will still contain the old value. That's why setting an appropriate TTL value is critical to ensure cache validity. Let's say you apply a TTL of 5 minutes to your cached data: this means you accept that the data could be up to 5 minutes out of sync with your primary database. Understanding the frequency of change for the underlying data and the effects out-of-sync data will have on the user experience is the first step of identifying the appropriate TTL value to apply. A common mistake some developers make is assuming that a few seconds of a cache TTL means that having a cache is not worthwhile. Remember that within those few seconds, millions of requests can be eliminated from your back end, speeding up your application and reducing the back-end database pressure. Performance testing your application with and without your cache, along with various caching approaches, will help fine-tune your implementation. In summary, the shorter the TTL, the more load you have on your underlying data store. The higher the TTL, the more out of sync the data gets.

The write-through strategy (caching data up front) is implemented differently to tackle the synchronization issue:

- 1 The application writes data to the data store and the cache (or the cache is filled asynchronously, for example in a cron job, AWS Lambda function, or the application).
- 2 If the application wants to read the data at a later time, it makes a request to the caching layer, which contains the data.
- 3 The value is returned to the client.

This strategy also comes with a problem. What if the cache is not big enough to contain all your data? Caches are in-memory and your data store's disk capacity is usually larger than your cache's memory capacity. When your cache reaches the available memory, it will evict data, or stop accepting new data. In both situations, the application stops working. In the gaming app, the global leaderboard will always fit into the cache. Imagine that a leaderboard is 4 KB in size and the cache has a capacity of 1 GB (1,048,576 KB). But what about team leaderboards? You can only store 262,144 (1,048,576 / 4) leaderboards, so if you have more teams than that, you will run into an capacity issue.

Figure 12.2 compares the two caching strategies. When evicting data, the cache needs to decide which data it should delete. One popular strategy is to evict the least recently used (LRU) data. This means that cached data must contain meta information about the time when it was last accessed. In case of an LRU eviction, the data with the oldest timestamp is chosen for eviction.

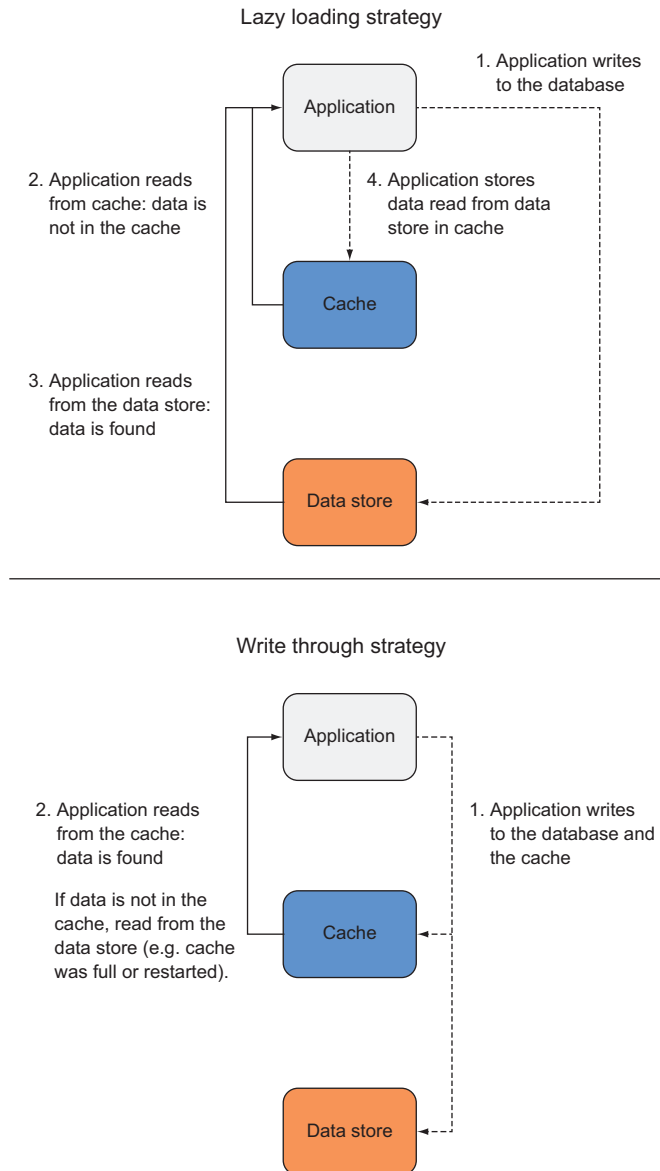


Figure 12.2 Comparing the lazy-loading and write-through caching strategies

Caches are usually implemented using key-value stores. Key-value stores don't support sophisticated query languages such as SQL. They support retrieving data based on a key, usually a string, or specialized commands, for example to extract sorted data efficiently.

Imagine that in your relational database you have a player table for your mobile game. One of the most common queries is `SELECT id, nickname FROM player ORDER BY score DESC LIMIT 10` to retrieve the top ten players. Luckily, the game is very popular. But this comes with a technical challenge. If many players look at the leaderboard, the database becomes very busy, which causes high latency or even time-outs. You have to come up with a plan to reduce the load on the database. As you already learned, caching can help. What technique should you employ for caching? You have a few options.

One approach you can take with Redis is to store the result of your SQL query as a String value and the SQL statement as your key name. Instead of using the whole SQL query as the key, you can hash the string with a hash function like md5 or sha256 to optimize storage and bandwidth ❶ as shown in figure 12.3. Before the application sends the query to the database, it takes the SQL query as the key to ask the caching layer for data ❷. If the cache does not contain data for the key ❸, the SQL query is sent to the relational database ❹. The result ❺ is then stored in the cache using the SQL query as the key ❻. The next time the application wants to perform the query, it asks the caching layer ❼, which now contains the cached table ❽.

To implement caching, you only need to know the key of the cached item. This can be an SQL query, a filename, a URL, or a user ID. You take the key and ask the cache for a result. If no result is found, you make a second call to the underlying data store, which knows the truth.

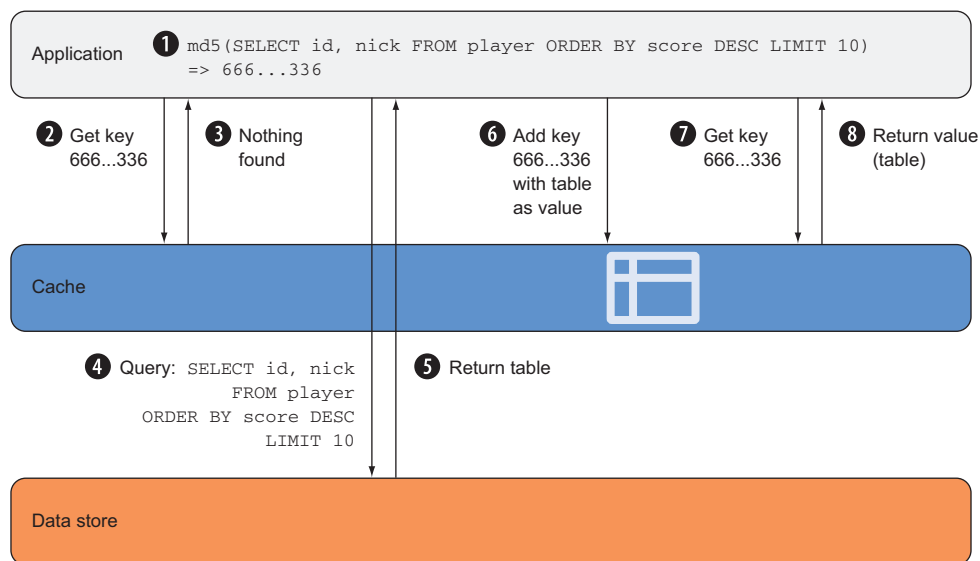


Figure 12.3 SQL caching layer implementation

With Redis, you also have the option of storing the data in other data structures such as a Redis SortedSet. If the data is stored in a Redis SortedSet, retrieving the ranked data will be very efficient. You could simply store players and their scores and sort by the score. An equivalent SQL command would be:

```
ZREVRANGE "player-scores" 0 9
```

This would return the ten players in a SortedSet named “player-scores” ordered from highest to lowest.

Examples are 100% covered by the Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don’t run the examples longer than a few days, you won’t pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you’ll clean up your account at the end.

The two most popular implementations of in-memory key-value stores are Memcached and Redis. Amazon ElastiCache offers both options. Table 12.1 compares their features.

Table 12.1 Comparing Memcached and Redis features

	Memcached	Redis
Data types	simple	complex
Data manipulation commands	12	125
Server-side scripting	no	yes (Lua)
Transactions	no	yes
Multi-threaded	yes	no

Amazon ElastiCache offers Memcached and Redis clusters as a service. Therefore, AWS covers the following aspects for you:

- *Installation*—AWS installs the software for you and has enhanced the underlying engines.
- *Administration*—AWS administers Memcached/Redis for you and provides ways to configure your cluster through parameter groups. AWS also detects and automates failovers (Redis only).
- *Monitoring*—AWS publishes metrics to CloudWatch for you.
- *Patching*—AWS performs security upgrades in a customizable time window.

- *Backups*—AWS optionally backs up your data in a customizable time window (Redis only).
- *Replication*—AWS optionally sets up replication (Redis only).

Next, you will learn how to create an in-memory cluster with ElastiCache that you will later use as an in-memory cache for an application.

12.1 Creating a cache cluster

In this chapter, we focus on the Redis engine because it's more flexible. You can choose which engine to use based on the features that we compared in the previous section. If there are significant differences to Memcached, we will highlight them.

12.1.1 Minimal CloudFormation template

You can create an ElastiCache cluster using the Management Console, the CLI, or CloudFormation. You will use CloudFormation in this chapter to manage your cluster. The resource type of an ElastiCache cluster is `AWS::ElastiCache::CacheCluster`. The required properties are:

- *Engine*—Either `redis` or `memcached`
- *CacheNodeType*—Similar to the EC2 instance type, for example `cache.t2.micro`
- *NumCacheNodes*—1 for a single-node cluster
- *CacheSubnetGroupName*—You reference subnets of a VPC using a dedicated resource called a subnet group
- *VpcSecurityGroupIds*—The security groups you want to attach to the cluster

A minimal CloudFormation template is shown in listing 12.1.

Listing 12.1 Minimal CloudFormation template of an ElastiCache Redis single-node cluster

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 12 (minimal)'
Parameters:
  VPC:
    Type: 'AWS::EC2::VPC::Id'
  SubnetA:
    Type: 'AWS::EC2::Subnet::Id'
  SubnetB:
    Type: 'AWS::EC2::Subnet::Id'
  KeyName:
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
Resources:
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
```

← Defines VPC and subnets as parameters

← The security group to manage which traffic is allowed to enter/leave the cluster

```

VpcId: !Ref VPC
SecurityGroupIngress:
- IpProtocol: tcp
  FromPort: 6379
  ToPort: 6379
  CidrIp: '0.0.0.0/0'
CacheSubnetGroup:
  Type: 'AWS::ElastiCache::SubnetGroup'
  Properties:
    Description: cache
    SubnetIds:
    - Ref: SubnetA
    - Ref: SubnetB
Cache:
  Type: 'AWS::ElastiCache::CacheCluster'
  Properties:
    CacheNodeType: 'cache.t2.micro'
    CacheSubnetGroupName: !Ref CacheSubnetGroup
    Engine: redis
    NumCacheNodes: 1
    VpcSecurityGroupIds:
    - !Ref CacheSecurityGroup

```

Redis listens on port 6379. This allows access from all IP addresses, but since the cluster only has private IP addresses, access is only possible from inside the VPC. You will improve this in section 12.3.

Subnets are defined within a subnet group (same approach is used in RDS).

List of subnets that can be used by the cluster

cache.t2.micro comes with 0.555 GiB memory and is part of the Free Tier.

redis or memcached

1 for a single-node cluster

The resource to define the Redis cluster.

As already mentioned, ElastiCache nodes in a cluster only have private IP addresses. Therefore, you can't connect to a node directly over the internet. The same is true for other resources as EC2 instances or RDS instances. To test the Redis cluster, you can create an EC2 instance in the same VPC as the cluster. From the EC2 instance, you can then connect to the private IP address of the cluster.

12.1.2 Test the Redis cluster

To test Redis, add the following resources to the minimal CloudFormation template:

```

Resources:
# [...]
VMSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'instance'
    SecurityGroupIngress:
    - IpProtocol: tcp
      FromPort: 22
      ToPort: 22
      CidrIp: '0.0.0.0/0'
    VpcId: !Ref VPC
VMInstance:
  Type: 'AWS::EC2::Instance'
  Properties:
    ImageId: 'ami-6057e21a'
    InstanceType: 't2.micro'
    KeyName: !Ref KeyName
    NetworkInterfaces:
    - AssociatePublicIpAddress: true

```

Security group to allow SSH access

Virtual machine used to connect to your Redis cluster

```

DeleteOnTermination: true
DeviceIndex: 0
GroupSet:
- !Ref VMSecurityGroup
SubnetId: !Ref SubnetA
Outputs:
  VMInstanceIPAddress:
    Value: !GetAtt 'VMInstance.PublicIp'
    Description: 'EC2 Instance public IP address
    (connect via SSH as user ec2-user)'
  CacheAddress:
    Value: !GetAtt 'Cache.RedisEndpoint.Address'
    Description: 'Redis DNS name (resolves to a private
    IP address)'

```

Public IP address of virtual machine

DNS name of Redis cluster node (resolves to a private IP address)

The minimal CloudFormation template is now complete. Create a stack based on the template to create all the resources in your AWS account using the Management Console: <http://mng.bz/Cp44>. You have to fill in four parameters when creating the stack:

- **KeyName**—If you’ve been following along with our book in order, you should have created a key pair with the name `mykey`, which is selected by default.
- **SubnetA**—You should have at least two options here; select the first one.
- **SubnetB**—You should have at least two options here; select the second one.
- **VPC**—You should only have one possible VPC here—your default VPC. Select it.

You can find the full code for the template at `/chapter12/minimal.yaml` in the book’s code folder.

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at <https://github.com/AWSinAction/code2/archive/master.zip>. The file we’re talking about is located at `chapter12/minimal.yaml`. On S3, the same file is located at <http://mng.bz/qJ8g>.

Once the stack status changes to `CREATE_COMPLETE` in the CloudWatch Management Console, select the stack and click on the Outputs tab. You can now start to test the Redis cluster. Open an SSH connection to the EC2 instance, and then you can use the Redis CLI to interact with the Redis cluster node.

```

Install the Redis CLI. | $ ssh -i mykey.pem ec2-user@$VMInstanceIPAddress
                        | $ sudo yum -y install --enablerepo=epel redis
                        | $ redis-cli -h $CacheAddress
Store the string value under the key key1. | > SET key1 value1
                                           | OK

```

Connect to the EC2 instance, replace `$VMInstanceIPAddress` with the output from the CloudFormation stack.

Connect to the Redis cluster node, replace `$CacheAddress` with the output from the CloudFormation stack.

```

> GET key1           ← Retrieve the value for key key1.
"value1"
> GET key2           ← If a key does not exist, you
(nil)                get an empty response.
> SET key3 value3 EX 5  ← Store the string ttl under the key key3
OK                    and expire the key after 5 seconds.
> GET key3
"value3"
> GET key3           ← After 5 seconds, key3
(nil)                no longer exists.
> quit

```

Within 5 seconds, get the key key3.

Quit the Redis CLI.

You've successfully connected to a Redis cluster node, stored some keys, retrieved some keys, and used Redis's time-to-live functionality. With this knowledge, you could start to implement a caching layer in your own application. But as always, there are more options to discover. Delete the CloudFormation stack you created to avoid unwanted costs. Then, continue with the next section to learn more about advanced deployment options with more than one node to achieve high availability or sharding.

12.2 Cache deployment options

Which deployment option you should choose is influenced by four factors:

- 1 *Engine*—Memcached or Redis
- 2 *Backup/Restore*—Is it possible to back up or restore the data from the cache?
- 3 *Replication*—If a single node fails, is the data still available?
- 4 *Sharding*—If the data does not fit on a single node, can you add nodes to increase capacity?

Table 12.2 compares the deployment options for the two available engines.

Table 12.2 Comparing ElastiCache deployment options

	Memcached	Redis: single node	Redis: cluster mode disabled	Redis: cluster mode enabled
Backup/Restore	no	yes	yes	yes
Replication	no	no	yes	yes
Sharding	yes	no	no	yes

Let's look at deployment options in more detail.

12.2.1 Memcached: cluster

An Amazon ElastiCache for a Memcached cluster consists of 1-20 nodes. Sharding is implemented by the Memcached client, typically utilizing a consistent hashing algorithm which arranges keys into partitions in a ring distributed across the nodes. The client essentially decides which keys belong to which nodes and directs the requests to those partitions. Each node stores a unique portion of the key-space in-memory. If a

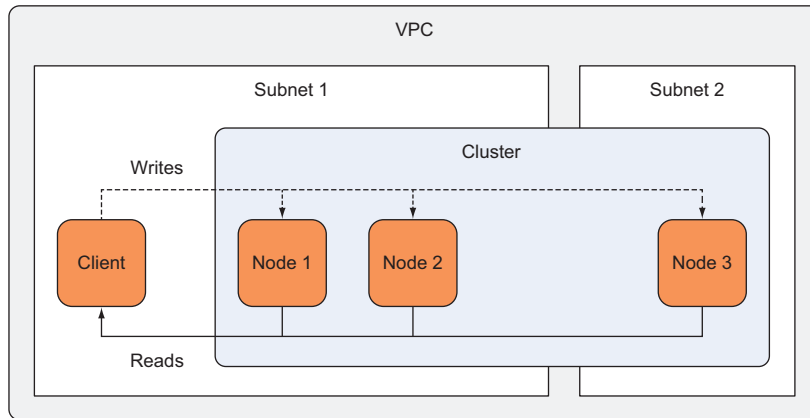


Figure 12.4 Memcached deployment option: cluster

node fails, the node is replaced but the data is lost. You can not back up the data in Memcached. Figure 12.4 shows a Memcached cluster deployment.

You can use a Memcached cluster if your application requires a simple in-memory store and can tolerate the loss of a node and its data. The SQL cache example in the beginning of this chapter could be implemented using Memcached. Since the data is always available in the relational database, you can tolerate a node loss, and you only need simple commands (GET, SET) to implement the query cache.

12.2.2 Redis: Single-node cluster

An ElastiCache for a Redis single-node cluster always consists of one node. Sharding and high availability are not possible with a single node. But Redis supports the creation of backups, and also allows you to restore those backups. Figure 12.5 shows a Redis single-node cluster. Remember that a VPC is a way to define a private network on AWS. A subnet is a way to separate concerns inside the VPC. Cluster nodes always run in a single subnet. The client communicates with the Redis cluster node to get data and write data to the cache.

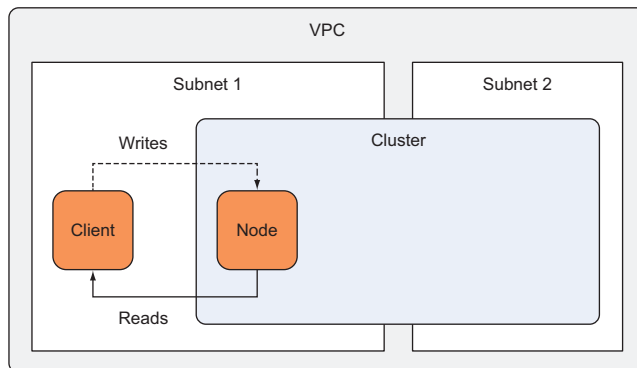


Figure 12.5 Redis deployment option: single-node cluster

A single node adds a single point of failure (SPOF) to your system. This is probably something you want to avoid for business-critical production systems.

12.2.3 Redis: Cluster with cluster mode disabled

Things become more complicated now, because ElastiCache uses two terminologies. We've been using the terms cluster/node/shard so far, and the graphical Management Console also uses these terms. But the API, the CLI, and CloudFormation use a different terminology: replication group/node/node group. We prefer the cluster/node/shard terminology, but in figures 12.6 and 12.7 we've added the replication group/node/node group terminology in parentheses.

A Redis cluster with cluster mode disabled supports backups and data replication, but no sharding. This means there is only one shard consisting of one primary and up to five replica nodes.

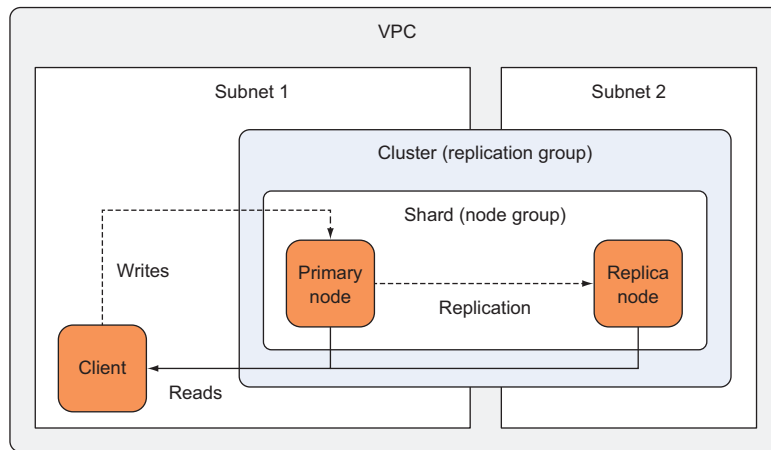


Figure 12.6 Redis deployment option: cluster with cluster mode disabled

You can use a Redis cluster with cluster mode disabled when you need data replication and all your cached data fits into the memory of a single node. Imagine that your cached data set is 4 GB in size. If your cache has at least 4 GB of memory, the data fits into the cache and you don't need sharding.

12.2.4 Redis: Cluster with cluster mode enabled

A Redis cluster with cluster mode enabled supports backups, data replication, and sharding. You can have up to 15 shards per cluster. Each shard consists of one primary and up to five replica nodes. The largest cluster size therefore is 90 nodes (15 primaries + (15 * 5 replicas)).

You can use a Redis cluster with cluster mode enabled when you need data replication and your data is too large to fit into the memory of a single node. Imagine that your cached data is 22 GB in size. Each cache node has a capacity of 4 GB of memory.

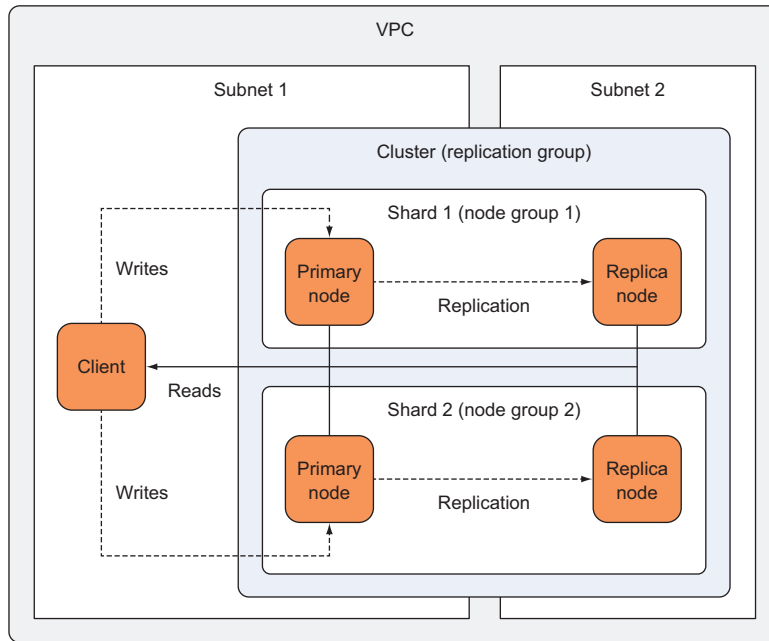


Figure 12.7 Redis deployment option: cluster with cluster mode enabled

Therefore, you will need six shards to get a total capacity of 24 GB of memory. ElastiCache provides up to 437 GB of memory per node, which totals to a maximum cluster capacity of 6.5 TB (15 * 437 GB).

Additional benefits of enabling cluster mode

With cluster mode enabled, failover speed is much faster, as no DNS is involved. Clients are provided a single configuration endpoint to discover changes to the cluster topology, including newly elected primaries. With cluster mode disabled, AWS provides a single primary endpoint and in the event of a failover, AWS does a DNS swap on that endpoint to one of the available replicas. It may take ~1–1.5min before the application is able to reach the cluster after a failure, whereas with cluster mode enabled, the election takes less than 30s.

More shards enable more read/write performance. If you start with one shard and add a second shard, each shard now only has to deal with 50% of the requests (assuming an even distribution).

As you add nodes, your blast radius decreases. For example, if you have five shards and experience a failover, only 20% of your data is affected. This means you can't write to this portion of the key space until the failover process completes (~15–30s), but you can still read from the cluster, given you have a replica available. With cluster mode disabled, 100% of your data is affected, as a single node consists of your entire key space. You can read from the cluster but can't write until the DNS swap has completed.

You are now equipped to select the right engine and the deployment option for your use case. In the next section, you will take a closer look at the security aspects of ElastiCache to control access to your cache cluster.

12.3 Controlling cache access

Access control is very similar to the way it works with RDS (see section 11.4). The only difference is, that cache engines come with very limited features to control access to the data itself. The following paragraph summarizes the most important aspects of access control.

ElastiCache is protected by four layers:

- *Identity and Access Management (IAM)*: Controls which IAM user/group/role is allowed administer an ElastiCache cluster.
- *Security Groups*: Restricts incoming and outgoing traffic to ElastiCache nodes.
- *Cache Engine*: Redis has the AUTH command, Memcached does not handle authentication. Neither engine supports authorization.
- *Encryption*: At rest and in transit.

SECURITY WARNING It's important to understand that you don't control access to the cache nodes using IAM. Once the nodes are created, Security Groups control the access.

12.3.1 Controlling access to the configuration

Access to the ElastiCache service is controlled with the help of the IAM service. The IAM service is responsible for controlling access to actions like creating, updating, and deleting a cache cluster. IAM doesn't manage access inside the cache; that's the job of the cache engine. An IAM policy defines the configuration and management actions a user, group, or role is allowed to execute on the ElastiCache service. Attaching the IAM policy to IAM users, groups, or roles controls which entity can use the policy to configure an ElastiCache cluster.

You can get a complete list of IAM actions and resource-level permissions supported at <http://mng.bz/anNF>.

12.3.2 Controlling network access

Network access is controlled with security groups. Remember the security group from the minimal CloudFormation template in section 12.1 where access to port 6379 (Redis) was allowed for all IP addresses. But since cluster nodes only have private IP addresses this restricts access to the VPC:


```
Resources:
  # [...]
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
```

```
VpcId: !Ref VPC
SecurityGroupIngress:
- IpProtocol: tcp
  FromPort: 6379
  ToPort: 6379
  CidrIp: '0.0.0.0/0'
```

You should improve this setup by working with two security groups. To control traffic as tight as possible, you will not white list IP addresses. Instead, you create two security groups. The client security group will be attached to all EC2 instances communicating with the cache cluster (your web servers). The cache cluster security group allows inbound traffic on port 6379 only for traffic that comes from the client security group. This way you can have a dynamic fleet of clients who is allowed to send traffic to the cache cluster. You used the same approach for the SSH bastion host in section 6.4.

```
Resources:
# [...]
ClientSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'cache-client'
    VpcId: !Ref VPC
CacheSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: cache
    VpcId: !Ref VPC
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
        ToPort: 6379
        SourceSecurityGroupId: !Ref ClientSecurityGroup
```

**Only allow
access from the
ClientSecurityGroup.**



Attach the `ClientSecurityGroup` to all EC2 instances that need access to the cache cluster. This way, you only allow access to the EC2 instances that really need access.

Keep in mind that ElastiCache nodes always have private IP addresses. This means that you can't accidentally expose a Redis or Memcached cluster to the internet. You still want to use Security Groups to implement the principle of least privilege.

12.3.3 Controlling cluster and data access

Both Redis and Memcached support very basic authentication features. Amazon ElastiCache does support Redis AUTH for customers who also want to enable token-based authentication in addition to the security features Amazon ElastiCache provides. Redis AUTH is the security mechanism that open source Redis utilizes. Since the communication between the clients and the cluster is unencrypted, such an authentication would not improve security using open source engines. But, Amazon ElastiCache does offer encryption in transit with Redis 3.2.6.

Neither engine implements data access management. When connected to a key-value store, you get access to all data. This limitation relies on the underlying key-value stores, not in ElastiCache itself. In the next section, you'll learn how to use ElastiCache for Redis in a real-world application called Discourse.

12.4 Installing the sample application Discourse with CloudFormation

Small communities, like football clubs, reading circles, or dog schools benefit from having a place where members can communicate with each other. Discourse is open-source software for providing modern forums for your community. You can use it as a mailing list, discussion forum, long-form chat room, and more. It is written in Ruby using the Rails framework. Figure 12.8 gives you an impression of Discourse. Wouldn't that be a perfect place for your community to meet? In this section, you will learn how to set up Discourse with CloudFormation. Discourse is also perfectly suited for learning about ElastiCache because it requires a Redis cache. Discourse requires PostgreSQL as main data store and uses Redis to cache data and process transient data.

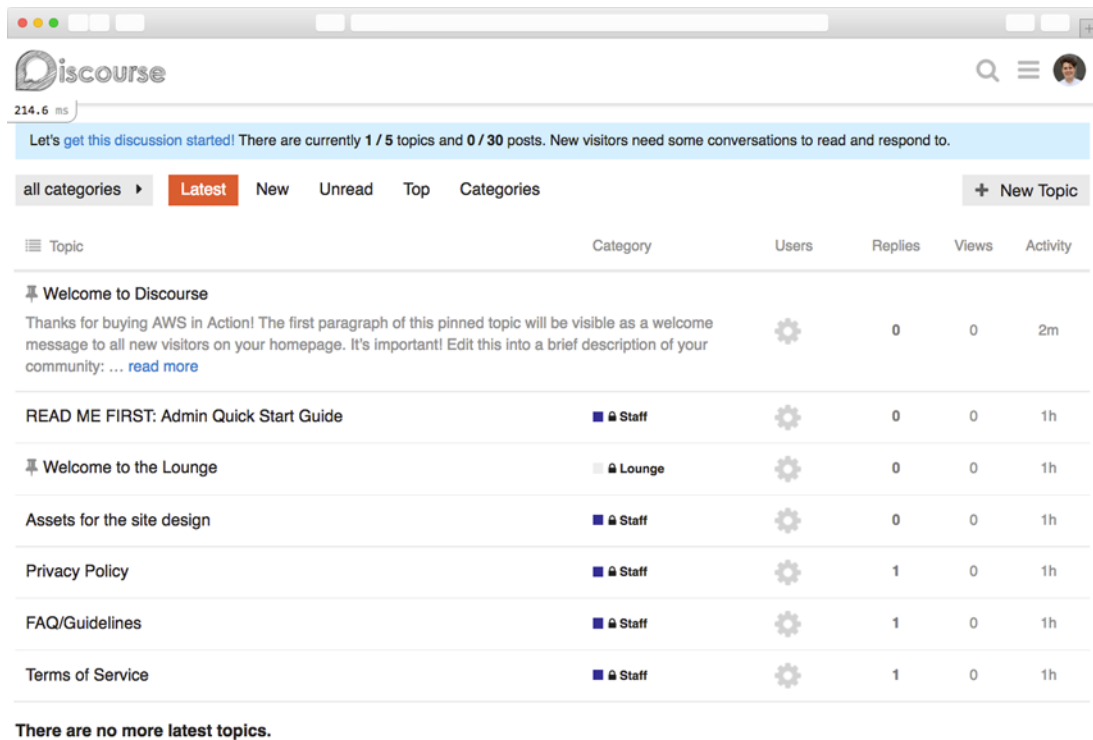


Figure 12.8 Discourse: a platform for community discussion

In this section, you'll create a CloudFormation template with all the components necessary to run Discourse. Finally, you'll create a CloudFormation stack based on the template to test your work. The necessary components are:

- *VPC*—Network configuration
- *Cache*—Security group, subnet group, cache cluster
- *Database*—Security group, subnet group, database instance
- *Virtual machine*—Security group, EC2 instance

Let's get started. You'll start with the first component and extend the template in the rest of this section.

12.4.1 VPC: Network configuration

In section 6.5 you learned all about private networks on AWS. If you can't follow listing 12.2, you could go back to section 6.5 or continue with the next step—understanding the network is not key to get Discourse running.

Listing 12.2 CloudFormation template for Discourse: VPC

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 12'
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
  AdminEmailAddress:
    Description: 'Email address of admin user'
    Type: 'String'
Resources:
  VPC:
    Type: 'AWS::EC2::VPC'
    Properties:
      CidrBlock: '172.31.0.0/16'
      EnableDnsHostnames: true
  InternetGateway:
    Type: 'AWS::EC2::InternetGateway'
    Properties: {}
  VPCGatewayAttachment:
    Type: 'AWS::EC2::VPCGatewayAttachment'
    Properties:
      VpcId: !Ref VPC
      InternetGatewayId: !Ref InternetGateway
  SubnetA:
    Type: 'AWS::EC2::Subnet'
    Properties:
      AvailabilityZone: !Select [0, !GetAZs '']
      CidrBlock: '172.31.38.0/24'
      VpcId: !Ref VPC
  SubnetB: # [...]
```

The key pair name for SSH access and also the email address of the Discourse admin (must be valid!) are variable.

Creates a VPC in the address range 172.31.0.0/16

We want to access Discourse from the internet, so we need an internet gateway.

Attach the internet gateway to the VPC.

Create a subnet in the address range 172.31.38.0/24 in the first availability zone (array index 0).

Create a second subnet in the address range 172.31.37.0/24 in the second availability zone (properties omitted).

```

RouteTable:
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
SubnetRouteTableAssociationA:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetA
    RouteTableId: !Ref RouteTable
RouteToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTable
    DestinationCidrBlock: '0.0.0.0/0'
    GatewayId: !Ref InternetGateway
  DependsOn: VPCGatewayAttachment
SubnetRouteTableAssociationB: # [...]
NetworkAcl:
  Type: AWS::EC2::NetworkAcl
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationA:
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetA
    NetworkAclId: !Ref NetworkAcl
SubnetNetworkAclAssociationB: # [...]
NetworkAclEntryIngress:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAcl
    RuleNumber: 100
    Protocol: -1
    RuleAction: allow
    Egress: false
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryEgress:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAcl
    RuleNumber: 100
    Protocol: -1
    RuleAction: allow
    Egress: true
    CidrBlock: '0.0.0.0/0'

```

← Create a route table that contains the default route, which routes all subnets in a VPC.

← Associate the first subnet with the route table.

← Add a route to the internet via the internet gateway.

← Create an empty network ACL.

← Associate the first subnet with the network ACL.

← Allow all incoming traffic on the Network ACL (you will use security groups later as a firewall).

← Allow all outgoing traffic on the Network ACL.

The network is now properly configured using two public subnets. Let's configure the cache next.

12.4.2 **Cache: Security group, subnet group, cache cluster**

You will add the ElastiCache for Redis cluster now. You learned how to describe a minimal cache cluster earlier in this chapter. This time, you'll add a few extra properties

to enhance the setup. This listing contains the CloudFormation resources related to the cache.

Listing 12.3 CloudFormation template for Discourse: Cache

```
Resources:
  # [...]
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
  CacheSecurityGroupIngress:
    Type: 'AWS::EC2::SecurityGroupIngress'
    Properties:
      GroupId: !Ref CacheSecurityGroup
      IpProtocol: tcp
      FromPort: 6379
      ToPort: 6379
      SourceSecurityGroupId: !Ref VMSecurityGroup
  CacheSubnetGroup:
    Type: 'AWS::ElasticCache::SubnetGroup'
    Properties:
      Description: cache
      SubnetIds:
        - Ref: SubnetA
        - Ref: SubnetB
  Cache:
    Type: 'AWS::ElasticCache::CacheCluster'
    Properties:
      CacheNodeType: 'cache.t2.micro'
      CacheSubnetGroupName: !Ref CacheSubnetGroup
      Engine: redis
      EngineVersion: '3.2.4'
      NumCacheNodes: 1
      VpcSecurityGroupIds:
        - !Ref CacheSecurityGroup
```

Redis runs on port 6379.

The security group to control incoming and outgoing traffic to/from the cache

To avoid a cyclic dependency, the ingress rule is split into a separate CloudFormation resource.

The VMSecurityGroup resource is not yet specified; you will add this later when you define the EC2 instance that runs the web server.

The cache subnet group references the VPC subnets.

Create a single-node Redis cluster.

You can specify the exact version of Redis that you want to run. Otherwise the latest version is used, which may cause incompatibility issues in the future. We recommend always specifying the version.

The single-node Redis cache cluster is now defined. Discourse also requires a PostgreSQL database, which you'll define next.

12.4.3 Database: Security group, subnet group, database instance

PostgreSQL is a powerful, open source, and relational database. If you are not familiar with PostgreSQL, that's not a problem at all. Luckily, the RDS service will provide a managed PostgreSQL database for you. You learned about RDS in chapter 11. Listing 12.4 shows the section of the template that defines the RDS instance.

Listing 12.4 CloudFormation template for Discourse: Database

```

Resources:
  # [...]
  DatabaseSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: database
      VpcId: !Ref VPC
  DatabaseSecurityGroupIngress:
    Type: 'AWS::EC2::SecurityGroupIngress'
    Properties:
      GroupId: !Ref DatabaseSecurityGroup
      IpProtocol: tcp
      FromPort: 5432
      ToPort: 5432
      SourceSecurityGroupId: !Ref VMSecurityGroup
  DatabaseSubnetGroup:
    Type: 'AWS::RDS::DBSubnetGroup'
    Properties:
      DBSubnetGroupDescription: database
      SubnetIds:
        - Ref: SubnetA
        - Ref: SubnetB
  Database:
    Type: 'AWS::RDS::DBInstance'
    Properties:
      AllocatedStorage: 5
      BackupRetentionPeriod: 0
      DBInstanceClass: 'db.t2.micro'
      DBName: discourse
      Engine: postgres
      EngineVersion: '9.5.6'
      MasterUsername: discourse
      MasterUserPassword: discourse
      VPCSecurityGroups:
        - !Sub ${DatabaseSecurityGroup.GroupId}
      DBSubnetGroupName: !Ref DatabaseSubnetGroup
      DependsOn: VPCGatewayAttachment

```

Traffic to/from the RDS instance is protected by a security group.

The VMSecurityGroup resource is not yet specified; you'll add this later when you define the EC2 instance that runs the web server.

PostgreSQL runs on port 5432 by default.

RDS also uses a subnet group to reference the VPC subnets.

The database resource

RDS created a database for you in PostgreSQL.

Disable backups; you want to turn this on (value > 0) in production.

Discourse requires PostgreSQL.

We recommend to always specify the version of the engine to avoid future incompatibility issues.

PostgreSQL admin password; you want to change this in production.

PostgreSQL admin user name

Have you noticed the similarity between RDS and ElastiCache? The concepts are similar, which makes it easier for you to work with both services. Only one component is missing: the EC2 instance that runs the web server.

12.4.4 Virtual machine—security group, EC2 instance

Discourse is a Ruby on Rails application so you need an EC2 instance to host the application. Listing 12.5 defines the virtual machine and the startup script to install and configure Discourse.

Listing 12.5 CloudFormation template for Discourse: Virtual machine

```

Resources:
  # [...]
  VMSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'vm'
      SecurityGroupIngress:
        - CidrIp: '0.0.0.0/0'
          FromPort: 22
          IpProtocol: tcp
          ToPort: 22
        - CidrIp: '0.0.0.0/0'
          FromPort: 80
          IpProtocol: tcp
          ToPort: 80
      VpcId: !Ref VPC
  VMInstance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: 't2.micro'
      KeyName: !Ref KeyName
      NetworkInterfaces:
        - AssociatePublicIpAddress: true
          DeleteOnTermination: true
          DeviceIndex: 0
          GroupSet:
            - !Ref VMSecurityGroup
          SubnetId: !Ref SubnetA
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -x
          bash -ex << "TRY"
          # [...]

          # download Discourse
          useradd discourse
          mkdir /opt/discourse
          git clone https://github.com/AWSinAction/discourse.git \
            &➤ /opt/discourse

          # configure Discourse
          echo "db_host = \"${Database.Endpoint.Address}\"" >> \
            &➤ /opt/discourse/config/discourse.conf
          echo "redis_host = \"${Cache.RedisEndpoint.Address}\"" >> \
            &➤ /opt/discourse/config/discourse.conf
          # [...]
          TRY
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
            &➤ --resource VMInstance --region ${AWS::Region}
      CreationPolicy:
        ResourceSignal:

```

Allow SSH traffic from the public internet.

Allow HTTP traffic from the public internet.

The virtual machine that runs Discourse

Only contains an excerpt of the full script necessary to install Discourse. You can find the full code at /chapter12/template.yaml in the book's code folder.

Download Discourse.

Configure the Redis cluster node endpoint.

Configure the PostgreSQL database endpoint.

Signal the end of the installation script back to CloudFormation.

```

    Timeout: PT15M
    DependsOn:
      - VPCGatewayAttachment
Outputs:
  VMInstanceIPAddress:
    Value: !GetAtt 'VMInstance.PublicIp'
    Description: 'EC2 Instance public IP address'
➡ (connect via SSH as user ec2-user)'

```

← Wait up to 15 minutes for the signal from the install script in UserData.

← Output the public IP address of the virtual machine.

You've reached the end of the template. All components are defined now. It's time to create a CloudFormation stack based on your template to see if it works.

12.4.5 Testing the CloudFormation template for Discourse

Let's create a stack based on your template to create all the resources in your AWS account. To find the full code for the template, go to `/chapter12/template.yaml` in the book's code folder. Use the AWS CLI to create the stack:

```

$ aws cloudformation create-stack --stack-name discourse \
➡ --template-url https://s3.amazonaws.com/awsinaction-code2/\
➡ chapter12/template.yaml \
➡ --parameters ParameterKey=KeyName,ParameterValue=mykey \
➡ "ParameterKey=AdminEmailAddress,ParameterValue=your@mail.com"

```

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at <https://github.com/AWSinAction/code2/archive/master.zip>. The file we're talking about is located at `chapter12/template.yaml`. On S3, the same file is located at <http://mng.bz/jP32>.

The creation of the stack can take up to 15 minutes. To check the status of the stack, use the following command:

```

$ aws cloudformation describe-stacks --stack-name discourse \
➡ --query "Stacks[0].StackStatus"

```

If the stack status is `CREATE_COMPLETE`, the next step is to get the public IP address of the EC2 instance from the stack's outputs with the following command:

```

$ aws cloudformation describe-stacks --stack-name discourse \
➡ --query "Stacks[0].Outputs[0].OutputValue"

```

Open a web browser and insert the IP address in the address bar to open your Discourse website. Figure 12.9 shows the website. Click **Register** to create an admin account.

You will receive an email to activate your account. This email will likely be in your spam folder! After activation, the 13-step setup wizard is started, which you have to

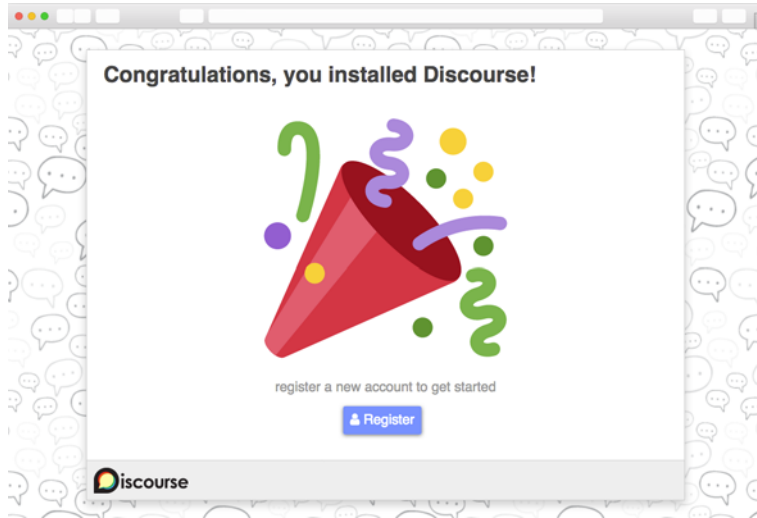


Figure 12.9
Discourse: first
screen after a
fresh install

complete. After you complete the wizard and have successfully installed Discourse, the screen shown in figure 12.10 should appear.

Topic	Category	Users	Replies	Views	Activity
Welcome to Discourse Thanks for buying AWS in Action! The first paragraph of this pinned topic will be visible as a welcome message to all new visitors on your homepage. It's important! Edit this into a brief description of your community: ... read more		⚙️	0	0	2m
READ ME FIRST: Admin Quick Start Guide	🔒 Staff	⚙️	0	0	1h
Welcome to the Lounge	🏠 Lounge	⚙️	0	0	1h
Assets for the site design	🔒 Staff	⚙️	0	0	1h
Privacy Policy	🔒 Staff	⚙️	1	0	1h
FAQ/Guidelines	🔒 Staff	⚙️	1	0	1h
Terms of Service	🔒 Staff	⚙️	1	0	1h

There are no more latest topics.

Figure 12.10 Discourse: a platform for community discussion

Don't delete the CloudFormation stack because you'll use the setup in the next section.

You learned how Discourse uses Redis to cache some data while using ElastiCache to run Redis for you. If you use ElastiCache in production, you also have to set up monitoring to ensure that the cache works as expected, and you have to know how to improve performance. Those are the topics of the next two chapters.

12.5 Monitoring a cache

CloudWatch is the service on AWS that stores all kinds of metrics. ElastiCache nodes send useful metrics. The most important metrics to watch are:

- **CPUUtilization**—The percentage of CPU utilization.
- **SwapUsage**—The amount of swap used on the host, in bytes. *Swap* is space on disk that is used if the system runs out of physical memory.
- **Evictions**—The number of non-expired items the cache evicted due to the memory limit.
- **ReplicationLag**—This metric is only applicable for a Redis node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. Usually this number is very low.

In this section we'll examine those metrics in more detail, and give you some hints about useful thresholds for defining alarms on those metrics to set up production-ready monitoring for your cache.

12.5.1 Monitoring host-level metrics

The virtual machines report CPU utilization and swap usage. CPU utilization usually gets problematic when crossing 80–90%, because the wait time explodes. But things are more tricky here. Redis is single-threaded. If you have many cores, the overall CPU utilization can be low but one core can be at 100% utilization. Swap usage is a different topic. You run an in-memory cache, so if the virtual machine starts to swap (move memory to disk) the performance will suffer. By default, ElastiCache for Memcached and Redis is configured to limit memory consumption to a value smaller than what's physical available (you can tune this) to have room for other resources (for example, the kernel needs memory for each open socket). But other processes (such as kernel processes) are also running, and they may start to consume more memory than what's available. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards.

Queuing theory: why 80–90%?

Imagine you are the manager of a supermarket. What should be the goal for daily utilization of your cashier? It's tempting to go for a high number. Maybe 90%. But it turns out that the wait time for your customers is very high when your cashiers are utilized for 90% of the day, because customers don't arrive at the same time at the queue.

The theory behind this is called *queuing theory*, and it turns out that wait time is exponential to the utilization of a resource. This not only applies to cashiers, but also to network cards, CPU, hard disks, and so on. Keep in mind that this sidebar simplifies the theory and assumes an M/D/1 queuing system: Markovian arrivals (exponentially distributed arrival times), deterministic service times (fixed), one service center. To you want to learn more about queuing theory applied to computer systems, we recommend *Systems Performance: Enterprise and the Cloud* by Brendan Gregg (Prentice Hall, 2013) to get started.

When you go from 0% utilization to 60%, wait time doubles. When you go to 80%, wait time has tripled. When you to 90%, wait time is six times higher. And so on.

So if your wait time is 100 ms during 0% utilization, you already have 300 ms wait time during 80% utilization, which is already slow for a e-commerce web site.

You might set up an alarm to trigger if the 10-minute average of the `CPUUtilization` metric is higher than 80% for 1 out of 1 data points, and if the 10-minute average of the `SwapUsage` metric is higher than 67108864 (64 MB) for 1 out of 1 datapoints. These numbers are just a rule of thumb. You should load-test your system to verify that the thresholds are high/low enough to trigger the alarm before application performance suffers.

12.5.2 Is my memory sufficient?

The `Evictions` metric is reported by Memcached and Redis. If the cache is full and you put a new key-value pair into the cache, an old key-value pair needs to be deleted first. This is called an eviction. Redis only evicts keys with a TTL by default (volatile-lru). These additional eviction strategies are available: allkeys-lru (remove the least recently used key among all keys), volatile-random (remove a random key among keys with TTL), allkeys-random (remove a random key among all keys), volatile-ttl (remove the key with the shortest TTL), and noeviction (do not evict any key). Usually, high eviction rates are a sign that you either aren't using a TTL to expire keys after some time, or that your cache is too small. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards.

You might set an alarm to trigger if the 10-minute average of the `Evictions` metric is higher than 1000 for 1 out of 1 data points.

12.5.3 Is my Redis replication up-to-date?

The `ReplicationLag` metric is only applicable for a node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. The higher this value, the more out-of-date the replica is. This can be a problem because some users of your application will see very old data. In the gaming application, imagine you have one primary node and one replica node. All reads are performed by either the primary or the replica node. The `ReplicationLag` is 600,

which means that the replication node looks like the primary node looked 10 minutes before. Depending on which node the user hits when accessing the application, they could see 10-minute old data.

What are reasons for a high ReplicationLag? There could be a problem with the sizing of your cluster; for example, your cache cluster might be at capacity. Typically this will be a sign to increase the capacity by adding shards or replicas.

You might set an alarm to trigger if the 10-minute average of the ReplicationLag metric is higher than 30 for 1 consecutive period.



Cleaning up

It's time to delete the running CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name discourse
```

12.6 Tweaking cache performance

Your cache can become a bottleneck if it can no longer handle the requests with low latency. In the previous section, you learned how to monitor your cache. In this section you learn what you can do if your monitoring data shows that your cache is becoming the bottleneck (for example if you see high CPU or network usage). Figure 12.11 contains a decision tree that you can use to resolve performance issues with ElastiCache. The strategies are described in more detail in the rest of this section.

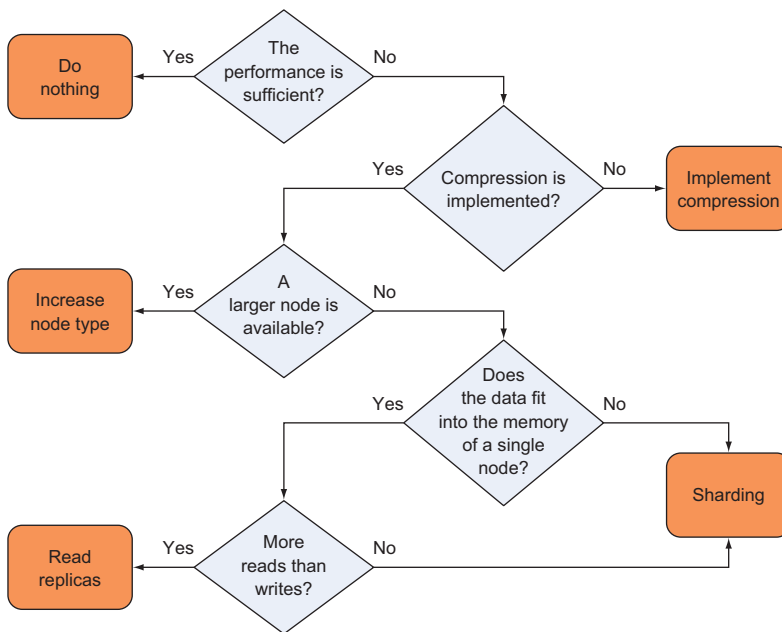


Figure 12.11 ElastiCache decision tree to resolve performance issues

There are three strategies for tweaking the performance of your ElastiCache cluster:

- 1 *Selecting the right cache node type*—A bigger instance type comes with more resources (CPU, memory, network) so you can scale vertically.
- 2 *Selecting the right deployment option*—You can use sharding or read replicas to scale horizontally.
- 3 *Compressing your data*—If you shrink the amount of data being transferred and stored, you can also tweak performance.

12.6.1 Selecting the right cache node type

So far, you used the cache node type `cache.t2.micro`, which comes with one vCPU, ~0.6 GB memory, and low-to-moderate network performance. You used this node type because it's part of the Free Tier. But you can also use more powerful node types on AWS. The upper end is the `cache.r4.16xlarge` with 64 vCPUs, ~488 GB memory, and 25 Gb network. Keep in mind that Redis is single-threaded and will not use all cores.

As a rule of thumb: for production traffic, select a cache node type with at least 2 vCPUs for real concurrency, enough memory to hold your data set with some space to grow (say, 20%; this also avoids memory fragmentation), and at least high network performance. The `r4.large` is an excellent choice for a small node size: 2 vCPUs, ~16 GB, and up to 10 Gb of network. This may be a good starting point when considering how many shards you may want in a clustered topology, and if you need more memory, move up a node type. You can find the available node types at <https://aws.amazon.com/elasticache/pricing/>.

12.6.2 Selecting the right deployment option

By replicating data, you can distribute read traffic to multiple nodes within the same replica group. Since you have more nodes in your cluster, you can serve more requests. By sharding data, you split the data into multiple buckets. Each bucket contains a subset of the data. Since you have more nodes in your cluster, you can serve more requests.

You can also combine replication and sharding to increase the number of nodes in your cluster.

Both Memcached and Redis support the concept of sharding. With sharding, a single cache cluster node is no longer responsible for all the keys. Instead the key space is divided across multiple nodes. Both Redis and Memcached clients implement a hashing algorithm to select the right node for a given key. By sharding, you can increase the capacity of your cache cluster.

Redis supports the concept of replication, where one node in a node group is the primary node accepting read and write traffic, while the replica nodes only accept read traffic. This allows you to scale the read capacity. The Redis client has to be aware of the cluster topology to select the right node for a given command. Keep in mind that the replicas are synchronized asynchronously. This means that the replication node eventually reaches the state of the primary node.

As a rule of thumb: when a single node can no longer handle the amount of data or the requests, and if you are using Redis with mostly read traffic, then you should use replication. Replication also increases the availability at the same time (at no extra costs).

12.6.3 Compressing your data

This solution needs to be implemented in your application. Instead of sending large values (and also keys) to your cache, you can compress the data before you store it in the cache. When you retrieve data from the cache, you have to uncompress it on the application before you can use the data. Depending on your data, compressing data can have a significant effect. We saw memory reductions to 25% of the original size and network transfer savings of the same size.

As a rule of thumb: Compress your data using a compression algorithm that is best suited for your data, most likely the zlib library. You have to experiment with a subset of your data to select the best compression algorithm that is also supported by your programming language.

Summary

- A caching layer can speed up your application significantly, while also lowering the costs of your primary data store.
- To keep the cache in sync with the database, items usually expire after some time, or a write-through strategy is used.
- When the cache is full, the least frequently used items are usually evicted.
- ElastiCache can run Memcached or Redis clusters for you. Depending on the engine, different features are available. Memcached and Redis are open source, but AWS added engine-level enhancements.

Amazon Web Services IN ACTION Second Edition

Michael Wittig • Andreas Wittig

The largest and most mature of the cloud platforms, AWS offers over 100 prebuilt services, practically limitless compute resources, bottomless secure storage, as well as top-notch automation capabilities. This book shows you how to develop, host, and manage applications on AWS.

Amazon Web Services in Action, Second Edition is a comprehensive introduction to deploying web applications in the AWS cloud. You'll find clear, relevant coverage of all essential AWS services, with a focus on automation, security, high availability, and scalability. This thoroughly revised edition covers the latest additions to AWS, including serverless infrastructure with AWS Lambda, sharing data with EFS, and in-memory storage with ElastiCache.

What's Inside

- Completely revised bestseller!
- Secure and scale distributed applications
- Deploy applications on AWS
- Design for failure to achieve high availability
- Automate your infrastructure

Written for mid-level developers and DevOps engineers.

Andreas and **Michael Wittig** are software engineers and DevOps consultants focused on AWS. Together, they migrated the first bank in Germany to AWS in 2013.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/amazon-web-services-in-action-second-edition

“Slices through the complexity of AWS using examples and visuals to cement knowledge in the minds of readers.”

—From the Foreword by
Ben Whaley
AWS community hero and author

“The authors’ ability to explain complex concepts is the real strength of the book.”

—Antonio Pessolano
Consoft Sistemi

“Useful examples, figures, and sources to help you learn efficiently.”

—Christof Marte, Daimler-Benz

“Does a great job of explaining some of the key services in plain English so you have the knowledge necessary to dig deeper.”

—Ryan Burrows
Rooster Park Consulting



ISBN-13: 978-1-61729-511-9
 ISBN-10: 1-61729-511-6



9 781617 295119



MANNING

\$54.99 / Can \$72.99 [INCLUDING eBook]